Adaptive Microburst Control Techniques in Incast-Heavy Datacenter Networks

by

Hamed Rezaei
M.S., University of Qom, Iran, 2014
B.S., Payame-Noor University, Iran, 2012

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2021

Chicago, Illinois

Defense Committee:
Prof. Balajee Vamanan, Chair and Advisor
Prof. Venkatakrishnan Venkatesan Natarajan
Prof. Jakob Eriksson
Prof. Mark Grechanik
Prof. Hulya Seferoglu (UIC ECE department)

ACKNOWLEDGMENTS

To my wife, Sanam, who stayed by my side in every single second of this mission, and to my parents, who are my two flying wings ...

HR

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (Continued)

FIGURE **PAGE**

LIST OF ABBREVIATIONS

RTT	Round Trip Time
ECN	Explicit Congestion Notification
RTO	Retransmission Timeout
FIFO	First In First Out
AQM	Active Queue Management
FCT	Flow Completion Time
ACK	Acknowledgment
EWMA	Exponential Weighted Moving Average
ToR	Top of Rack
ToS	Type of Service
VM	Virtual Machine
TCP	Transmission Control Protocol
SJF	Shortest Job First
CDF	Cumulative Distribution Function
ECMP	Equal Cost Multiple Paths
RR	Round Robin
SRPT	Shortest Remaining Processing Time

LIST OF ABBREVIATIONS (Continued)

MLFQ	Multiple Level Feedback Queue
OSPF	Open Shortest Path First
LUT	Lookup Table
SRAM	Static Random Access Memory
TM	Traffic Manager
OLDI	Online Data Intensive

SUMMARY

Datacenters host a mix of applications which generate qualitatively distinct traffic patterns and impose varying network objectives. Online, user-facing applications generate many-to-one, incast traffic of mostly short flows, which are sensitive to tail of Flow Completion Times (FCT). Data analytics applications generate all-to-all traffic (e.g., Web search) of mostly short flows that saturate network bisection and the job completions require all flows to complete. Background applications (e.g., Map-Reduce) generate large flows and are throughput sensitive due to the sheer amount of data that they transfer over the network. While datacenter fabric provides good bisection bandwidth to handle all-to-all traffic and background traffic, incast traffic is bottle-necked at edge switches and causes queue buildup at the switch port connected to the receiver server. Incasts are synchronized bursts of many-to-one short flows that fundamentally cause an over-subscription of the receiver (aggregator) link. Because datacenter switches use shallow buffers to reduce cost and latency, the queue buildup problem is further exacerbated as the shallow buffers easily overflow causing packet drops and expensive TCP timeouts.

To address incast problem in datacenter networks, we proposed five solutions. Slytherin is the first discussed method, which is a novel flow scheduling scheme that targets those packets that fall in the tail (i.e., those that are delayed at multiple switches) and prioritizes them in the next hop switches. Therefore, most delayed packets are drained faster, and also, congestion report arrives faster at the receiver server as well. ICON is our next discussed method, which is a novel scheme that reduces incast-induced packet loss by setting a fine-grained control over

sending rate by pacing traffic. ResQueue is the third discussed scheme that uses a combination of *flow size* ***and*** *packet history* to calculate the priority of each flow. ResQueue detects those packets that were dropped before (possibly during an incast) and then increases their priority in the next round. Therefore, these packets will not be dropped again even in severe cases of congestion. Our evaluation shows that ResQueue improves tail flow completion times of short flows by up to 60% over the state-of-the-art flow scheduling mechanisms. Superways, the fourth discussed method, is a heterogeneous datacenter topology that provides higher bandwidth for for those servers that host incast application to absorb incasts, as incasts occur only at a small number of servers that aggregate responses from other senders. Our design is based on the key observation that a small subset of servers which aggregate responses are likely to be network bound, whereas most other servers that communicate only with random servers are not. Superways guarantees 95% improvement (on average) in tail latency when it is implemented on top of state-of-the-art topologies in which the servers run DCTCP as congestion control method. Finally, we will discuss Smartbuf, which is an online learning algorithm that accurately predicts buffer requirement of each switch port *before* the onset of congestion. Our key novelty lies in fingerprinting bursts based on the gradient of queue length and using this information to provision *just enough* buffer space. Our preliminary evaluations show that our algorithm can predict buffer demands accurately within an average error margin of 6% and achieve an improvement in the 99^{th} percentile latency by a factor of 8x at high loads, while providing good fairness among ports.

CHAPTER 1

INTRODUCTION TO DATACENTER NETWORKS

Datacenters have emerged as the de facto platform for hosting user facing applications that query vast amounts of Internet data (e.g., Web search) [1, 2]. In this section, we provide a brief overview of datacenters' characteristics such as architecture, flow characteristics, and incast type traffic pattern.

1.1 Datacenter architecture

Most datacenters' network design is based on a layered architecture (e.g., Fat-tree, Spine-Leaf, etc), which has been evaluated and improved over the past few years. The layered approach is the basic foundation of the datacenter design that seeks to improve scalability, flexibility, and maintenance. Figure 1 shows the basic tree topology. In this architecture, the links that connect aggregation layer to core layer are more powerful in terms of bandwidth in contrast to other links such as links from edge switches (a.k.a ToR switches) to endhosts. This is because of the higher load at the upper layers of the network compared to the lower layers (e.g., edge switches). Each of these layers are described as follows:

Core layer — Provides the high-speed packet switching backbone for all flows travelling across the datacenter. This layer provides connectivity to multiple aggregation layer switches and provides a high bandwidth fabric (layer 3) with no single point of failure [3].

Aggregation layer — Provides important functions, such as service module integration, span-

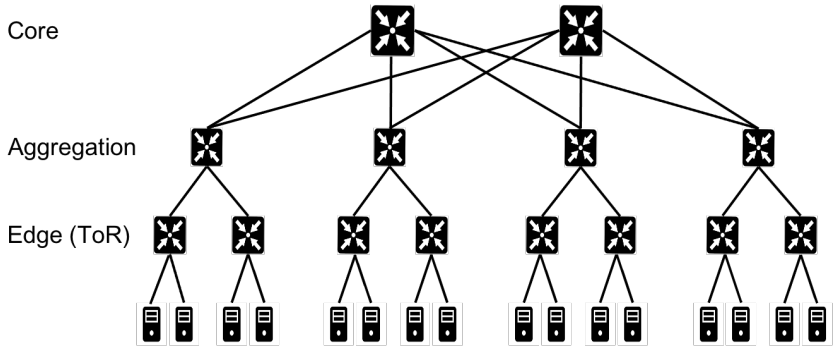

Figure 1. An example of tree topology in datacenter networks

ning tree processing, and default gateway redundancy. Server-to-server multiple-tier traffic flows through the aggregation layer and can use services, such as firewall and server load balancing, to optimize and secure applications. In large datacenter networks, many of aggregate switches are clustered in a "pod" to provide more granularity.

Edge layer — Where the servers physically connect to the network. The edge layer network infrastructure is not as powerful as core or aggregate switches.

The above named layers are part of Fat-Tree topology terminology. In spine-Leaf topologies, spine layer is the mix of core and aggregate layers and leaf switch is called edge switch in a Fat-Tree topology.

1.2 Datacenter applications and traffic characteristics

Modern datacenters host various applications including user-facing *foreground* applications that query large amounts of Internet data (e.g., Web Search) and *background* applications that perform background tasks such as data reorganization, replication, and backup [1, 2].

The nature of these two broad categories of applications determine the traffic dynamics and objectives of the underlying datacenter network. Foreground applications such as Web Search access data that is distributed among hundreds of servers. Every Web Search query must wait for a response from most of the worker servers (e.g., 99% of servers) to achieve a good trade-off between query response time and result quality. Therefore, foreground applications' performance is sensitive to higher percentiles (i.e., 99^{th} to 99.9^{th}) of Flow Completion Times (FCT) [4]. Google search is a good example of such traffic that foreground applications produce. While foreground applications generate mostly short flows (e.g., 1KB to 10KB) [5], background applications transfer large amounts of data (e.g., 1MB <) [6] across the network, which requires high throughput [6]. Hadoop is a good example of such applications that produce background traffic. A well-designed datacenter network must provide low tail flow completion times for short flows and high throughput for long flows.

Achieving this goal is challenging because both short flows (from foreground applications) and large flows (from background applications) compete for network bandwidth at switches. It is likely for short flows to get stuck behind several long flows and suffer elongated tail FCTs. Similarly, long flows incur packet loss from competing bursts of short flow packets and lose throughput. Additionally, modern switches are equipped with flow classification techniques, which normally schedule long flows behind short flows (or flows with earliest deadline such as [7]), and therefore, long flows may suffer starvation in extreme cases (e.g., a highly loaded network, which is full of bursty short flows).

In summary, we list datacenter network traffic characteristics as follows:

- We have a wide variety of applications across the datacenters, ranging from customer-facing applications, such as Web services, file stores, and custom enterprise applications to data intensive applications, such as MapReduce and Web-search indexing.
- Most flows in the datacenters are small in size (typically less than 10KB), a significant fraction of which last under a few hundreds of microseconds. In user-facing applications, these packets arrive all at a time, which creates a sudden burst at one port of the edge switch connected to the receiver server (i.e., incast).
- There is no evidence that shows traffic is rack local in datacenter networks [8].
- Losses occur within the datacenters; however, losses are not localized to links with persistently high utilization. Instead, losses occur at links with low average utilization implicating momentary spikes as the primary cause of losses (a.k.a incast).
- Packet drop rate is higher at edge switches compared to other switches.

1.3 Incast

Because foreground applications concurrently fetch data from a large number of servers, they cause synchronized responses from servers that cause severe queue buildup at the receivers' switch port. This phenomenon, called *incast*, is known to dilate tail FCTs (i.e., degrade response times) and cause packet drops (i.e., reduce goodput) [9, 10]. Figure 2 depicts a schematic view of this problem.

Existing approaches alleviate incast by employing clever techniques in congestion control [7, 10–14], packet scheduling [6, 15–17], and load balancing [18–20]. Even with precise feedback

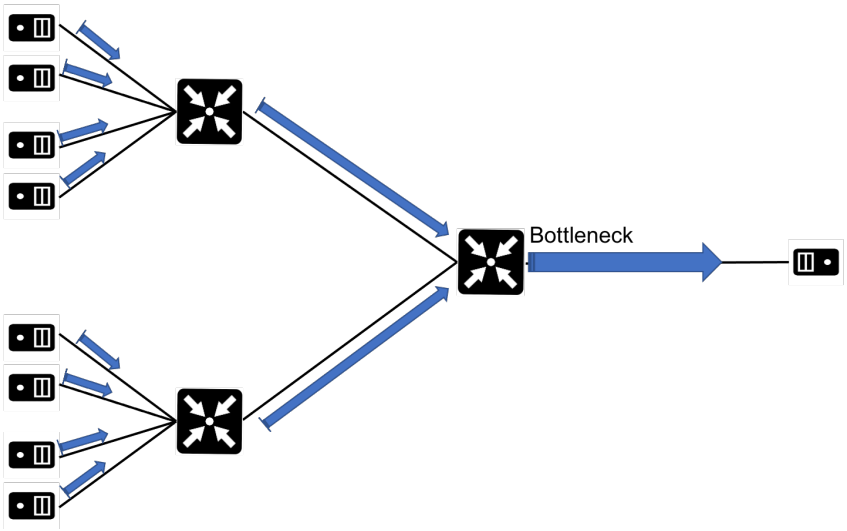

Figure 2. An example of incast in datacenter networks

that would enable us to accurately detect incast and state-of-the-art algorithms for setting the rates, congestion control approaches require at least one round-trip time (RTT) to optimally respond to incast. However, recent measurement studies on datacenters show that the duration of most incasts is shorter than typical RTTs (i.e., 100–200 μs) in datacenter networks [5], which makes it extremely difficult for congestion control methods to respond to congestion in a timely manner. Packet scheduling approaches prioritize short flows (or flows with near deadlines [17]) ahead of long flows [6], but do not help when short, incast flows that have similar deadlines contend for limited buffer capacity, which occurs often. Similarly, load balancing approaches find the least congested path among all other paths to forward the packets through this route;

but since congestion happens at the last hop switch, there is little to no chance for load balancing methods to alleviate incast problem.

Datacenter switches use shallow buffers [21–23], and therefore, it is likely that short, incast flows overrun the port buffer. Today's incast-heavy applications (e.g., Web search, social networks) and high-bandwidth network topologies (e.g., fat-trees [24]) imply that receiver congestion is more severe than congestion in the network core, as reported by Facebook [8], Google [25], and Microsoft [26]. While most research proposals improve *all-to-all* throughput by providing high bisection bandwidth, which helps applications such as map-reduce [27], the previously proposed schemes do not alleviate incast. Incast is fundamentally a *many-to-one* over-subscription of the receiver link capacity that needs to be detected at the switch right before buffer overflow. Throughout this thesis, we discuss our solutions for incast problem in datacenter networks.

CHAPTER 2

RELATED WORK

NOTE: Some of the contents of this chapter have been published before in:

(1) Rezaei, Hamed, and Balajee Vamanan. "ResQueue: A Smarter Datacenter Flow Scheduler." Proceedings of The Web Conference 2020. 2020. 2599-2605

(2) Rezaei, Hamed, and Balajee Vamanan. "Superways: A Datacenter Topology for Incast-heavy workloads." Proceedings of the Web Conference 2021. 2021. 317-328

(3) Rezaei, Hamed, Malekpourshahraki, Mojtaba, and Balajee Vamanan. "Slytherin: Dynamic, network-assisted prioritization of tail packets in datacenter networks". 27th International Conference on Computer Communication and Networks. 2018. 1-9

(4) Rezaei, Hamed, Almasi, Hamidreza, and Balajee Vamanan. "Icon: Incast congestion control using packet pacing in datacenter networks". 11th International Conference on Communication Systems and Networks. 2019. 125-132

In this section, we discuss all possible solutions for incast in datacenter networks, their strengths, and their weaknesses. We first start with flow scheduling techniques.

2.1 Flow scheduling techniques

There are two types of flow schedulers in datacenter networks: (1) information agnostic flow schedulers and, (2) information aware flow schedulers. While information agnostic flow schedulers are desired (because flow information is difficult to obtain prior to flow transmission), information aware flow schedulers provide higher performance. Below we discuss these two methods.

2.1.1 Information-agnostic scheduling:

As we mentioned in section 2.1, these methods try to schedule packets while there is no prior knowledge about the flow characteristics like flow size or flow deadline. In these methods, the scheduling algorithm usually assigns different priorities to packets and then each packet will be assigned to a specific queue based on the given priority. As an instance, one of past proposals, PIAS [16], tries to leverage multiple priority queues to implement Multiple Level Feedback Queue (MLFQ), in which corresponding packets of a flow get gradually demoted from higher priority queues to lower ones based on bytes it has already sent [16].

The biggest advantage of information-agnostic scheduling schemes is ease of implementation which comes from not requiring prior hard-to-achieve information about the flows (e.g., flow size). Moreover, although they may add small complexities to switching fabric but since they do not rely on prior information about the flows, they can be implemented in real datacenter networks.

The disadvantage of information-agnostic scheduling schemes is their inability to improve *tail* flow completion times. This is because they schedule packets based on just limited information about the flow. More specifically, they try to treat each single packet (regardless of packet's history) based on a scheduling algorithm which can not discriminate among packets that suffered more queuing delay in previous hops (e.g., tail packets) and other packets. This is considered as a big problem because if packets of a flow experience different amounts of queuing delay at multiple hops, flow completion times increase, which is not tolerable in most datacenter networks.

In conclusion, current information-agnostic scheduling methods cannot improve tail flow completion times because of their inability to discriminate among congested packets and normal packets. In general, information-agnostic schemes provide lower performance in contrast to information-aware methods. For example, D^3 [28] shows that as much as 7% of flows may miss their deadlines with DCTCP [10] which is an information-agnostic scheme. However, as long as they do not require prior information about the flows, it is quite fair to not to expect them to provide as good performance as information-aware scheduling methods. We will discuss information-aware methods in the following section.

2.1.2 Information-aware scheduling:

Information-aware scheduling schemes provide higher performance by giving prior information about flows to servers and switches. In the other words, these schemes assume that switches know some key information about the flows like flow deadline, flow size or even flow remaining

processing time. This information is usually given to switches either by a central controller or by endhost applications. In general, we can divide these methods to two different groups:

2.1.2.1 Prior knowledge about flow deadlines

These methods assume flow deadlines all are known apriori and switches greedily schedule flows with nearest deadline ahead of others. Having comprehensive knowledge about flow deadlines guarantees that almost none of the deadlines are missed, which drastically improves the performance of short, deadline sensitive flows. Some of methods which use this approach are D^3 [28], D^2TCP [7], and Karuna [15].

Remember from section 1 that most of flows in datacenters are short and deadline sensitive. Furthermore, deadline aware schemes usually meet the requirements of vast fraction of flows in datacenters. However, although they have a big advantage over deadline-agnostic schemes, they still suffer from severe implementation issues. For many applications, such information (e.g., flow deadline) is difficult to obtain, and may even be unavailable [16].

2.1.2.2 Prior knowledge about flow sizes

These schemes assume flow sizes all are known to servers and switches. These methods try to emulate Shortest Job First (SJF), which is known to minimize average flow completion times. These methods greedily schedule shorter flows ahead of longer flows in their simplest form. In a more complicated form, they try to give higher priority to flows with shortest remaining size. Some of well-known schemes that use this approach are pFabric [6], PASE [29], and PDQ [30].

Although information-aware scheduling methods provide better performance in contrast to information-agnostic approaches, they try to use some information that are not easy to collect

in datacenters. Prior proposals argue that flow sizes(or deadlines) can be tagged on packets by end host applications or a central controller may provide this information to switches. In both cases, we need to add extensive complexities to end-hosts and switches, which is not only easy at all but even sometimes impossible.

As we saw, packet scheduling methods all suffer from either implementation issues or lack of smart scheduling decisions in case of scheduling congested tail packets. It turns out that most of current scheduling schemes are not effective enough or are very difficult to implement in real datacenters. Our biggest motivation is to provide a scheme which is both implementation friendly and smart enough to schedule mix flows while giving higher priorities to congested tail packets. Later in chapter 3 we introduce Slytherin, which is an information-agnostic method that not only improves tail flow completion times but even does not require any modifications to existing switches.

2.2 Load balancing techniques

There are two types of load balancing in datacenter networks: (1) static load balancing (e.g., ECMP), and (2) dynamic load balancing (e.g., CONGA [19]).

2.2.1 Static load balancing

Static load balancing methods select the best route regardless of the status (load) of the links. In other words, they use a fixed function to select the best path among a set of paths. ECMP (Equal Cost Multiple Paths) and RR (Round Robin) methods are well-known static load balancing methods. While ECMP uses a hash function to select the best path, RR randomly selects a path to send each flow. This is obvious that static load balancing techniques cannot

solve incast problem because blindly distribute the load among different ports. Also, we know from chapter 1 that incast happens at the last hop switch, which means no load balancing technique (including static) cannot solve incast.

2.2.2 Static load balancing

Dynamic flow scheduling techniques select the best path based on the online information that they receive from other switches. CONGA [19], Flow Bender [31], Presto [18], and HULA [20] are examples of dynamic load balancing methods.These methods all work well in datacenter networks because they *accurately* pick the best possible path based on the collected information from other switches. Although dynamic load balancing methods perform much better than static load balancing schemes, they still cannot eliminate incast because they do not provide higher bandwidth at the last hop switch that connects to receiver server.

2.3 Congestion control techniques

There has been a renewed interest in congestion control for datacenters over the last few years. Recent studies [8] on datacenter traffic reveal that most flows are extremely small. Because most congestion control schemes require multiple round trips to adjust rates, they are not effective when flows are very small. Also, proactive grant-based congestion control methods, which are not reactive, will elongate tail flow completions because they need to wait for grant messages for all flows (including those that have just one packet to send). Recently, a receiver-driven congestion control called Homa [32] has been proposed, which improves the tail latency compared to its predecessors. Homa approximates SRPT by scheduling packet transmissions from the receivers. In fact, Homa divides each flow to unscheduled and scheduled parts, and

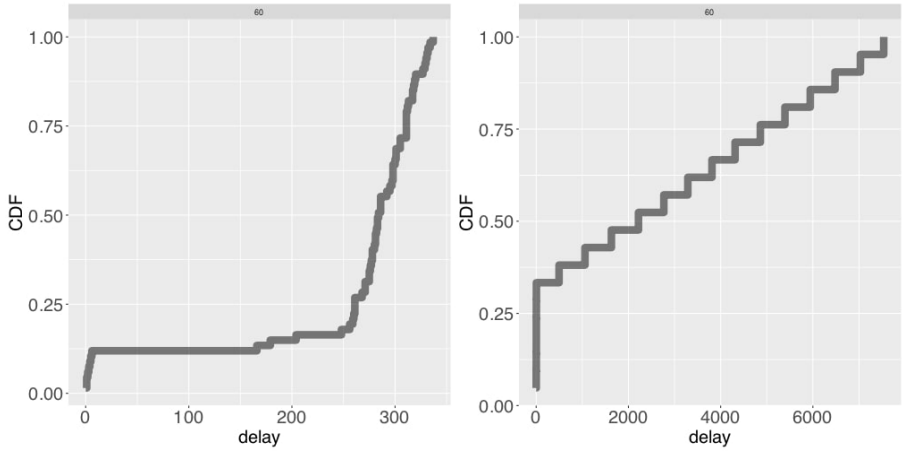

Figure 3. Homa's performance with and without incast of extremely short flows

sends the scheduled part if and only if the receiver asks to do so when it receives the unscheduled part. However, since datacenter flows are extremely small, it is difficult to divide them and almost all the data will be transmitted as a unscheduled (i.e., no grant needed) data portion. Therefore, many packets will be dropped when incast of short flows happens. We simulated a small scale datacenter in OMNET++ [33] to check performance of Homa when incasts of extremely short flows exist. Figure 3 shows the result of this experiment at 60% load. The left hand side figure shows Homa's performance when incast does not exist and the figure on the right shows its performance when incast of short flows exists. X-axis shows the total delay in milliseconds. We see from the figure that some packets that fall in tail are dropped even more than twice (RTO=3). Thus, receiver-driven methods such as Homa can help but only after the first RTT, which is late in such networks that require very low latency (i.e., zero packet drop).

The current trend of increasing intensity of short flows and incast imply that congestion control schemes are unlikely to solve this problem. Therefore, modern datacenters must resort to other solutions such as large (shared) buffers or use higher capacity topologies such as Subways [34] that provides higher bandwidth for endhosts. Modern shared-buffer switches are equipped with dynamic sharing technologies such as Dynamic Threshold (DT) [35] or Enhanced Dynamic Threshold (EDT) [36]. Although these methods perform well in case of mild congestions, they face various issues when incast congestion happens. For example, DT always keeps a specific amounts of buffer reserved for other inactive ports, while a port that experiences incast might need it. To further evaluate DT's performance, we simulated a datacenter network using reported numbers in previous studies (e.g., [8] and [5]) to measure the performance of DT in case of incast. we explain the workload details later in chapter 4. Figure 4 shows the performance of DT in terms of tail FCT. We set the Retransmission timeout to 3 seconds. Figure 4 shows that DT suffers from excessive packet drops at high loads (60% to 80%) due to its design that keeps the buffer underutilized. Therefore, while DT is a great solution for mildly congested networks, it will not help in busty environments such as datacenter networks. EDT, however, performs better; but it may hurt fairness as it assigns the whole buffer to a single port when a burst of packets arrives at that port. In such cases that more than one server is experiencing incast, EDT performs even worse.

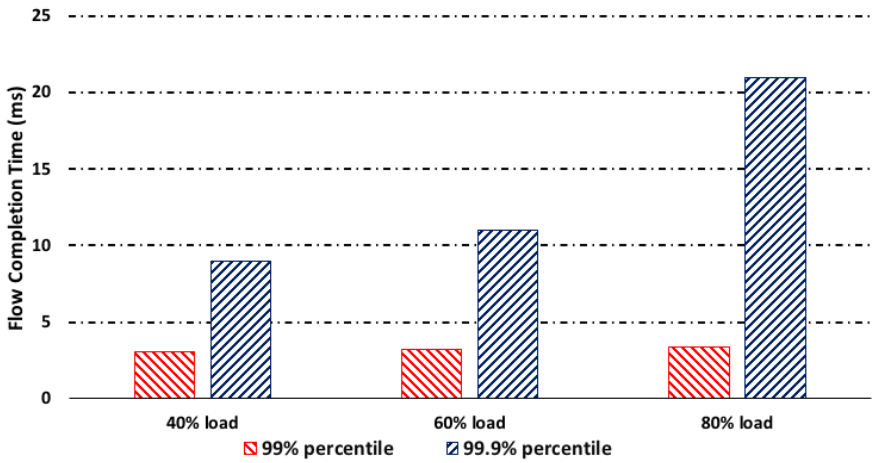

Figure 4. DT's performance in shared buffers

CHAPTER 3

PROPOSED METHODS

NOTE: Some of the contents of this chapter have been published before in:

(1) Rezaei, Hamed, and Balajee Vamanan. "ResQueue: A Smarter Datacenter Flow Scheduler." Proceedings of The Web Conference 2020. 2020. 2599-2605

(2) Rezaei, Hamed, and Balajee Vamanan. "Superways: A Datacenter Topology for Incast-heavy workloads." Proceedings of the Web Conference 2021. 2021. 317-328

(3) Rezaei, Hamed, Malekpourshahraki, Mojtaba, and Balajee Vamanan. "Slytherin: Dynamic, network-assisted prioritization of tail packets in datacenter networks". 27th International Conference on Computer Communication and Networks. 2018. 1-9

(4) Rezaei, Hamed, Almasi, Hamidreza, and Balajee Vamanan. "Icon: Incast congestion control using packet pacing in datacenter networks". 11th International Conference on Communication Systems and Networks. 2019. 125-132

In this section, we propose our solutions for incast problem in datacenter networks. Our proposed methods are Slytherin, ICON, ResQueue, Superways, and Smartbuf. Among the above-mentioned methods, Slytherin and ResQueue use flow classification techniques to improve tail latency of short flows in presence of incast. ICON is a new congestion control scheme that is designed for incast heavy workloads. Finally, Superways changes the underlying topology to provide more resources for incast applications, and Smartbuf, makes switches more intelligent by providing a prediction technique at switches that creates a knowledge base for current bursts (i.e., learns each burst's behavior) and uses this information for future bursts.

3.1 Slytherin

In this section, we discuss *Slytherin* which unlike prior schemes specifically targets packets that are more likely to fall in the tail. This is important because the performance of foreground datacenter applications (e.g., Web search) are sensitive to the tail of flow completion times. Slytherin leverages Active Queue Management (AQM) schemes which are available in today commodity switches to identify congested tail packets and then assigns higher priorities to those packets.

A TCP flow is not finished successfully unless all transmitted packets reach the destination. If some packets get delayed, the total flow completion time of flow will be increased. This problem caused by delayed tail packets of a flow because they face queuing delay at multiple switches on the path. Thus, although we may have most of packets arrived in a short time but delayed tail packets will increase the total flow completion time. Our main contribution is our

TABLE I

Slytherin's opportunity

Load	40%	50%	60%	70%	80%	90%
Fraction of packets	1.3%	3.8%	19%	40%	56%	65%

novel insight that packets that are more likely to fall the tail often incur congestion at multiple points in the network.

To achieve our goal of identifying packets that incur congestion at more than one point in the network and prioritize them quickly to improve tail flow completion times, we need:
(1) A fast and a reasonably accurate signal to pinpoint packets that are likely to fall in the tail. We set out with the explicit goals of not requiring custom hardware and supporting coexistence with legacy transport protocols like TCP [37]. To have higher response time, we should use in-network mechanisms to detect congestion at each switch independently.
(2) Provide a fast prioritization method to improve tail flow completion times. Consequently, the packet prioritization mechanism should be in-network because short flows only last for a few RTTs.

3.1.1 Identifying tail packets

Explicit Congestion Notification (ECN) [38] is widely used AQM scheme in datacenters. With ECN, switches mark packets when the queue length exceeds a certain predefined threshold.

Instead of using expensive, unreliable packet timestamps, we leverage ECN to pinpoint tail packets. More specifically, we infer that marked packets have experienced congestion in their paths, and therefore, those packets must be prioritized ahead of packets without ECN marks.

We performed an opportunity study to confirm our intuition that packets that are more likely to fall the tail often incur congestion at multiple points in the network and that we can identify those packets using ECN marks. For this study, we simulated typical datacenter traffic patterns, as reported in prior papers [39], in a spine-leaf topology with 400 hosts (see chapter 4 section 4.1). All the hosts run DCTCP. Table I shows the fraction of tail packets (i.e., here we only consider packets whose flow completion times are greater than the 95^{th} percentile flow completion times; these packets are more likely to impact tail FCT than packets at lower percentiles) that are ECN marked at more than one switch in their path. In other words, it shows what fraction of critical packets incur ECN marks at more than one switch. We clearly see that a *significant* fraction of these packets are marked at multiple switches. This study clearly validates our approach.

Since each of switches perform ECN marking individually, congestion is recognized independently at each hop. This ensures that whether there is only one congestion point or multiple points of congestion, it will be reported by the corresponding switches to end hosts. Slytherin uses this bit of information not to let the end hosts to decide about the congestion but react to it just in-network. More specifically, each Slytherin switch check this bit of information individually to see whether the packet has experienced congestion at previous hops or not.

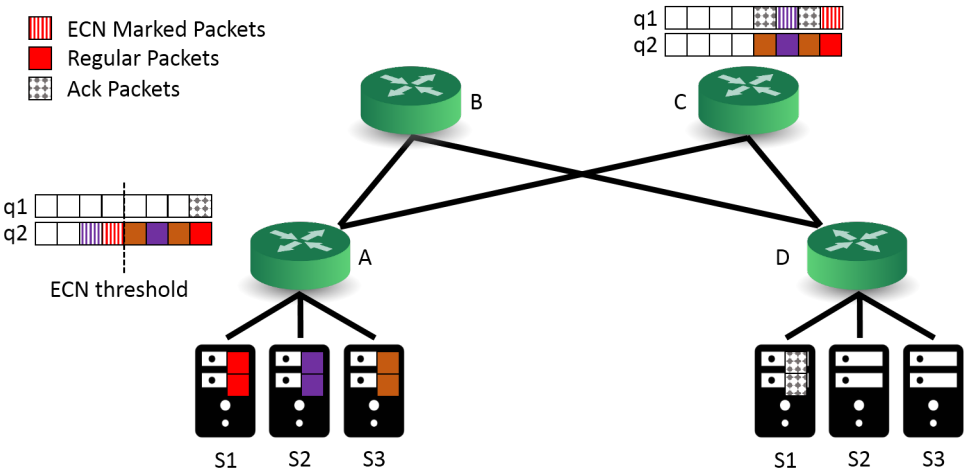

Figure 5. Slytherin's high level idea

Since AQM schemes are available in today commodity switches, we can simply leverage them to discriminate among packets that suffered from congestion in previous hops vs. other packets. Because ECN is meant to inform end hosts about congestion, existing switches do not check whether this bit is set or not. However, Slytherin requires switches to check this bit and prioritize such packets.

3.1.2 Prioritizing tail packets

As we discussed above, ECN is a widely used, in-network congestion notification mechanism at switches. Our analysis shows that this bit of information should be used by the switches to prioritize previously congested packets over the others. If the CE bit in the packet's header is set, this packet is considered as a congested packet; which means it requires priority escalation at next hop switches to minimize flow latency. Scheduling packets that suffered from congestion

in the higher priority queue assures that tail packets that faced congestion earlier will not be delayed at the current switch. It significantly improves tail flow completion times which is a key performance metric in datacenters.

Slytherin switches require *two* queues per switch port. Packets are assigned to one of those queues based on their priority. The first queue is a priority queue which is dedicated to congestion experienced packets and the second one is a normal queue which is shared among all other packets regardless of their flow size or flow deadline. Both queues drain packets in a First-In-First-Out (FIFO) fashion and packets in the second queue get served if and only if the first queue is empty. Figure 5 shows Slytherin's congestion detection and packet prioritization mechanism.

Alongside prioritizing congested packets, we prioritize ACK packets over other packets by scheduling them in the priority queue. Although it's not our novelty to prioritize ACK packets but our empirical analysis shows that ACK packets prioritization could be a good scheme to be used in conjunction with Slytherin.

Slytherin is designed to prioritize packets *after* the first hop upon observing ECN marks. Slytherin's Packet prioritization mechanism is totally application-agnostic. If a packet gets ECN marked, whether it's part of a short flow or a long flow, the packet will be prioritized at the next hop switch. Although prioritizing congested packets of short flows may be more efficient, it requires application knowledge and/or significant effort. Our experiments show that there is only a small degradation in performance in case of prioritizing both long and short flows in contrast to only prioritizing short flows.

Algorithm 1: Slytherin's packet prioritization pseudocode

```
1 for Each packet "P" to be enqueued do
2     if CE bit is set
3         put "P" in higher priority queue
4     else
5         put "P" in lower priority queue
```

Algorithm 1 shows both congestion detection and packet prioritization steps at Slytherin's switches. Slytherin's design is such simple that its complete implementation requires handful lines of code on the switches. Irrespective of where packets are ECN marked (i.e., enqueue vs. dequeue), Slytherin prioritizes marked packets to reduce flow completion times for both short and long flows. Overall, Slytherin's implementation is simpler than other schemes such as Pfabric [6], PIAS [16], and PDQ [30].

3.1.3 Parameters setting

In this section, we will discuss important parameters settings that directly affect Slytherin's performance.

3.1.3.1 Transport protocol

Slytherin is designed to work in conjunction with DCTCP [10] as the underlying transport protocol. DCTCP aggregates the ECN feedback from multiple packets to form a weighted-average metric for adjusting the window [7]. Using this feedback, the senders adjust their

congestion window sizes in a graceful manner, so that any congestion over the path from source to destination will be proactively treated.

As long as Slytherin is an in-network scheme, it can work with any other transport protocols but we chose DCTCP due to the better throughput it provides for long flows and better flow completion time that it provides for short flows. Our analysis shows DCTCP's congestion window adjustment could be improved if ECN marked packets are drained faster at switches. In the other words, if ECN marked packets arrive faster, DCTCP adjusts its congestion window faster to avoid further congestion which leads to less packet drops, higher throughput, and lower flow completion times.

3.1.3.2 ECN threshold

Slytherin's most important parameter to tune is the ECN threshold at each switch queue. This threshold should be set carefully as it directly affects the performance. The threshold is set to 25% of the total queue size. If the threshold is too short, many packets are assigned to the higher priority queue which means still many previously congested packets will be scheduled behind others in the current switch. Briefly, if the threshold is too short, Slytherin's performance will be degraded.

If the threshold is too big, probably no packet gets prioritized over others but only some packets that are about to drop. In this case, the switch marks a lot of packets as not-congested while they are potentially congested. It means that setting up a big threshold leads to performance degradation as well. We will study the sensitivity of our proposed method to ECN threshold in chapter 4.

3.1.4 Fairness and high load scenarios

It is important to discuss Slytherin's fairness and its performance under sustained high load. Because Slytherin only prioritizes a small fraction (i.e., tail packets) of flows, fairness is largely unaffected; our results show that Slytherin achieves better tail flow completion times for short flows and better throughput for long flows than PIAS and DCTCP. Another important question is about Slytherin's performance when there is simultaneously high queuing along most (all) switches along the path. Our experiments show that it is so rare for many switches along the path to have high queuing *at the same time* that it does not affect 99^{th} percentile FCT (e.g., such scenarios happen less than 1% of the time).

3.2 ICON

Slytherin does not completely solve incast problem because it assumes incast flows are large enough so that there is always some packets left that need to be transmitted in the next TCP windows. However, recent studies reveal that incast flows are super small so that most of them finish just in the first sending window [5]. Thus, Slytherin helps in milder forms of congestion. Also, while Slytherin expedites packets in the *middle* of the network, incast mostly happens at the last hop switch that is connected to the receiver server. Therefore, we need an accurate congestion control scheme that directly addresses incast.

In this section, we present the technical details of *ICON*, which specifically targets incast problem in datacenter networks. Incast is predominately induced by a set of servers (e.g., 32) that respond to a server's query. For example, in distributed storage clusters, a file may be divided into multiple chunks that are stored on different servers across the datacenter. In such a case, a single server, which is responsible for retrieving and reassembling the whole file, queries *all* the servers to retrieve the file. Such distributed lookup results in incast. Incast causes many packet drops at the edge switch connected to the retrieving (sink) server. As we discussed in section 1, incast is an important and critical issue in datacenter networks, and incasts degrade the performance of both foreground and background applications.

ICON's incast control algorithm consists of two parts: (1) end-to-end application level knowledge sharing about the number of concurrent senders (i.e., incast degree), and (2) pacing TCP window over RTT proportional to the shared number. We will discuss the application-level information sharing in section 3.2.1 and the traffic pacing in section 3.2.2.

3.2.1 Application-level knowledge sharing

We make the key observation that incast, which bottlenecks the receiver (or the last switch's output port that connects to the receiver), it is inherently application-depended, and, therefore, the incast problem could be solved at end hosts rather than involving switches and routers.

The application layer of the sink server, which is responsible for distributed lookup, knows the number of the servers that it is going to the search query. Therefore, simply communicating this *incast degree* to all those servers that are part of the distributed lookup could go a long way in addressing incasts. At the very beginning of the communication between all senders and the sink server, this number will be piggybacked by TCP packets from sink server to sender servers (during TCP 3-way handshaking). At this point, all senders know how many different servers will share a same route through the sink server. Giving this prior knowledge to servers is key as it can serve to right the correct sending rate at the sender. However, there are a few challenges that we must solve to realize the goal:

Communicating incast degree to senders. We can use unused fields in IP or TCP header for this purpose. For example, we can use "options" field in TCP header or "Type-of-Service" (ToS) field in IP header but we chose the first one as it does not affect the network functionality.

Handling persistent connections. We know the calculated number of servers will be reported to sender servers only during the TCP handshake. If the TCP connection between end hosts is persistent, sender servers save the reported number in their memory and use it unless a new information arrives from the sink server. In the other words, the very first reported

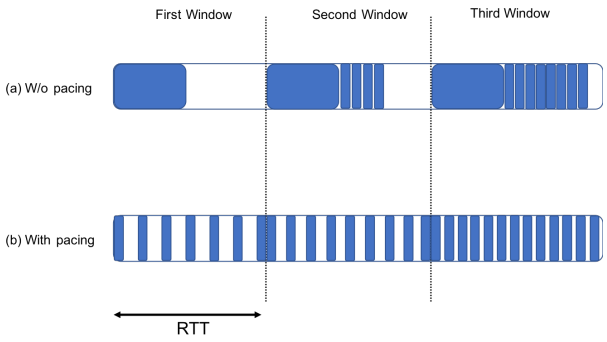

Figure 6. (a) Normal TCP vs (b) TCP with Pacing

number will be valid until a change happens in the number of active servers. This number rarely changes because the number of senders is usually fixed as they are chosen to hold a particular piece of data. However, if this number changes, the sink server notifies other servers through a small update message. These update messages are designed to only convey the number of parallel senders. As long as only *one* server will send these small update packets, bandwidth consumption is almost negligible.

3.2.2 Traffic pacing

In this section, we cover details of traffic pacing at sender NICs. Pacing and rate control are two *different* aspects of congestion control. While rate control adjusts the sending rate to avoid any long-term over-subscription of the bottleneck link capacity, rate control alone is not sufficient to avoid congestion. Pacing controls the rate at a much more finer grain (e.g., how the packets are spaced within an RTT). Two schemes with the same sending rate but different packet pacing can lead to different queuing delays in the network. In general,

uniformly distributing the packets over the sending "time period" can smooth out queue build-ups and reduce congestion. Figure 6 shows the difference between Normal TCP and the one which uses pacing.

ICON paces TCP window over RTT by providing a gap (pause time) between packets. As we can see in Figure 6, as TCP window grows, this gap becomes smaller because of more packets that it needs to send. Since TCP normally sends a window of packets as a bulk, it is more prone to cause packet drops as more packets are likely to collide on the bottleneck link. However, ICON paces packets over RTT to provide more time for the switch port at the bottleneck link to drain packets.

3.2.3 Combination of pacing and application knowledge sharing:

ICON uses a combination of application knowledge sharing and traffic pacing in order to improve tail flow completion time and throughput. Once the sender servers receive the number of active senders from the sink server, they start to calculate the pacing rate (i.e., the gap between each packet) using this number. Equation 3.1 shows how ICON calculates the new sending rate.

$$Sending\_rate = \frac{TCP\_window\_size}{RTT \times number\_of\_parallel\_senders} \tag{3.1}$$

As Equation 3.1 implies, if TCP window grows up, the sending rate will increase which might be problematic if the flow size is very large. However, this is not a big issue in incast

as incast is mostly caused by short flows. The newly calculated sending rate ensures that the sending rate of all senders is proportional to the number of competing flows on the bottleneck link. This technique guarantees that the bottlenecked switch port buffer has enough space to absorb all packets heading to the sink server. As an example, if TCP window is 10 MSS, RTT is 120 microseconds and we have only 1 sender, the new sending rate requires 10.8 microseconds gap (pause time) between sending each packet (assuming link speed is 10 Gbps). As you can see, our new congestion control scheme requires only a tiny modification over current transport protocols and is implementation-friendly.

In the absence of application knowledge, we assume that the "$number\_of\_parallel\_senders = 1$" in the previous equation. We show in section 4 that ICON outperforms other schemes, even without application knowledge.

3.3 ResQueue

ICON performs very well in theory but there are some obstacles that make it difficult to implement. First, every server's TCP stack should be changed so that it can set/read tags on the outgoing/incoming packets, which means current datacenter administrators will be relentless to adopt this method because it requires significant amounts of effort to update each single server in the datacenter. Also, it is relatively difficult for the querying server to know the exact number of servers that it is going to query. Lastly, if the querying server is simultaneously participating in two or more incast type conversations, since each calculation is done separately for each incast scenario, the calculated number is wrong as the combined input rate is higher due to higher number of senders that were not included in the calculations.

We introduce ResQueue, which is a more effective solution to incast that does not rely on changing congestion control. ResQueue detects those packets that were dropped in the past, and prioritizes them over packets of that same size that did not incur loss. This is critically important because the performance of foreground applications (e.g., Web search) is highly sensitive to packet loss rate as TCP timeouts are multiple orders of magnitude larger than typical flow completion times. Our at-scale simulations show that while a large fraction of the Web application flows are delivered to the destination server in less than 100 microseconds, others that are dropped more than once could be delivered after 3-4 milliseconds (assuming RTO = 1ms). Below we discuss the mechanism of ResQueue, which is designed to rescue those packets that belong to short flows from being dropped again.

ResQueue is a novel flow scheduling method that is designed to give more accurate priority to flows based on *their size* and *their history of packet drops*. At its core, similar to PIAS [16], ResQueue implements a Multi Level Feedback Queue (MLFQ) at switches, in which a flow is demoted from higher-priority queues to lower-priority queues according to their number of sent bytes. The priority of each flow is calculated based on the number of bytes sent, and then, this priority is tagged on the packet by the sender server. In MLFQ mechanism, if there are N queues in the switch, all flows will be scheduled in the first queue when they start the data transmission (bytes sent = 0). Flow priority demotes when the flow has already sent some packets and exceeds a predefined threshold. Packets that are scheduled in the second queue will not be dequeued unless all packets in the first queue are dequeued. Due to space constraints, we do not provide the details of priority demotion here. The whole mechanism is explained in [16].

In ResQueue, we reserve the first queue for those *retransmitted* packets that belong to short flows (the size of this queue is 20KB in our experiments. We further discuss the size of reserved buffer in chapter 4, section 4.3.2.4). We do this reservation for only short flows because Web search workload is dominated by short flows, and, therefore, the performance of the whole network will be determined by these short flows.

If the sender server is transmitting a normal packet, ResQueue acts like PIAS. However, if the sender server is *retransmitting* a packet (or a window of packets), it calculates the priority of the packet(s) in **two** rounds: first, it finds the appropriate queue level based on its number of sent bytes (as in [16]), and, second, it subtracts *one* from the calculated value because of its

drop history. For example, if a flow has only one packet to send and this packet was dropped before, its bytes sent indicates that the packets should be enqueued in queue level-2; however, the server tags the retransmitted packet with priority 1 due to its drop history, and, therefore, all switches will enqueue this packet in queue level-1. We provision a reserved high-priority queue *only* for short flows that experienced packet loss. As in [5], Web flows are so small that they are often enqueued either in level-1 or level-2 queues, depending on their drop history.

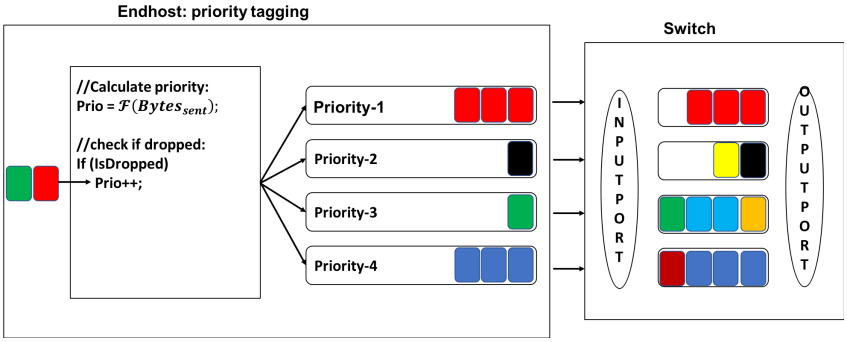

Figure 7. Endhosts: Red packets were dropped before; their priority will be escalated. Switches: Red packets are enqueued in the appropriate queue based on their priority

Note that although ResQueue does not schedule large flows in high priority queue, it still improves throughput of these flows. While those large flows that experience one or more packet drops in the past will not collide with short flows, they get higher priority compared to other

same-size flows that did not suffer from packet drops in the past. Imagine that a large flow's packets are scheduled in queue level-4, and, meanwhile, a window of its packets get dropped due to high congestion. When the window of packets is retransmitted, they will be scheduled in queue level-3 to avoid further delays to these packets. Although queue level-3 is not reserved for dropped packets, packets in this queue will drain faster than packets in queue level-4, which improves overall throughput to some extent.

Figure 7 depicts a comprehensive example of ResQueue's mechanism. As we see in Figure 7, the sender is retransmitting the red packet that was dropped in the previous round. Although the first round of priority calculation indicates that this packet should be scheduled in queue level-2 (based on sent bytes), dropped flag is set for this packet, which leads to priority escalation. Therefore, when the red packet arrives at the switches, this packet will be scheduled in an isolated high priority queue (queue level-1), which guarantees that this packet will not collide with a burst of packets again.

For the best performance, all servers across the datacenter and all switches in the path from source to destination need to support ResQueue to save the retransmitted packets. ResQueue's implementation is similar to PIAS — ResQueue tags packets with priority information based on bytes sent at end-hosts; upon timeouts, the priority is decremented.

3.3.1 Why ResQueue works?

Recent studies on datacenter networks show that incast inter-arrival time is so small that about 30% of the incasts arrive at most 50 microseconds after the previous one [5]. Considering this traffic pattern and the size of Web search flows in modern datacenter networks, retrans-

mitted packets are likely to collide with a burst of packets, which causes excessive packet drops. Note that RTT in datacenter networks is usually higher than 50 microseconds, which further complicates the situation. Because flows are so small that they last only for a handful of RTTs (assuming drops), prioritizing retransmitted packets would save them from being dropped again after colliding with a burst of packets. Although there are not too many of retransmitted packets that collide with a burst of packets, it still negatively affects the higher percentiles (i.e., 99^{th} percentile) of flow completion times. Note that in datacenter networks we only care about tail (i.e., 99^{th} to 99.9^{th}) flow completion time of short flows. Therefore, ResQueue has the potential to play a crucial role in improving tail flow completion times of short flows.

3.3.2 Why not prioritize flows instead of packets?

If a flow already suffered from packet drops in the past, the performance would be higher if the flow remains prioritized until it finishes. However, there are some challenges that we need to address if the *whole* flow remains prioritized:

- If the whole flow is prioritized, large flows may remain prioritized for a long time, which causes performance degradation to short flows.
- We need a relatively large table at the endhost to track the flows that already suffered from packet drops. This mechanism requires a high number of unnecessary operations at the endhost.

There is at least one solution for the first challenge. For instance, the endhost can increase flow priority if larger number of packets are dropped, and decrease the priority if larger number of

packets are yet to be transferred. However, this solution is indeed complicated and needs to be further analyzed.

For the second challenge, our key insight is that because most short flows only consist of handful of packets (1-2 packets), we can consider each packet as a separate flow without adding complexity to the system. Therefore, prioritizing *retransmitted packets* would be more efficient and easier to implement.

3.3.3 Packet reordering

In this section we investigate the effect of ResQueue on packet reordering at the receiver server. Packet reordering is an important issue as it directly affects the flow completion time of the flow. If some packets arrive out of order, the receiver server needs to spend considerable number of CPU cycles on reordering them, which further delays the flow completion time. ResQueue only schedules *retransmitted* packets in the higher priority queue. These packets are already out of order and ResQueue's prioritization mechanism guarantees that they will not arrive at the receiver later than the new window of packets. Thus, ResQueue does not worsen packet reordering. In fact, ResQueue somewhat alleviates packet reordering by making those late packets to arrive earlier.

3.4 Superways

ResQueue addresses the shortcomings of Slytherin and ICON but may not provide the highest possible performance when there are a large volume of same size incast flows in the network. In other words, if many same size bursts exist, ResQueue gives the same priority to almost all the packets by design, which does not solve the incast (repeated packet drops) problem. Also, it still requires setting/reading tags on packets, which adds some complexities to the network. Although our congestion control (ICON) scheme and flow scheduling techniques (Slytherin and ResQueue) outperform their predecessors, each of those methods come with their own challenges that makes them difficult to implement in current datacenter networks. What if we change the underlying topology to provide more resources for incast applications? Changing topology does not change the TCP stack and does not require extensive modifications to switches' architecture.

The high-level idea of Superways is to provision more bandwidth for incast aggregators. We argue that providing more bandwidth for incast aggregators is achievable because incast aggregators contribute to small fraction of total number of servers in a datacenter network. Our analysis on publicly available Facebook datacenter traffic dataset shows that around 3% of servers are likely incast applications, as they receive a large number of flows in a short period of time. These servers receive 18 times more flows compared to the rest of servers (i.e., 97% of servers), on average.

Above analysis shows that the number of incast applications in a real datacenter is reasonably small so that providing additional links only for incast applications is feasible. A naive

approach is to provision one link from leaf switch to incast aggregator for each worker server, so that all worker servers can send at full rate. However, this would be prohibitively expensive. Further, incast degree would vary drastically across applications, and perhaps may even vary with time. To tackle this problem, we leverage the key observation that most incast flows tend to be short, and are unlikely to utilize the full line rate. Therefore, we propose provisioning receive bandwidth based on a number of factors, including the number of links, average size of flows, and buffer sizes of routers. We discuss the details of our link provisioning method later in this section.

Although providing additional links for servers is not a new idea, Superways brings two novelties: (1) it selectively provides additional links **only** for those servers that require more bandwidth (i.e., those servers that host incast applications), and (2) it greedily finds the optimal number of additional links proportional to incast degrees of incast applications. At its core, Superways co-locates incast applications on certain physical servers (e.g., multiple virtual machines each of which holds one or more incast applications) and then greedily finds the optimal number of additional links that are required to absorb all those incast applications' packets combined. For example, we select a physical server in one of the racks to move a certain number of incast applications to this server, and then we calculate the number of additional links required for serving all the incast applications' traffic combined.

The question is: is this feasible to relocate applications in the network? Fortunately, due to rapidly growing use of containerized applications in datacenters, incast applications can be deployed in containers, and therefore, managing these applications (e.g., moving them to

another cluster) is easy to do using application container management tools such as Kubernetes [40]. Nowadays, almost all modern datacenters use containerized applications, which is a great advantage for topologies such as Superways that require freedom in server placement. Moreover, recent studies such as [41] show that although traffic characteristics such as load and packet sizes change frequently, large scale network characteristics such as workload changes infrequently. Thus, there is no need to constantly relocate the incast applications when Superways is implemented.

In contrast to other datacenter topologies that provide additional links between servers and leaf switches, Superways connects incast applications directly to spine switches (or inner switches in random graph topologies) to make a trade-off between cost and performance. By connecting to spine switches, Superways does not saturate uplink capacity (i.e., links from leaf to spine switches), which improves throughput and tail latency of the flows. This is critically important because provisioning additional links between servers and leaf switches will increase the over-subscription ratio of the network, which increases the number of in-network packets. Note that the network capacity remains unchanged. This issue leads to unpredictable consequences such as excessive packet drops and high queuing delays. That is why previous studies that provision additional links from servers to leaf switches require more leaf switches to absorb the extra packets. Also, providing additional links from servers to spine switches may seem more expensive compared to other methods that connect to leaf switches; however, since spine switches are more powerful in terms of link speed and number of links, Superways cuts the overall cost by decreasing the number of required cables, number of server NICs, and number

of additional switches. Note that link speed directly affects the number of required links. For example, a zero-buffer switch port that operates at 40 Gbps, supports 4 senders that are sending at 10 Gbps, without loss of throughput and without any packet drops. We will further discuss the reasons for connecting to spine switches in chapter 4.

3.4.1 Placement heuristic

Knowing the incast degree of each of incast applications is key in finding the number of additional links. This could be achieved through monitoring the network traffic or previous knowledge about the number of machines that each incast application will query. Although each incast application is expected to query a fixed number of servers, it may query a different number of servers in some cases, depending on the task that is requested by the end user. We argue for considering average incast degree of each of incast applications. We will discuss our decision in details later in chapter 4. Once we measured the average incast degree of each incast application, we sort them in a *descending* order to see which of incast applications needs more additional link(s) (i.e., more bandwidth+buffer space) and which ones require less or even no extra links.

We start assigning the incast application with *highest* average incast degree to a physical server. This can be done through moving the VM that hosts this incast application to an elected physical server, or installing the incast application on the elected server in a containerized environment. The physical server that hosts one or more incast applications is called *incast server* throughout this paper. We start from one of the racks that is closest to the rack of spine switches, and then we pick one of the servers in this rack as our elected server, which will be

hosting the first incast application. In random graph topologies, we use the same approach as we need to keep the wiring lengths as short as possible.

Once we placed the first incast application with highest incast degree on the designated incast server, we have multiple rounds of calculations to find the minimum number of additional links. Table II shows the parameters that we use in these calculations. Below we will explain some of these parameters.

Due to various delays at kernels (e.g., handling interrupts, etc), all incast senders do not start sending their data at the same time. Therefore, we consider a jitter factor in our formulas (shown by α). α should be set to a value close to 1 (e.g., 0.8 - 0.9). In our experiments, we observed that about 80% of the sender servers send their messages at the same time.

We know from previous studies that incast senders of a particular incast application can be anywhere in the network. We call those incast servers that reside on a different rack *remote incast senders* (e.g., S3 through S6 in Figure 9). We need to differentiate remote and local incast senders because local incast senders could be able to use the current link (i.e., leaf switch to server) without requiring any extra links. Also, even remote incast senders might be able to use the current link without any packet drops. This will definitely change Superways' design. Therefore, we use β to show the ratio of local incast senders to the total number of incast senders ($\beta = 1/3$ in Figure 9).

Average flow size varies depending on the workload in different datacenter networks. However, as shown in figure 6 of [8], about 60% of the flows are 1 KB or below in Web search workloads. Our analysis on Facebook datacenter traffic confirms that flows in Web search

workloads are mostly small so that the average flow size for all flows is close to 1500 bytes. Although this number may vary among different datacenters and different workloads, the nature of current datacenter workloads implies that most flows are in the range of 1 KB to 10 KB in size. Similar numbers have been reported in previous studies such as [42].

In the first round of calculations, we use Equation 3.2 to check if incast senders that reside on the same rack (i.e., local incast senders: S1 and S2 in Figure 9) exceed the buffer capacity of the existing link from leaf switch to incast server. Since buffer size of each port plays a key role in handling incast (i.e., reduces packet drops by storing packets in a queue), we consider both bandwidth and buffer size in our calculations.

$$U_l = \frac{\alpha \times (\beta \times d) \times s}{B_l} \tag{3.2}$$

U_l is the ratio of used buffer to the total buffer capacity of the link that connects the leaf switch to an incast server, and therefore, could be either less or greater than one. We analyze both cases in the following sections.

3.4.1.1 U_l is greater than one

If U_l is greater than one, then it means local incast senders of that particular incast application will overflow the leaf switch's buffer. For example, although incast degree of R in Figure 9 is 6, if S1 and S2 (local incast senders) start sending their data at the same time, they will overflow the leaf switch's buffer assigned to the port that connects to incast server. Therefore,

TABLE II

Parameters and descriptions

Parameter	Description
α	Jitter factor
β	Ratio of local incast senders to all senders
θ	Ratio of spine link speed to leaf link speed
B_s	Spine switch per-port buffer
B_l	Leaf switch per-port buffer
d	Incast degree
s	Average flow size in Bytes
U_l	Fraction of occupied buffer - leaf switch
r_s	Per-port residual buffer - spine switch
r_l	Per-port residual buffer - leaf switch
N	Maximum number of parallel senders
L	Number of additional links

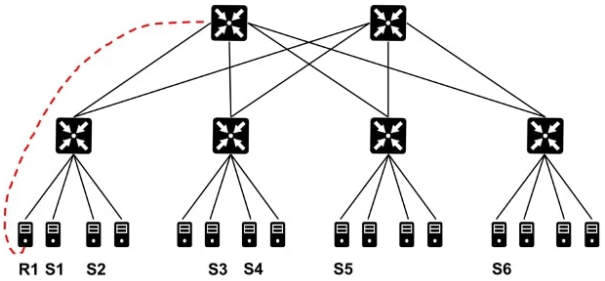

Figure 8. Superways/leaf-spine

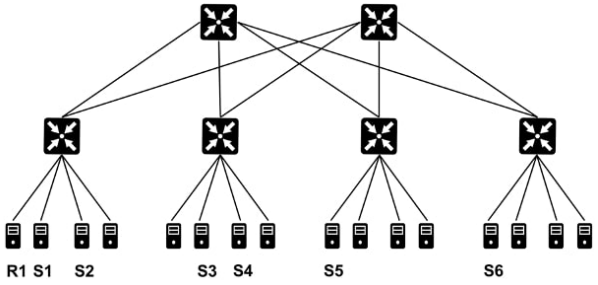

Figure 9. Regular leaf-spine

we use Equation 3.3 to calculate the maximum number of flows (shown by N) that can pass through the *leaf* switch towards the incast server without any packet drops.

$$\frac{\alpha \times N \times s}{B_l} = 1 \tag{3.3}$$

Because local incast senders already exceed the leaf switch's port buffer, some of them should use the additional link(s) that we will provide for the remote incast senders. Therefore, we need to update the ratio of local incast senders to the incast degree in order to consider some of them as remote incast senders. We use Equation 3.4, to calculate the updated number of local incast senders that can use the current leaf switch's link capacity:

$$\beta = \frac{\lfloor N \rfloor}{d} \tag{3.4}$$

We use flooring to provide a safe margin for our calculations. At this point, we have the updated number of remote incast senders that should use the additional link(s). Next, we calculate the number of additional links (from incast server to spine switch) that is required by the remote incast senders (1-β):

$$L = \frac{\alpha \times ((1-\beta) \times d) \times s}{\theta \times B_s} \tag{3.5}$$

We consider the value of $\lceil L \rceil$ as the minimum number of additional links that is sufficient to avoid packet drops. We do not consider the existing link (i.e., from spine switch to leaf switch) as a usable link for incast senders. Instead, we leave this link for other servers that are located in the same rack. Note that due to different link speeds at spine and leaf switches, we need to add another factor (θ) to our formula. This factor is always equal or greater than one, as spine switches are equipped with faster NICs. The larger the θ the better because that particular spine port can support more incast applications.

Depending on the value of L, we might have a large value of residual buffer that could be used by other incast applications. For example, If $L = 1.2$, then we need to provide 2 additional links to absorb all incast packets; but, 80% of the second link's buffer capacity is still available for other incast applications. Therefore, we calculate the residual buffer of the spine link as follows:

$$r_s \leftarrow B_s \times (\lceil L \rceil - L) \tag{3.6}$$

At this point, we need to find an incast application that is able to use this residual buffer. This is the classic Knapsack problem: we pick the incast application with highest incast degree that fits in the residual buffer. We start from the top of the sorted list of incast degrees and calculate the value of $\frac{\alpha \times d \times s}{\theta \times r_s}$ for each incast application. Once this value becomes less than one, we assign the corresponding incast application to the current incast server. Note that all local and remote senders of the recently added incast application should use the additional link(s); because there is no more available buffer in the link from leaf switch to incast server.

We update the value of r_s after assigning the second incast application to the incast server (using Equation 3.5 and Equation 3.6). If still any of incast applications can use the updated residual buffer, we use the same approach to install it on this incast server. On the other hand, if the residual buffer is not enough for any of incast applications, we move forward to the next server in this rack and repeat the whole process on a new incast server.

3.4.1.2 U_l is less than one

If U_l is less than one, then it means current local senders do not overflow the leaf switch's buffer. In other words, depending on the size of residual buffer, one (or even more) of the remote incast senders might be able to use the leaf switch's port. We continue our calculations by measuring this residual buffer:

$$r_l = B_l \times (1 - U_l) \tag{3.7}$$

Now we check if all the remote incast senders fit in the residual buffer or not:

$$U_l = \frac{\alpha \times ((1-\beta) \times d) \times s}{r_l} \tag{3.8}$$

There are two possibilities depending on the value of U_l:

- U_l is still less than one: This means all local and remote incast senders can use the existing link connected from leaf switch to server, without any packet drops. Next, we calculate the residual buffer as follows:

  $$r_l = \lceil U_l \rceil - U_l \tag{3.9}$$

  Lastly, we check the sorted list of incast degrees to find the incast application that can use the residual buffer. Again, we greedily pick the application with highest incast degree that fits in the residual buffer. Once found, we co-locate it with existing incast application on the current incast server. If the residual buffer is not enough for any of those incast applications, we leave it unassigned. This helps in having extra capacity when a burst of packets arrives at the switch port.

- U_l is larger than one: We need to provide additional links from incast server to a spine switch to absorb remote incast senders' packets. However, we should fully utilize the leaf switch's residual buffer first. Therefore, we repeat the previous calculations (Equation 3.3 and Equation 3.4) to see how many of remote incast senders can use the link from leaf switch to incast server, and how many of them should use the additional links. Next, we use Equation 3.5 and Equation 3.6 to find the optimal number of additional links, and

then we calculate the residual buffer of these additional links that can be used by other incast applications:

If the residual buffer is enough to absorb incast packets of any of the incast applications (depending on their incast degree), we bin-pack it on the current incast server. However, if there is no more residual buffer, we move to the next server in the current rack and select it as our new incast server. Above calculations will continue until all incast applications are assigned to their incast servers. If the number of incast applications is very large, we may need to proceed to the next rack (which is the second closest to spines rack) and pick a server as the new incast server. Note that each incast server is connected to one spine switch only (through one or more links), while each spine switch might be connected to multiple incast servers.

3.4.2 Routing and load balancing

Superways creates a heterogeneous topology, and therefore, requires a load balancing method other than equal-cost load balancing schemes such as Equal Cost Multi Paths (ECMP). This is because there will be at least one shortcut route between an incast server and its senders (i.e., workers), which means the final cost of all possible routes is no longer the same. However, the routing mechanism will not change as we need incast senders to use the shortcut routes, which will be picked by existing routing protocols such as OSPF, by default (due to their lower cost compared to regular routes). Nevertheless, this holds for remote incast senders only and routing for local incast senders could be different. Below we provide the details of routing and load balancing for local and remote incast senders.

3.4.2.1 Load balancing for remote incast senders

Assuming a cost based routing protocol, packets originated from remote incast senders are always forwarded through the additional link(s) due to their lower cost compared to other paths. As an example, in Figure 8, imagine S5 is sending a message to R1. S5's packets will be forwarded through the red link because of its lower cost compared to the normal paths (black links). If more than one additional link is provided, all the additional links can be bundled to a fat link, and therefore, still no further actions are required for load balancing remote senders' packets. In this case, an equal-cost load balancing method such as ECMP (or any forms of packet spraying) would work as well.

If our calculations require some of the remote servers to use the leftover bandwidth of the initial link (i.e., existing link between server and leaf switch), the network operator needs to set a static route on the spine switch to route the flows accordingly. However, the leftover capacity of the initial link is likely very limited (due to high incast degrees in modern datacenters) so that we do not expect that many remote incast senders use the initial link's capacity.

3.4.2.2 Load balancing for local incast senders

New studies on modern datacenters' traffic reveal that datacenter traffic is no longer rack local (i.e., only 13% of flows remain in the rack [8]). Thus, packets originated from local incast senders may or may not need further load balancing actions as it is unlikely to have too many local incast senders. In other words, if *some* of the local senders must use the additional link(s) (i.e., initial link's capacity is not enough even for absorbing local senders' packets - see section

3.4), we need to change the normal route to incast applications only for these servers. There are two solutions for this issue:

- **Source routing:** Incast senders can explicitly inform the switches about each packet's route. While source routing is easy to implement, it may increase latency (by adding processing delay), which is critical in datacenter networks.
- **Static routing:** We may need to add a couple of static routes to leaf switches to re-route the flows that must use the additional link(s). The static route should match the source and destination of each packet. Note that we need to check both the source and destination because the static route should affect *incast senders* only. Since most existing switches support OpenFlow, we can apply these simple configurations remotely on the SDN controller. As we mentioned earlier, we do not expect to have many local incast senders, and therefore, we may need to add only a handful of routes into the routing table of switches.

In the case that local incast senders do not need to use the extra capacity, no further action is required as existing link from leaf switch to server is perfectly handling the incoming traffic.

3.4.3 Topology management

In this section, we address physical considerations of implementing Superways, such as topology upgrade and wiring complexities.

3.4.3.1 Topology upgrades

Superways' upgrading complexity is very similar to that of the underlying topology. The reason is Superways only targets a small number of servers in the topology, which does not lead to huge complexity in topology upgrades. In Superways, some specific servers are chosen to host bandwidth hungry applications, and then more links are provisioned for these servers proportional to their load and their fan-in degree. Thus, it would be a good idea to have a designated area (e.g., some racks or some pods in extreme cases) in the datacenter network, and place incast applications on servers of this area. As a result of this isolation, whenever a new incast application is placed in a datacenter network, it will be installed on one of these servers based on the calculations described in section 3.4.1.

There are two main reasons for colonizing incast applications: (1) incast applications will be able to use the potential extra capacity left by other incast applications, and (2) simplifying topology management. The latter is crucial as more incast applications may be added to the datacenter network over the course of time, and colonizing incast applications helps us in lowering the degree of heterogeneity of the topology. In other words, if there are many incast servers spread across the topology, while they have extra connections to spine switches, there will be many small islands in the topology that further complicates routing and load balancing, requires longer cables, and complicates topology management. Therefore, we suggest to focus on one area of the topology and move every incast aggregator to this area. To see a clear picture of this process, refer to Figure 10. This figure shows a schematic view of a typical tree topology deployed in Facebook [43]. We can pick as many racks as we need and then connect

the servers inside those racks to the spine switches in one of the spine planes (e.g., spine plane 1). For wiring simplicity, we always pick closest racks to one of the spine planes. In Figure 10, as an instance, servers in rack 1 of pod 1 and switches in spine plane 1 are good candidates to host incast applications as they are the closest to the spine plane 1.

Someone may note that colonizing some servers on a specific rack (or some racks) may not always be doable as there are some services that have placement restrictions. However, recent studies on datacenter networks show that servers do not tend to host rack local services [8], and therefore, most of them are not location sensitive. Other studies such as [12] confirm this fact that incast degrees can be much higher than number of ports in a leaf switch, and therefore, there must be many more worker servers outside of this rack.

Due to high growth rate of using online services (e.g., social networks), more incast applications are expected in modern datacenters. Therefore, if incast servers in the current designated rack cannot handle new incast applications, the datacenter operator can select another rack as the n-th designated rack and repeat the above-mentioned process for application placement. Since there are more than one spine switch in a spine plane, there is no problem with connecting to the next spine in the same plane, if there are no more available ports in the current spine switch. Note that the next spine switch should be the second closest spine switch to our designated rack. In some rare cases, spine switches in the closest spine plane may not have any available ports to connect to incast servers. In this case, we need to place extra spine switches with higher number of ports in that plane (i.e., spine rack) to connect to incast servers. Note that this spine switch should be connected to all leaf switches in that pod (see Figure 10).

Due to high number of ports in spine switches (i.e., 128 - 256), however, we expect to have enough free ports at spine switches given that there are small number of incast applications in a datacenter network.

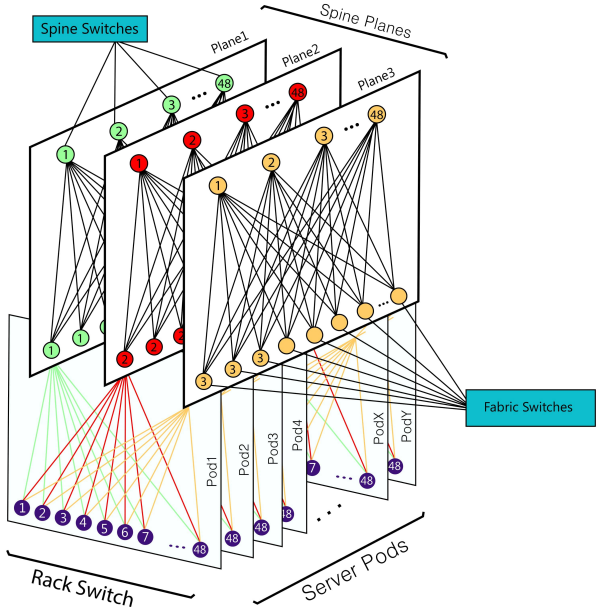

Figure 10. A schematic view of Facebook datacenter

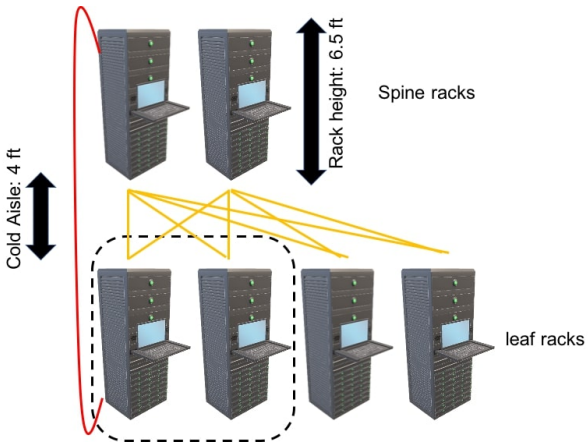

Figure 11. Wire length in a Superways datacenter

3.4.3.2 Wiring length

Superways does not require long wires if closest racks to spine switches are selected to host incast applications. Figure 11 shows an example of Superways' architecture. Closest racks

to spine switches are shown by dotted lines, which are the best candidates for hosting incast servers. Assuming a rack height of 6.5 ft and cold aisle width of 4 ft [44], the length of a wire that connects an incast server in a leaf rack to a spine switch in the spine rack is less than 15 ft, on average. The red link in Figure 11 shows this extra link. In the extreme case (i.e., providing an extra link from a server at the bottom of leaves' rack to spine switch at the top of spines' rack), we need a wire that is between 20 to 25 ft long, which is not a long cable given that it is still less than average cable length in production datacenter networks [45]. If we need to move to another rack due to increased number of incast servers, we still connect these servers to a new spine switch in the closest spine rack (e.g., second spine rack in Figure 11), and therefore, the average cable length in Superways will not increase.

3.4.3.3 Server replacement

Superways changes the placement of incast applications in a datacenter network to achieve higher performance. Although it is despite of common server placement routines in datacenter networks, Superways replaces about 3% - 4% of all servers (see section 3.4), which is doable in small to medium scale datacenter networks. Implementing Superways on existing tree or graph topologies is not challenging because some of incast applications remain in the same rack even after colonizing incast applications in a rack. Also, since many modern datacenters are virtualized, migrating VMs from one server to the other one is indeed easy. In case of non-virtual environments, in the worst case scenario, we need to replace all incast applications; and, in the best case scenario, almost no incast application needs to be replaced. Therefore, we only need to replace 1% to 2% of all servers, on average.

3.4.3.4 Link aggregation

If incast degrees are high, more additional links are required to absorb the burst of packets. Thus, we suggest to use a link aggregation protocol to create a fat link out of the additional links, and create a high bandwidth link that is able to process more packets at a given time. Link aggregation simplifies load balancing decisions and increases bandwidth. The latter is critically important because all co-located incast applications are not likely to receive their incast replies at the same time, and therefore, an incast aggregator can aggregate responses much faster due to high bandwidth at the spine switch. In other words, if multiple bursts arrive at a single incast application, while other incast applications installed on the same server are not active at this moment (i.e., they are not receiving a burst of packets), the overloaded incast application can use a big fraction of the fat link's capacity that was provisioned for *all* incast applications installed on that server. Therefore, extra capacity will be available for those incast applications that are actively receiving back-to-back incasts or experience higher incast degrees than usual. Having back-to-back bursts is quite likely as reported in [5]. This study reveals that incast inter-arrival time could be extremely short so that many of bursts arrive only 50 microseconds after the previous burst.

In summary, since not all incast applications receive their packets at the same time (i.e., all extra links are not 100% utilized at all times), considering *average* incast degree of each incast application would be sufficient to ensure that our calculations guarantee the enough number of additional links. Our simulation and testbed results confirm this assumption.

3.5 Smartbuf

All previously discussed methods improve tail latency of short flows and throughput of long flows but they either need endhost modifications (i.e., TCP stack change or server replacement), or require prior information about the flows. Superways provides the best performance so far; however, it could be difficult for datacenter administrators to redesign the network, specially when the datacenter is relatively large. Therefore, we need an efficient scheme that does not require endhost modifications, and also, does not incur significant changes to the network (e.g., redesigning the whole network). Smartbuf, is the last discussed method that fulfills the mentioned objectives. Below we elaborate on the details of Smartbuf.

While it is difficult to predict the arrival time of a burst in datacenter networks, some ports (e.g., those that are connected to incast aggregators) are more susceptible to large bursts and require more buffer space than other ports. Our high-level idea is to learn the upper bound of buffer occupancy that every burst could lead to and capping the allocated buffer for each port that experiences a similar burst at that level.

Fortunately, the traffic dynamics of datacenter networks is such that we can predict the buffer demands of ports by observing past behavior. and it suggests that observing previously seen bursts' signature could help provisioning for future similar ones. During high fan-in bursts, data from multiple input ports get forwarded to the same output port within a switch, which causes a sharp increase in the output port's queue length in a short time. Because most flows that are involved in such a burst happen to be short flows [5,8], *fan-in (incast degree) is a good predictor of the buffer demand* (i.e., other factors like flow sizes and protocols play a relatively

minor role). We see a clear one-on-one association between fan-in (incast degree) and buffer demand. This *key observation* enables us to estimate buffer demands of ports and allocate space accordingly.

Fan-in, however, is application-level knowledge and there is no easy way for switches (network layer) to access this information without substantial changes to the protocol stack and applications. Because the gradient of queue length is proportional to the aggregate incoming rate (i.e., $average\ sending\ rate \times fan\text{-}in$), we make the *key insight* that *gradient of queue length*, which is a local information, can be effectively used as a *proxy* for fan-in. Thus, instead of using fan-in, we use the gradient of queue length to index into our database of past demands, which is maintained as a key-value store. To further reduce the sensitivity of our approach to event-based switch measurements, we propose to use the smoothed (averaged) gradient of queue lengths as the signature for the bursts and use it as a key to store and retrieve the buffer requirements of subsequent bursts. We use EWMA (Exponential Weighted Moving Average) over a short time window to ensure that we capture the presence of back-to-back packets in a burst and also filter extreme outliers to improve accuracy.

The switch starts observing burst signatures and records the maximum buffer occupancy seen at a port for each key (gradient of queue length). It continuously learns these key-value pairs and updates the values whenever a higher upper bound is encountered. The set of key-value pairs is used to match on future signatures and to allocate per-port buffers. Due to the dynamic workloads, the entries eventually age out after some time (i.e., soft state), and new signatures are learned. In chapter 4, we show that dynamically allocating buffer space, which

is a scarce resource, to ports *proportional* to their demand is key to achieving high performance in today's datacenter networks and show that existing proposals fall short of this goal.

3.5.1 Smartbuf algorithm for buffer allocation

In this section, we explain the details of Smartbuf's buffer allocation mechanism. As we mentioned in Section 3.5, Smartbuf focuses on catching bursts' behaviour and uses this knowledge to accurately predict the buffer demand of future bursts. Details of this algorithm are shown in Algorithm 2. This algorithm consists of two phases: *Learning* phase and *Real* phase. In the Learning phase, the switch records gradient of each port's queue length alongside with the maximum queue occupancy corresponding to this gradient *per each packet dequeue*. Since not all the information is useful (i.e., there is no need for recording non-burst flows' information), we need *two* data structures to record the pair of gradient of queue length and maximum buffer occupancy observed: (1) Temp_map, which is a hash map of an integer (port ID) and a pair of gradient of queue length and maximum buffer occupancy (line 6 of Algorithm 2), and records **all** the information. (2) Main_map, which is a hash map as well, and records the burst-related information **only** (line 7 of Algorithm 2). Smartbuf kicks in only if a burst is detected. Thus, when no burst is detected, each port's buffer threshold is calculated by DT [35] (line 8 and line 35 in Algorithm 2), which is an efficient buffer management scheme in absence of incast.

While Temp_map records per port information only, Main_map is global across all ports so that all ports can use the learned knowledge of other ports. Therefore, if a port is experiencing

Algorithm 2: Smartbuf

Begin

Input: instantaneous queue length

Learning-phase

Initializations:

$max\_buf\_seen, Gradient, Gradient_{prev} \leftarrow 0$

$Temp\_map[port, [Gradient, max\_buf\_seen]] = \phi$

$Main\_map[Gradient, max\_buf\_seen] = \phi$

$Port\_buf\_threshold = DTthreshold$ (see [35])

for *each packet dequeue at port* P **do**

$Gradient = (Qlen - Qlen_{prev})/(T - T_{prev})$

$Qlen_{prev} \leftarrow Qlen$, $T_{prev} \leftarrow T$

$Gradient \leftarrow 1/4 \times Gradient + 3/4 \times Gradient_{prev}$

$Gradient_{prev} \leftarrow Gradient$

if $(cur\_buf\_in\_use > max\_buf\_seen)$

$max\_buf\_seen = cur\_buf\_in\_use$

$Temp\_map[P] \leftarrow [Gradient, max\_buf\_seen]$

else if $(cur\_buf\_in\_use < \beta * max\_buf\_seen)$

if $(max\_buf\_seen > (1/\sigma) * total\_buf\_size)$

$Main\_map \leftarrow Temp\_map[P]$

else

$Gradient, max\_buf\_seen \leftarrow 0$

$Temp\_map[P] \leftarrow [Gradient, max\_buf\_seen]$

else

Continue

(25)

(26) **Real-phase**

(27) **at line 17 of learning phase:**

(28) **if** $(max\_buf\_seen > (1/\sigma) * total\_buf\_size)$

(29) **for** $i = 1; i < K; i = i + 1$ **do**

(30) find i^{th} larger than current Gradient key in Main_map and store it in min(i)

(31) $avg = (min(1) + ... + min(K))/K$

(32) $Port\_buf\_threshold = avg$

(33) $Main\_map \leftarrow Temp\_map[P]$

(34) **else**

(35) $Port\_buf\_threshold = DTthreshold$ (see [35])

incast, other ports are able to manage similar bursts efficiently, if the workload changes and they suddenly become overloaded.

Calculating the gradient of queue length is challenging because the sign of gradient could change even at the early stages of an incoming burst. In other words, because not all packets in a burst arrive at the same time, gradient of queue length could be negative even inside a burst that more packets are yet to come. Therefore, Smartbuf calculates the *moving average* of gradients while it gives higher weight (e.g., 0.75) to the previous samples. This is a key step in Smartbuf's algorithm as it has a huge effect on accuracy of the calculations.

In lines 14–16 of Algorithm 2, we update Temp_map entries if a larger buffer occupancy is observed. Note that we aim to match gradient of queue length and maximum buffer occupancy observed (so far), and therefore, this step is important in recording the correct highest observed buffer occupancy. If buffer occupancy is being smaller than maximum buffer occupancy, there is no need to record this value as the switch already saved the largest buffer occupancy. Lines 17 through 24 focus on inserting Temp_map values into the Main_map (i.e., moving burst related information to the global hash map). There are *two* main conditions before this insertion: (1) if current buffer occupancy of a port drops below a certain value (line 17), then it means the saved max_buf_seen is a global maximum, and, *probably*, it is time to insert this entry in Temp_map into the Main_map. This condition (i.e., burst is finished) is satisfied if the current buffer occupancy is less than a certain fraction of the previously observed maximum buffer occupancy (i.e., max_buf_seen). This fraction is determined by a threshold called β. We need this threshold to accurately detect the correct finish time of a burst because there are some

large spikes in buffer usage that happen right after a temporary drop in buffer usage of a port, which we need to catch them too. We used $\beta = 0.5$ in our experiments. Therefore, if current buffer occupancy of a port drops below half of the observed maximum buffer occupancy, we may be able to add this information into Main_map because burst is most likely gone. Later in chapter 4, we discuss the sensitivity of our algorithm to β.

(2) As we mentioned above, we do not need to insert every entry of the Temp_map into Main_map unless that entry is certainly associated with a burst. Line 18 of Algorithm 2 is about this condition. Similar to line 17, we need a threshold that indicates the spike in buffer usage that was captured before, is certainty associated with an incast induced burst. Possible values for this threshold depend on the definition of burst in that particular network. As an instance, in a workload that most servers participate in partition-aggregate traffic (i.e., incast), a single burst could occupy the whole buffer, depending on the size of buffer. Therefore, this threshold should be configurable so that it matches the current workload.

In our experiments, we observed that in many cases that each port consumed about 20% to 40% of the whole buffer (i.e., $2.5 < \sigma < 5$), it is experiencing incast that eventually ends up consuming even more buffer space. Thus, if the existing max_buf_seen is larger than 20% of the whole buffer space (i.e., $\sigma = 5$ in a 4 MB shared buffer), then we insert the pair of queue gradient and max_buf_seen into the global hash map (i.e., Main_map). We will further discuss sensitivity to σ later in section 4.5.

In the Real phase, all operations in learning phase are still in progress. However, if line 17 of the algorithm holds, Smartbuf takes different actions that are shown in lines 28–35 of

Algorithm 2. When incast is detected, the switch looks for K saved gradients in the Main_map that are *larger* than current gradient, and are *closest* to it. The switch calculates the average of max_buf_seens corresponding to these K gradients and picks this value as the buffer limit for this port. Therefore, this port takes a certain fraction of the buffer that is enough for absorbing a burst, rather than taking the whole buffer space. Note that Smartbuf keeps inserting the new pair of gradient and buffer occupancy into the Main_map even in the real phase (line 33), which improves the buffer assignment accuracy even when workload is changing. Smartbuf uses the K-NN approach to increase the accuracy of its buffer allocation algorithm. Later in section 4.5, we discuss the range of values of K that lead to high performance. Also, we will discuss the total overhead of implementing Smartbuf on programmable switches as well.

There is a possible scenario that two or more ports are experiencing incasts, and the overall buffer that Smartbuf assigns to all ports exceeds the total buffer size. In such cases, we will have a FIFO in which each flow greedily gets its buffer share. If at any point, a port requires more space but there is no free memory left to be allocated, we have two options—simply drop the packet destined to that port or borrow memory from another port. We resort to the simpler option of dropping in this paper. However, dropping the packet seems the only viable option unless we dedicate a fall-back memory but tuning its size would then be another problem to solve.

3.5.2 Realization in programmable switches

Figure 12 shows one way to realize the proposed algorithm on a re-configurable match-action pipeline, per each output port. Because queue length and current buffer in use are needed as

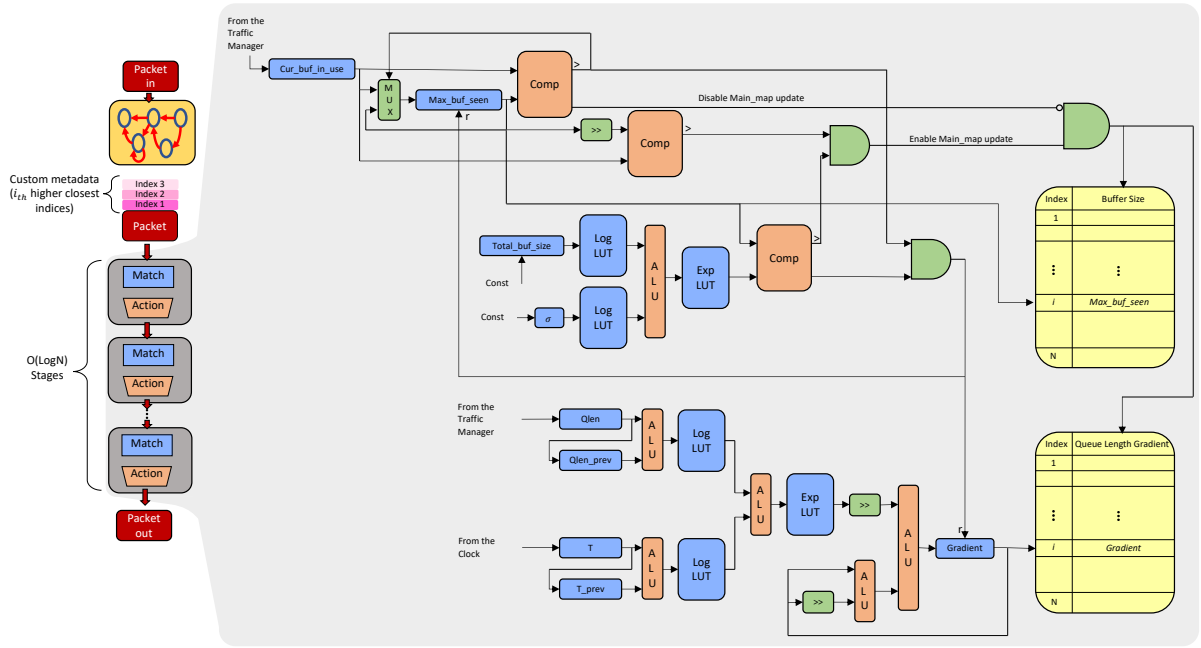

Figure 12. Block diagram of the proposed algorithm on a programmable switch pipeline

inputs to the algorithm, the algorithm should be implemented in the egress pipeline where the traffic manager can provide current values for queue length and current buffer in use for every port P. Because today's programmable switches do not support multiplication and division, we rely on look up tables (LUT) to implement them after converting those operations into logarithm and exponentiation. The calculated values for the gradient and the corresponding maximum buffer used are stored in the SRAM as outputs of the learning phase (see the two tables on the right in Figure 12). In the real phase, after the input packet is parsed by the

parser, we augment the header vectors with three integers as custom metadata that hold indices for three higher closest indices of gradients. As the packet passes through $O(logN)$ stages with N being the size of our main map, these metadata are populated and ready for buffer threshold setting.

The bottom portion of the figure computes the gradient per each packet. The top portion implements the comparison of the maximum observed buffer to the current buffer in use and the middle part compares the maximum buffer observed to the total buffer size. The green gate-level logic on the right decides whether to update the SRAM depending on the outcome of comparisons. The general pipeline for updating the indices for closest higher values to the gradient in $O(logN)$ stages and averaging them is depicted on the left but we omit the circuit-level details for brevity. We emphasize that this is one way to realize our algorithm but other implementations may be possible, depending on the constraints of the platform. Enforcing the buffer size (output from Figure 12) requires access to the Traffic Manager. The existing framework in Intel Tofino switches does not allow changes to this module but we are hopeful that switches will become more open in the future [46]. Therefore, a complete implementation Tofino programmable switches is not possible because the switches do not (yet) support modifying the queuing logic. In chapter 4, we provide bounds and show that our design is feasible for implementing on programmable switches.

CHAPTER 4

RESULTS

NOTE: Some of the contents of this chapter have been published before in:

(1) Rezaei, Hamed, and Balajee Vamanan. "ResQueue: A Smarter Datacenter Flow Scheduler." Proceedings of The Web Conference 2020. 2020. 2599-2605

(2) Rezaei, Hamed, and Balajee Vamanan. "Superways: A Datacenter Topology for Incast-heavy workloads." Proceedings of the Web Conference 2021. 2021. 317-328

(3) Rezaei, Hamed, Malekpourshahraki, Mojtaba, and Balajee Vamanan. "Slytherin: Dynamic, network-assisted prioritization of tail packets in datacenter networks". 27th International Conference on Computer Communication and Networks. 2018. 1-9

(4) Rezaei, Hamed, Almasi, Hamidreza, and Balajee Vamanan. "Icon: Incast congestion control using packet pacing in datacenter networks". 11th International Conference on Communication Systems and Networks. 2019. 125-132

In this chapter, we explain our experimental methodology and we will elaborate on efficiency of our proposed methods.

4.1 Slytherin

4.1.1 Experimental terminology

In this section, we present the details of our simulator implementation, topology, and workload, and then we provide the results of our experiments in the next section.

4.1.1.1 Topology

We use ns-3 [47] to simulate leaf-spine datacenter topology. Leaf-spine is a commonly used topology in modern datacenters [6]. In our simulations, the fabric interconnects 400 hosts through 20 leaf switches connected to 10 spine switches in a full mesh manner which provides over-subscription factor of 2. Each of leaf switches have 20 10 Gbps downlinks to the servers and 10 10 Gbps uplinks to the spine switches. The end-to-end Round-Trip Time (RTT) across the fabric is 80 μs.

4.1.1.2 Workload and Traffic

To evaluate our method, we simulate web search workload that is very common in modern datacenters. We consider two flow size distributions; short flows and long flows. All flows arrive according to Poisson process and the source and destination for each flow is chosen uniformly randomly. Short flow sizes are uniformly chosen in the range of 8 KB to 32 KB and our long flow's size is 1 MB. Since in web search workloads vast amount of all bytes are produced by 30% of flows that are larger than 1 MB [6], we use the same approach to produce the loads. We

also use those short flows to produce incast type traffic which are quite common in web search workloads.

To further evaluate Slytherin's performance, we consider two metrics; one for short flows which is Flow Completion Time (FCT) and one for long flows which is throughput. Moreover, we check both *average and* 99^{th} percentile flow completion times of short flows to measure the performance of our proposed method. Similarly, we evaluate Slytherin's performance in both average and tail flow completion times in different incast scenarios. We use incast degrees of 24, 32, and 40 (number of parallel senders to a single receiver) to check the sensitivity of our method to incast degree. Next, we analyze Slytherin's performance using other metrics such as queue length, convergence speed and number of reordered packets.

4.1.1.3 Compared schemes

- **DCTCP:** We implemented DCTCP, capturing all details in their paper [10]. We use DCTCP as baseline.
- **PIAS:** We implemented PIAS [16] on top of ns-3 [47] simulator. The implementation assigns 4 queues to each switch port. At the very beginning, all flows get mapped to to the highest priority queue (queue 1). If number of sent bytes of a flow reach a threshold, the priority of corresponding flow will be decreased and then the rest of packets of the flow will be assigned to lower priority queues (queues 2 to 4). We set the ECN threshold to 25% of the queue size and the load balancing method is flow ECMP.
- **Slytherin:** We implement Slytherin on top of DCTCP [10]. Our implementation uses two queues per each switch port, recall from section 3 that we use one high priority queue

for expediting ECN marked packets and ACKs, and one low priority queue for other packets. We set the ECN threshold to 25% of the queue size and we use ECMP for load balancing. We set out the rest of our network configuration to match PIAS.

- **Ideal SJF:** We also implemented an ideal SJF scheduler. Our SJF scheduler is aware of flow sizes and maps short flows to higher priority queues. While this is not realistic as flow sizes are often not known apriori, we use our ideal SJF implementation for deeper analysis of Slytherin's queuing delays.

As such, we use the same values for parameters that are common to DCTCP, PIAS, and Slytherin (e.g., ECN threshold), and the values match those used in previous papers. There are a number of packet scheduling approaches in the last few years such as pFabric [6], PDQ [30], and PASE [29]. However, because PIAS compares to and outperforms these approaches, we only compare Slytherin to PIAS.

4.1.2 Slytherin results

In this section, we evaluate the performance of Slytherin in different scenarios on top of ns-3 [47] simulator. Our performance evaluation consists of six parts:

- Bottomline comparison of Slytherin's Flow Completion Time (FCT) and throughput to PIAS
- Analysis of Slytherin's average queue length to explain our performance gains vs. PIAS and ideal SJF.
- Convergence analysis of DCTCP and Slytherin

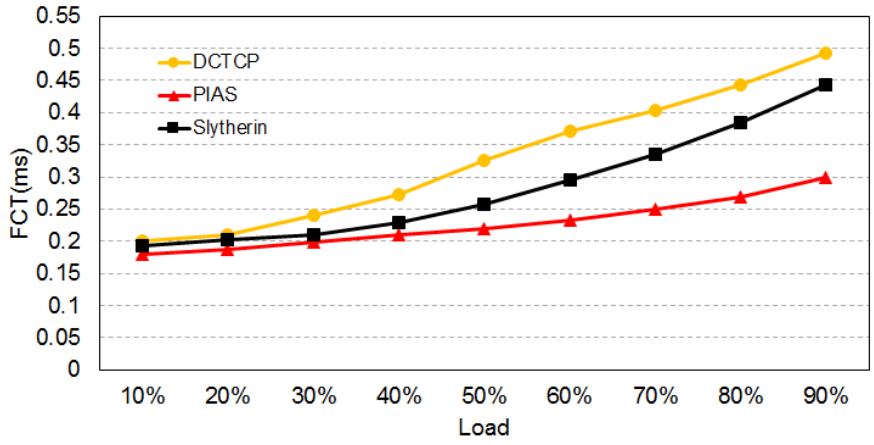

Figure 13. Tail (99^{th}) percentile FCT

- Sensitivity to varying incast degrees
- Sensitivity to ECN marking threshold
- Analysis of packet reordering in Slytherin vs. PIAS.

4.1.3 Tail FCT and throughput

In this section we will show how Slytherin performs in terms of both flow completion times (for short flows) and throughput (for long flows). The results for mean flow completion times, tail flow completion times, and throughput are shown in Figure 13, Figure 14, and Figure 15 respectively. In all figures, the X-axis shows the load factor on network. In Figure 13 and Figure 14 Y-axis shows flow completion time in milliseconds and in Figure 15 the Y-axis shows throughput in Gbps.

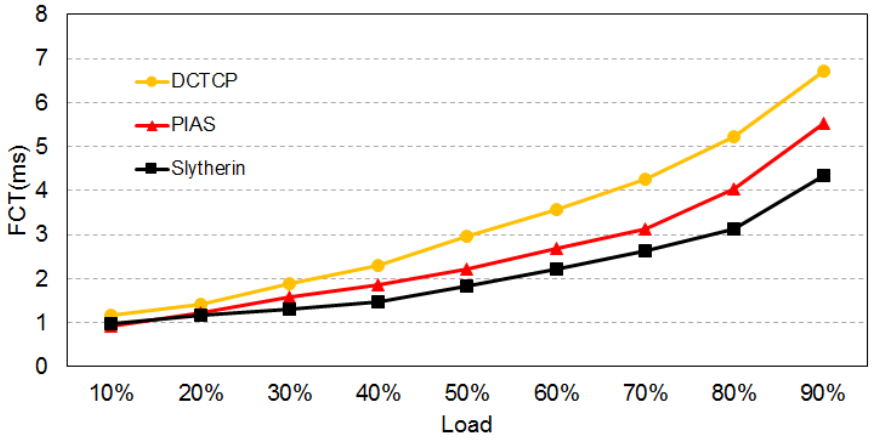

Figure 14. Average FCT

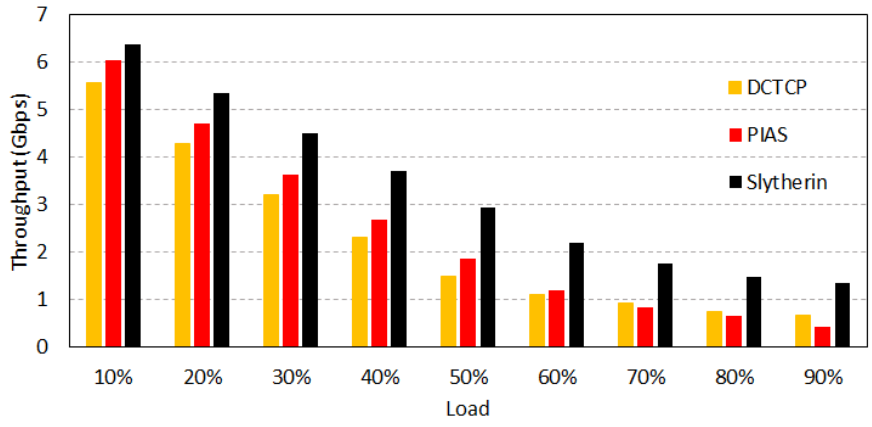

Figure 15. Average throughput (long flows)

4.1.3.1 Flow Completion Time

Figure 14 compares the 99^{th} percentile completion times of Slytherin and PIAS. As load increases, tail FCT increases for all the schemes. PIAS and Slytherin significantly outperform DCTCP. PIAS greedily assigns higher priority to flows that sent less packets regardless of queuing delays which leads to high queuing delay for some packets. Although PIAS benefits from ECN marking to prevent long queuing delay for longer flows, it falsely classifies some packets that have incurred higher queuing into lower priority queues which worsens tail FCT. In contrast, Slytherin targets tail packets and prioritizes them. At higher loads, Slytherin achieves better reduction in tail FCTs as there is more opportunity to schedule tail packets. Slytherin achieves an average reduction in tail FCTs of about 20% for loads greater than 20% (typical operating point of most datacenters).

Figure 13 shows the *average* FCT for short flows. Although PIAS outperforms Slytherin in average FCT but since the performance of foreground datacenter applications (e.g., Web search) are sensitive to the tail of FCT and not mean, Slytherin makes the right trade-off by prioritizing tail packets over average (or median) packets. Nevertheless, we expect to see better average FCT for PIAS because it emulates SJF, which is known to minimize average FCTs.

4.1.3.2 Throughput

To see the performance of Slytherin for long flows we measure *average* throughput in our simulations. Schemes that try to mimic SJF usually suffer from throughput issues; because they give strict priority to shorter flows. On the other hand, Slytherin's mechanism to expedite ECN marked packets would speed up the congestion reaction processes at end host which increases

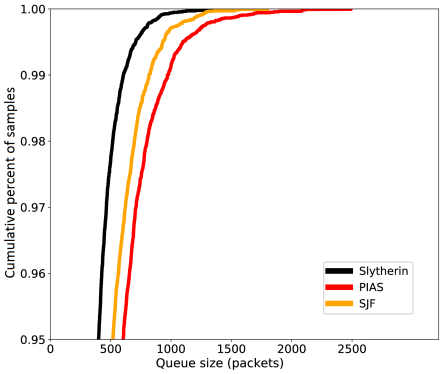

Figure 16. Load=40%

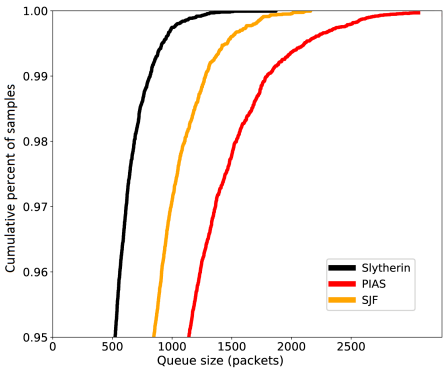

Figure 17. Load=60%

the control over sender's rate. Figure 15 shows the average throughput of Slytherin and PIAS for long flows. We see that Slytherin achieves higher throughput for long flows in contrast to PIAS. Overall, Slytherin achieves an average increase in long flows throughput by about 32%.

4.1.4 Queue length

Any packet scheduling scheme which keeps lower amount of packets in queues can successfully decrease the risk of packet drops in case of congestion. Recall from section 3.1 that since Slytherin works in conjunction with DCTCP, it tends to store lower amount of packets in queues because of its nature which transmits congested packets faster. More specifically, By expediting congested tail packets, end host's transport protocol would receive congestion signals faster and consequently it cuts its Congestion Window (CWND) faster to speed up congestion recovery processes.

In Figure 16 and Figure 17 we show the Cumulative Distribution Function (CDF) of queue lengths of all switches for Slytherin, PIAS and Ideal SJF, for 40% and 60% load, respectively. In ideal SJF, we assume that we know flow sizes apriori and switches can, therefore, faithfully implement SJF.

Figure 17 shows with a higher load factor (60%), the stored number of packets in queues increases for all schemes but Slytherin outperforms others even more. Overall, Slytherin significantly reduces 99^{th} percentile queue length in switches by about a factor of 2x on average. While SJF is known to substantially improve *average* flow completion times, our comparison with ideal SJF highlights Slytherin's ability to *specifically* target and improve tail latency beyond SJF.

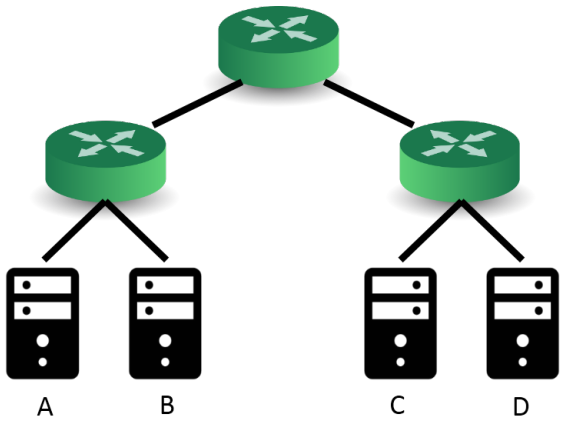

Figure 18. Scenario used for convergence time evaluation

4.1.5 Convergence time

Expediting ECN marked packets helps TCP flows to converge faster to their fair share bandwidths when multiple flows compete on a bottleneck link. We evaluate Slytherin's convergence using a small network (Figure 18) consisting of four servers. We initiate a long flow from host A to host C and then, after 10 RTTs, we start 20 concurrent flows from server B to server D.

Figure 19 shows the convergence time (i.e., time to reach the fair share rate) of DCTCP (baseline) and Slytherin; X-axis shows time and Y-axis shows throughput (measured per each RTT). We see that DCTCP takes 6 RTTs (480 μs) to converge to fair share whereas Slytherin converges in 4 RTTs (320 μs). Because Slytherin provides faster convergence, it effectively mitigates utilization and fairness issues in multi-bottleneck scenarios, as reported in prior studies [48].

4.1.6 Sensitivity to incast degree

We study Slytherin's sensitivity to incast degree and compare its tail flow completion times to those of PIAS. Figure 20 shows the 99^{th} percentile flow completion times of Slytherin and PIAS as we vary the incast degree as 24, 32 (our default), and 40 along X-axis, for different loads. As expected, the flow completion times increase with increasing incast degree and load. Slytherin relative gains are robust across incast degrees and loads. Overall, Slytherin achieves an average reduction in 99^{th} percentile FCT by about 21%.

4.1.7 Sensitivity to ECN threshold

Next, we analyze the Slytherin's sensitivity to ECN threshold. While a lower threshold would aggressively mark packets, promote more packets to the high priority queue, and cause

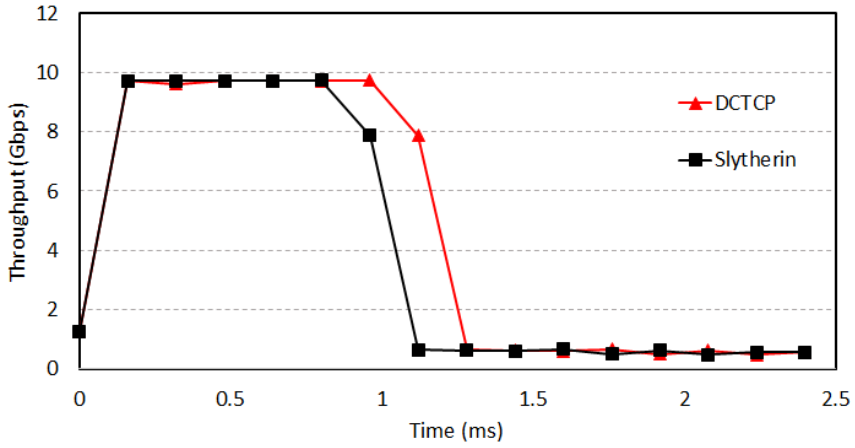

Figure 19. Convergence time

congestion, a higher threshold would be slow to react to congestion. Figure 21 shows the tail flow completion times of Slytherin for three different ECN threshold values of 12.5%, 25%, and 37.5% of total buffer size for varying loads. We see that Slytherin with threshold of 12.5% of queue szie suffers at higher loads as more packets get promoted to high priority queue and cause congestion. While a larger threshold of 37.5% of total buffer size achieves better (lower) 99^{th} percentile FCT at higher loads, we observed loss of throughput for long flows (not shown) due to slower reaction to congestion.

4.1.8 Packet Reordering

In this section we investigate the effect of expediting ECN marked packets on TCP packet reordering. Slytherin schedules congested packets ahead of others to decrease tail flow completion time which may cause packet reordering at the end hosts. We compare the number of

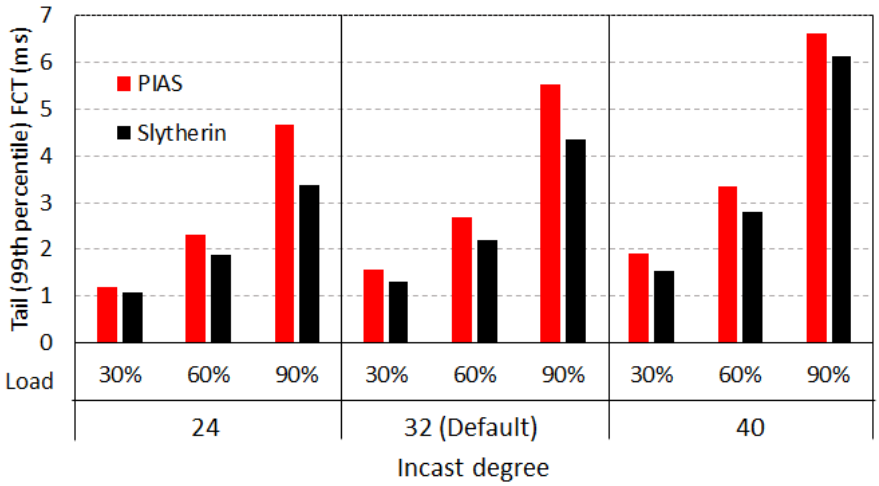

Figure 20. Sensitivity to incast

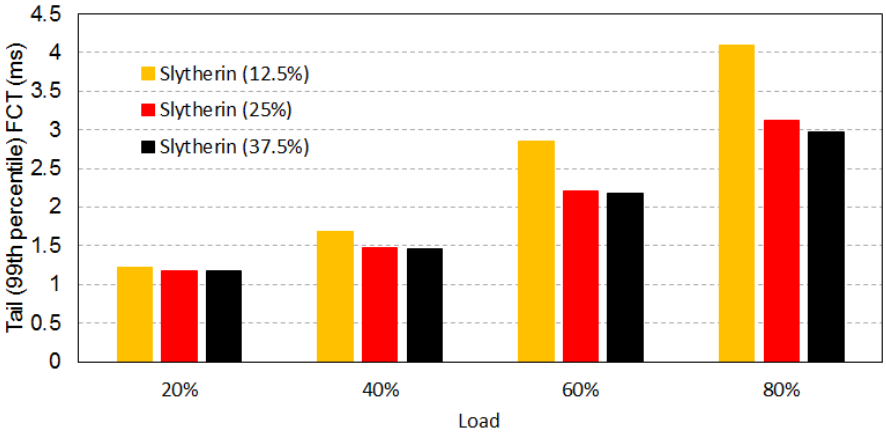

Figure 21. Sensitivity to threshold

reordered packets in Slytherin and PIAS to check Slytherin's performance in terms of packet reordering efforts. PIAS authors assume the switch has enough rooms in the buffer so that each of those sub-queues (e.g., priority queues) can accommodate all incoming packets. Furthermore, PIAS sets a priority for the whole flow and consequently the number of reordered packets could be nearly zero. However, in a more realistic scenario, when the capacity of each of those sub-queues is limited, PIAS needs to either drop or demote packets to a lower priority queue which both lead to significant packet reordering.

Figure 22 shows the number of reordered packets in Slytherin compared to PIAS. X-axis shows the load factor on network and Y-axis shows the ratio of PIAS's number of reordered packets to Slytherin. As shown in Figure 22, while PIAS reorders fewer packets than Slytherin at low loads, PIAS incurs higher packet reordering than Slytherin at high loads (i.e., at 60% and higher loads). As load increases, PIAS demotes more packets. Our experiments show that while Slytherin reorders only 0.56% of total number of packets (on average), PIAS reorders 1.2% of all packets (on average) across all loads. Because the absolute number of packet reordering would be far greater at high loads than at low loads, Slytherin is more effective than PIAS in reducing reordering effort.

4.2 ICON

In this section, we provide in-depth analysis of ICON's performance in datacenter networks.

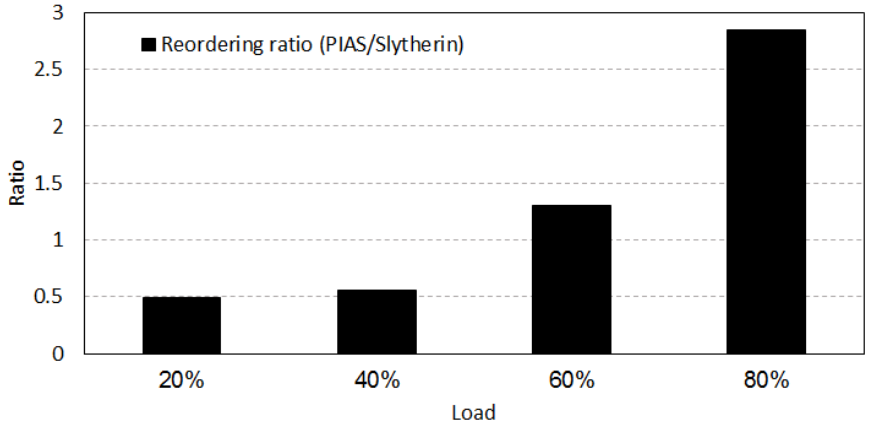

Figure 22. Packet reordering ratio (PIAS/Slytherin)

4.2.1 Experimental Methodology

We use ns-3 [47] simulator to simulate a datacenter network and evaluate ICON's performance. In this section, we present the details of our evaluation methodology including topology, workload, simulation parameters, etc.

4.2.1.1 Topology

We use ns-3 to simulate a *leaf-spine* datacenter topology. Leaf-spine is a commonly used topology in many datacenters [19]. In our topology, the fabric interconnects 400 servers through 20 leaf switches and 10 spine switches. Our topology uses an *over-subscription factor* of 2, which is typical. So, each of those leaf switches have 20 ports connected to the servers and 10 ports connected to upper spine switches. All the links from servers to leaf switches, and from leaf

switches to spine switches are 10 *Gbps* links. The maximum unloaded round trip time (RTT) of our network is 80 μs.

4.2.1.2 Workload

We model our workloads based on the results presented in [39]. Our workload has a mix of short and long flows. The flow arrivals are based on a Poisson process and source and destination servers are chosen uniformly randomly for both short and long flows. We define incast flows as a subset of short flows; the total load produced by short flows is equally divided among incast flows and other normal short flows (one-to-one short flows). Our workload model matches the one used in [49]. Our short flow sizes are randomly selected from a range of 8 KB to 32 KB and we set long flow size at 1 MB. Our long flows contribute to 70% of the overall network load, which matches previous work (e.g., pFabric [6]). We use a default *incast degree* of 26, but we vary the incast degree and stufy our sensitivity in section 4.2.2. We evaluate ICON's performance using both tail and median flow completion times for short flows, and throughput for long flows.

4.2.1.3 Compared schemes

We compare ICON to the following schemes:

- **DCTCP:** We implemented DCTCP, matching all the details from the original paper [10]. We set DCTCP to be our baseline, as it is commonly used in today's datacenters.
- **ICTCP:** We implemented ICTCP based on their paper and we set the parameters as suggested in the paper [11]. Our ICTCP implementation measures available bandwidth

at end hosts and sets the next receive window according to the available bandwidth. In the other words, the end host predicts the throughput and sets the receiver window based on the difference between expected throughput and the observed throughput. The receive window will be set to a smaller value if the difference is large. However, if the measured difference is small, TCP receive window is allowed to increase normally. ICTCP is able to set the fair share rate when a lot of packets arrive at a same time. ICTCP specifically targets incast.

- **ICON:** While ICON would work on top of any transport protocol, we implement it on top of DCTCP [10] as it is widely deployed in modern datacenters. Our proposed scheme consists of two parts. First, each server which is a part of TCP incast, paces its TCP window over RTT by setting appropriate pause times between sending each packet. Many modern NICs already support this feature [50]. Second, to find the best sending rate and pause duration for pacing, senders divide their sending rate by the number of parallel senders which is provided by the receiver (we describe the design in chapter 3, section 3.2).

- **Information-agnostic ICON (IA-ICON):** We implement another version of ICON which does not rely on information provided by receiver application (i.e., number of parallel senders). Instead, *IA-ICON* paces the TCP window over RTT. While this scheme will pace sender data regardless of number of senders, we include this method for two reasons: (1) quantify the utility of application knowledge and (2) isolate the effect of

pacing from rate control (i.e., IA-ICON does not start with the correct sending rate but ICON does).

Finally, we would like to emphasize that we use identical values for parameters that are common to DCTCP, ICTCP, and ICON (e.g., ECN threshold, buffer size, RTT). While there are a number of packet scheduling and load balancing schemes in the last few years such as pFabric [6], PIAS [16], CONGA [19], HULA [20], etc., we only compare ICON to DCTCP and ICTCP as they target incasts. As such, ICON can be implemented on top of most load balancing schemes.

4.2.2 ICON Results

In this section, we evaluate performance of ICON and compare its performance to other competing schemes. We present the following comparisons:

- Tail (99^{th} percentile) and median (50^{th} percentile) flow completion times
- Average throughput of long flows
- Number of dropped packets
- Sensitivity to different incast degrees
- Distribution of queue lengths in switches

4.2.3 Flow completion time

In this section we show how ICON performs in terms of median and tail flow completion times compared to other schemes. The results for median and tail flow completion times are

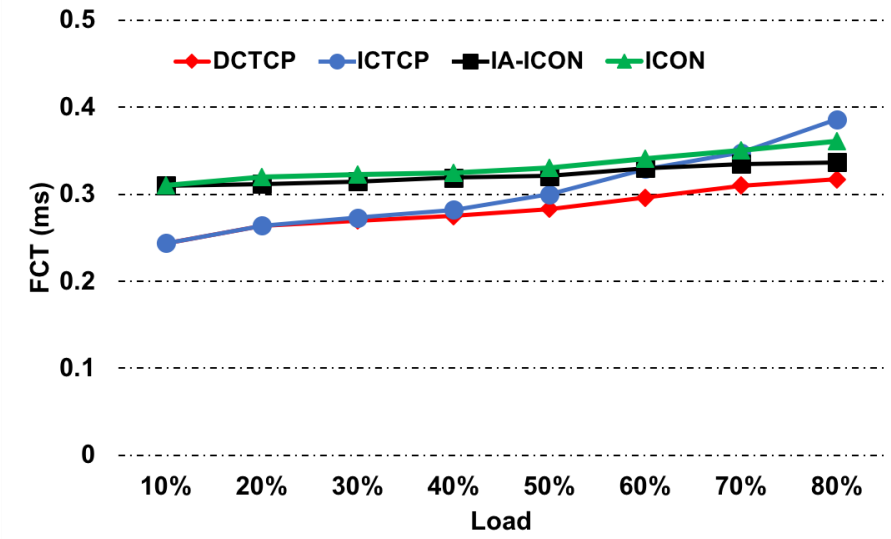

Figure 23. Median flow completion time

shown in Figure 23 and Figure 24, respectively. In both of these figures, X-axis shows the network load and Y-axis shows flow completion times in milliseconds. As expected, both tail and median flow completion times of all schemes increase with increasing load. We also see 1-2 orders of magnitude difference between median and tail flow completion times, matching several reported results [5, 10, 11]. Also, we see that tail rapidly increases at higher load values, which is also expected.

While ICON slightly under-performs other schemes (i.e., by about 13 %) in median flow completion times, it *drastically outperforms* (i.e., by about a factor of $7-8$ or by about 80%) all other schemes in tail flow completion times. Because most foreground applications are

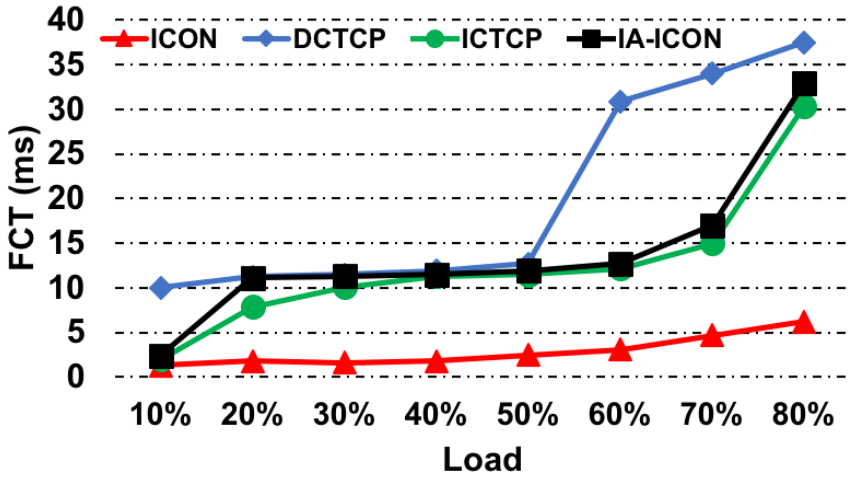

Figure 24. Tail flow completion time

sensitive to tail, not median flow completion times, ICON's contribution is well-suited to today's datacenters. In other words, ICON closes the gap between median and tail latencies, which reduces performance variability in datacenters. Also, it is interesting to note that IA-ICON performs quite close to ICTCP in tail flow completion times. Because IA-ICON isolates pacing from rate control, the fact that IA-ICON performs close to other complicated rate control schemes shows that pacing is as important as rate control in datacenter networks. The slightly worse median flow completion times, however, do not affect ICON's throughput, as we show next.

4.2.4 Average Throughput

In this section we compare average throughput of long flows for DCTCP, ICTCP, Information-Agnostic ICON, and ICON. As we can see in Figure 25, as the load increases, long flow throughput decreases for all schemes due to congestion in the network created by short, bursty flows; the queuing in the network increases with load. ICTCP outperforms DCTCP, specifically at higher loads. ICON *significantly* outperforms all schemes across a range of loads. Because ICON leverages application knowledge, ICON instantly converges to the correct sending rate; the other schemes require several RTTs to converge to the correct fair-share sending rate. On average, ICON achieves 18% higher throughput compared to ICTCP. IA-ICON performs similar to ICTCP, albeit with a much lesser complexity. Therefore, our throughput results also highlight the importance of pacing.

4.2.5 Number of dropped packets

We dissect the bottomline performance of ICON by comparing the number of dropped packets in all the schemes. Because re-transmission of dropped packets directly influences tail flow completion times, we compare maximum number of packet drops for different schemes. As the load increases, the number of packet drops for all schemes increase due to increased congestion. As we see in Figure 26, ICON substantially cuts the number of packet drops. Because ICON sets the right sending rate and paces the packets, ICON nearly eliminates packet drops.

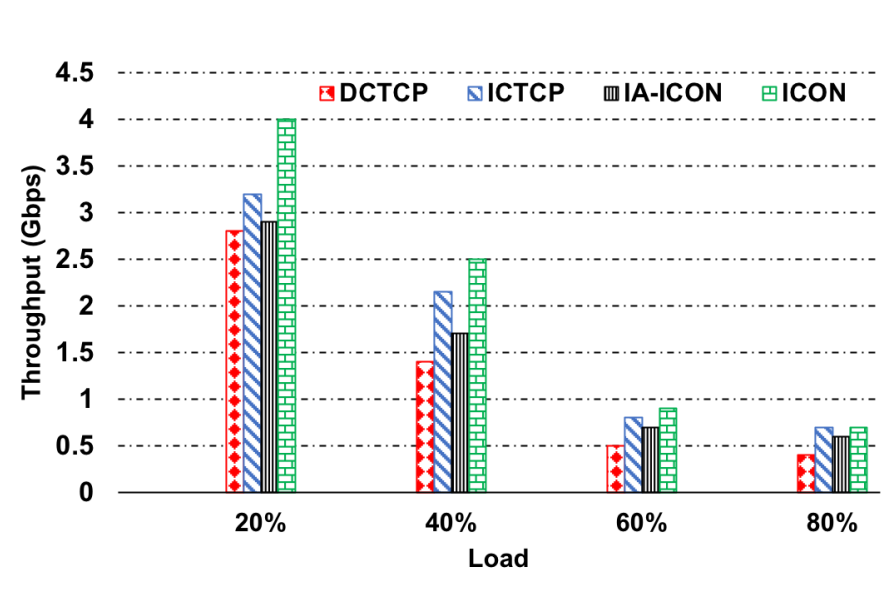

Figure 25. Average throughput of long flows

4.2.6 Sensitivity to different incast degrees

We now study sensitivity of ICON and other schemes as we vary the incast degree of applications. While our default incast degree value of 26 is typical for many datacenter applications, there is, indeed, a wide range in the incast degrees of various applications. In this experiment, we compare the tail flow completion times of foreground flows at an incast degree of 20 and 32 to those at an incast degree of 26, for varying loads. As shown in Figure 27, the tail increases with higher incast degrees. Nevertheless, we see that ICON substantially outperforms the competition, even for lower incast degrees. Because the "real" incast degrees of commercial applications

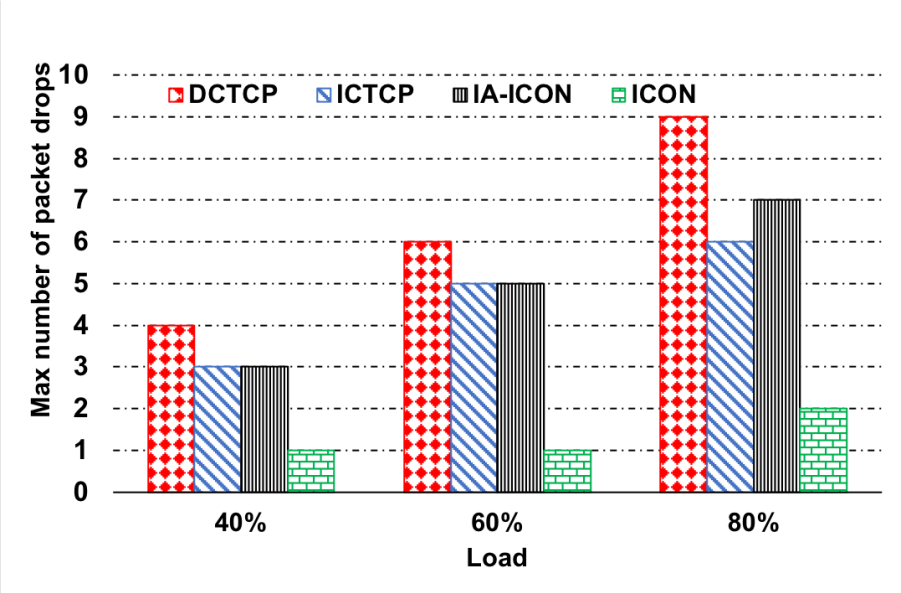

Figure 26. Maximum number of packet drops

are likely higher (e.g., NDP [12] uses incast degrees in the order of a few hundreds), ICON's potential improvements are likely to be even more impressive in production environments.

4.2.7 Queue length

Finally, we analyze the distribution of receiver switch's queue length for the three schemes. We sampled the queue lengths in one of the leaf switches connected to a sink server at 30% load, and we analyzed its cumulative distribution.

Figure 28 shows the CDF of DCTCP, ICTCP, and ICON. We see a *drastic* difference between the CDFs of these three schemes. While both DCTCP and ICTCP suffer from long tails in queue lengths, which corresponds to their long tail flow completion times, ICON's CDF is much

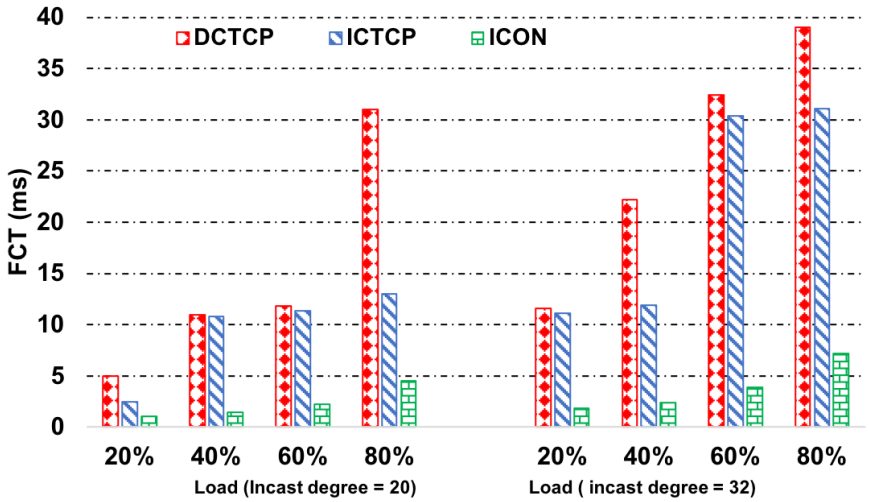

Figure 27. Sensitivity to different incast degrees

narrower than the other schemes. ICON reduces 99^{th} percentile queue length by a factor of 7.5 compared to ICTCP, *closely* matching our improvement in tail flow completion times. This analysis shows that both rate control and pacing play a key role in reducing queuing in the network.

4.3 ResQueue

In this section, we discuss the details of our testbed and workloads, and we provide an exhaustive analysis of ResQueue's performance in terms of flow completion time, throughput, and packet drop rate.

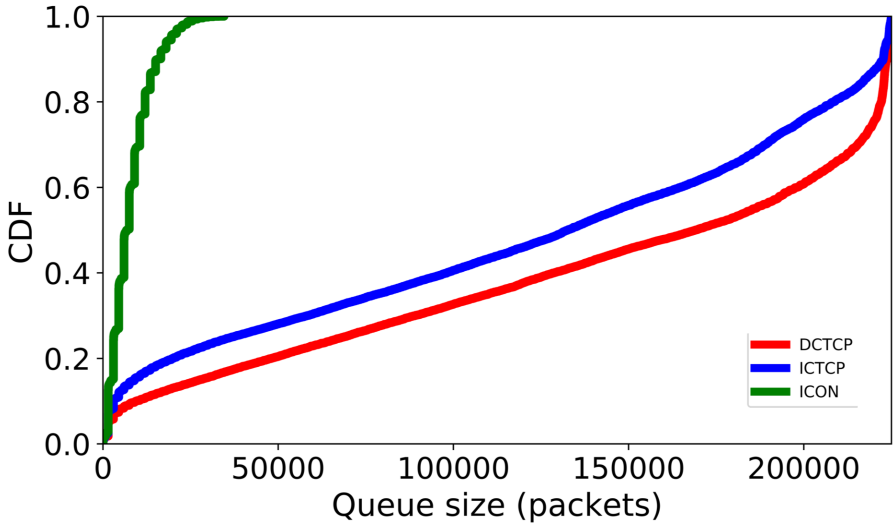

Figure 28. Distribution of queue lengths in switches

4.3.1 Experimental Methodology

We use ns-3 [47] simulator to simulate a leaf-spine datacenter topology, which is common in datacenters [6]. In our topology, the fabric interconnects 400 servers through 20 leaf switches that are connected to 10 spine switches (i.e., there is an over-subscription factor of 2). All links are 10 Gbps and the round trip time across the network is 80 microseconds. We use workload characteristics reported in recent studies [5, 8] to create a realistic traffic in our simulations. Hence, our short flows' sizes are in the range of 1KB to 6KB, while 80% of the flows are 1KB in size. Also, we use 1 MB flows as our large flows. All servers are spread across the network uniformly randomly and the switches use shallow buffers as suggested in prior work (e.g., [6]).

4.3.2 ResQueue Results

4.3.2.1 Flow Completion Times

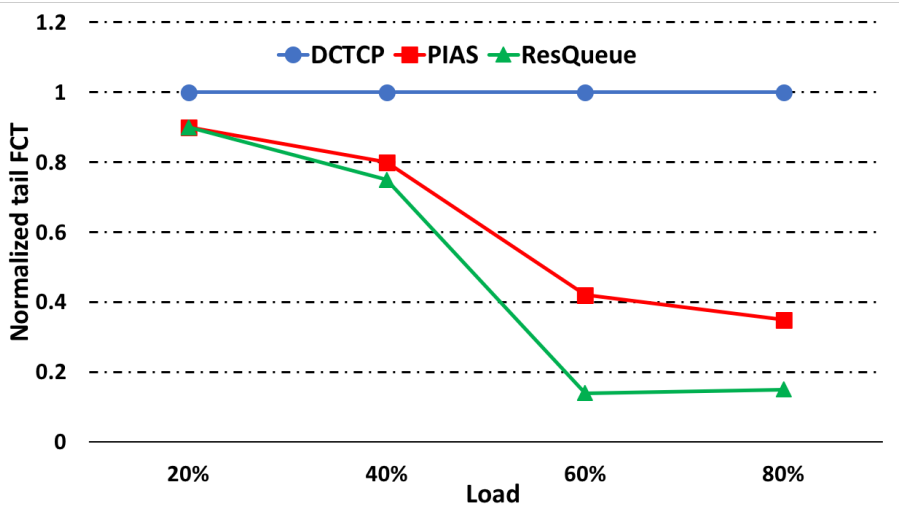

Figure 29. Normalized tail flow completion time of short flows

Tail flow completion time is the key determinant of application performance in datacenters. Thus, we only evaluate ResQueue's performance in lowering *tail* flow completion time of short flows. Note that median flow completion time will be the same in both PIAS and ResQueue, as ResQueue targets dropped packets only, which affects higher percentiles of flow completion time.

Our experiments show that ResQueue improves performance of PIAS (in terms of tail flow completion time) by a factor of 1.6x for loads greater than 20%, on average. Figure 29 shows the result of our experiments when incast degree (i.e., number of concurrent senders) is 32. We discuss more about sensitivity of ResQueue's performance to different incast degrees later in this section.

As we see in Figure 29, although PIAS outperforms DCTCP in lowering tail flow completion times, ResQueue lowers tail flow completion times more by reducing the total number of packet drops. In particular, ResQueue achieves impressive gains of over 2x at higher loads ($\geq$ 60%) over PIAS.

4.3.2.2 Throughput

Although ResQueue is mainly designed for lowering tail flow completion times of short flows, it improves the throughput of large flows as well. ResQueue schedules dropped packets in a higher priority queue depending on their size. Therefore, when some large flows compete for bandwidth, those packets that were previously dropped will get higher priority. For example, if a packet should be scheduled in queue level-4 (based on flow's bytes sent), it will be scheduled in queue level-3 if and only if it is a retransmitted packet. This mechanism will expedite the dropped packets, which provides higher throughput for large flows.

Figure 30 shows throughput comparison between PIAS and ResQueue. While the absolute throughput decreases when load increases due to higher contention with other flows at switches, ResQueue's throughput remains higher than PIAS even at high loads. Instead of repeatedly dropping packets from a small subset of flows, ResQueue distributes packet losses to more flows.

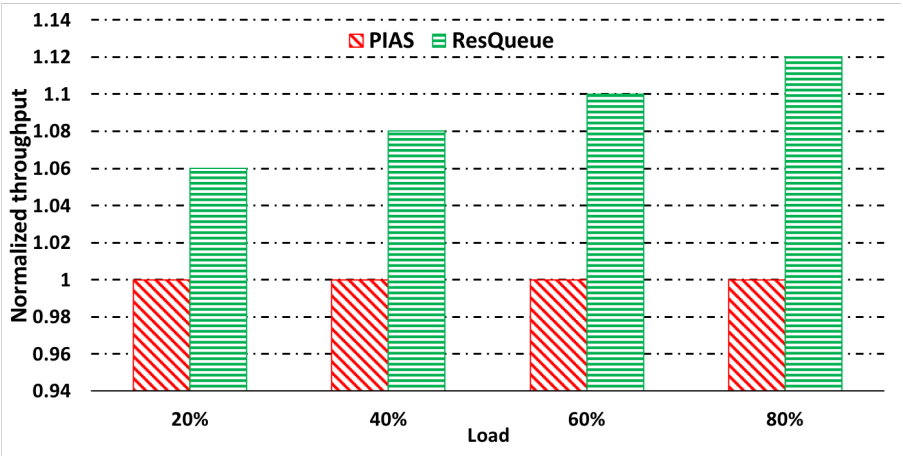

Figure 30. Normalized throughput of large flows

As a result, more senders throttle their rates, which helps alleviate saturation. Figure 30 shows that ResQueue's relative advantage over PIAS increases with load. Overall, ResQueue improves the throughput of large flows by a factor of 1.08x relative to competition, for loads greater than 20%, on average.

4.3.2.3 Packet drop rate

In this section, we analyze packet losses in DCTCP, PIAS, and ResQueue. Figure 31 shows the maximum number of packet drops for *any* packet in DCTCP, PIAS, and ResQueue for different loads. We clearly see that DCTCP and PIAS suffer higher packet loss than ResQueue and their packet drops worsen with load. In contrast, a packet does not get dropped more than once with ResQueue, even at high loads.

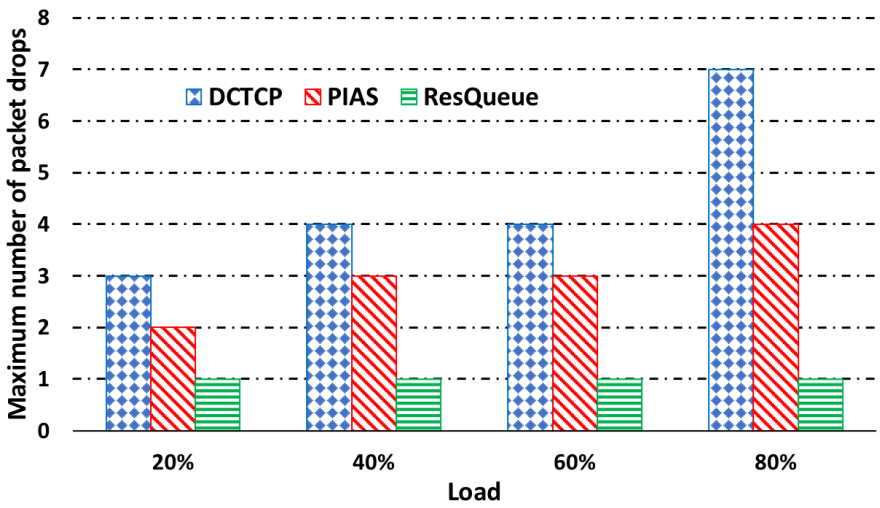

Figure 31. Maximum number of drops for any single packet

Note that our highest priority level is reserved for retransmitted packets only and small flow packets start from the second level and get promoted to the highest priority level upon loss. Because retransmissions constitute a small fraction of packets, we do not observe losses in the highest priority queue (i.e., maximum number of packet drops for any packet in Figure 31 for ResQueue is 1 across all loads).

4.3.2.4 Sensitivity Analysis

(1) **Sensitivity to incast degree:** In this section, we evaluate ResQueue's sensitivity to incast degree. Figure 32 shows the normalized tail flow completion times of ResQueue compared to PIAS (PIAS' tail FCT for each incast degree is normalized to 1). We vary the incast degree as 32 (our default incast degree), 40, and 48 along X-axis, for different loads from 40% to 80%

(typical operating point of datacenter networks). As expected, the tail flow completion times increase when incast degree and load increase. Overall, ResQueue achieves an average reduction in tail flow completion times by a factor of 1.6x, for loads greater than 40%. Thus, ResQueue achieves improvement over PIAS (i.e., normalized FCT less than 1) irrespective of incast degree and load.

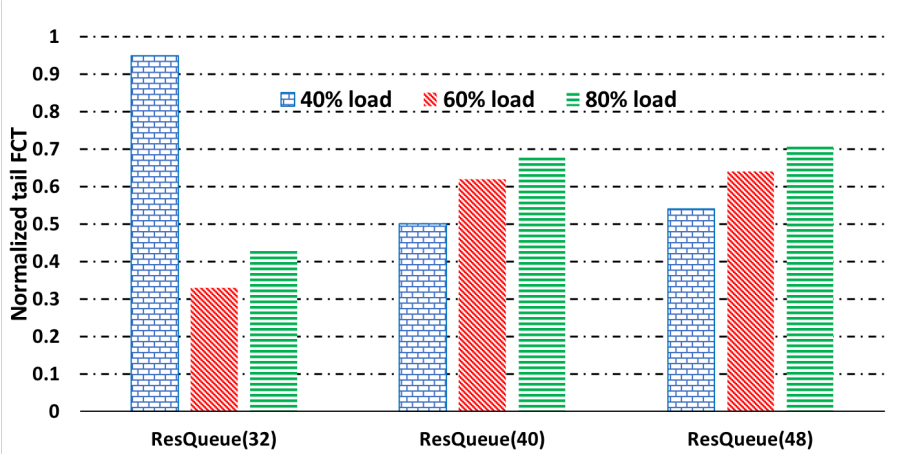

Figure 32. Sensitivity of ResQueue to incast degree (normalized FCT is 1 for PIAS)

(2) **Sensitivity to size of reserved buffer:** ResQueue's main mechanism is to escalate priority of dropped packets by storing them in a higher priority queue. Since we reserve the queue level-1 for those retransmitted packets that belong to short flows, we need to measure

the buffer size that we need to reserve to achieve the highest performance. Because most flows are short, they are highly likely to use this buffer during congestion.

Our analysis shows although a retransmitted packet is likely to collide with a burst of packets, the total number of retransmitted packets is not high.

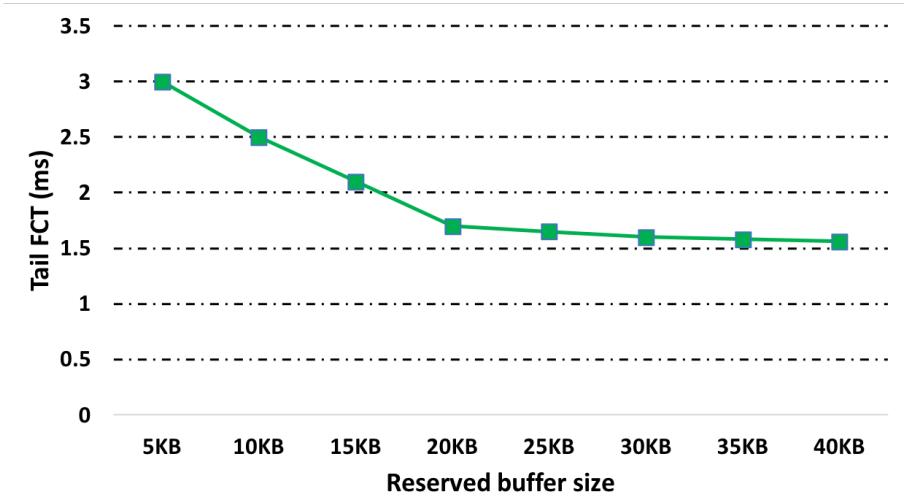

Figure 33. Sensitivity of ResQueue's performance to size of the reserved buffer

Figure 33 shows the sensitivity of ResQueue to size of the reserved buffer. As we see in the figure, there is no difference between performance of ResQueue and PIAS, if the size of reserved buffer is smaller than 5KB. Likewise, sizes larger than 20KB do not improve the performance of ResQueue as the buffer goes unused. Therefore, we have a *sweet spot* between 15KB and

20KB that minimizes packet drops for short flows and achieves high throughput for large flows. Clearly, 20KB of dedicated buffer is not a big amount of buffer and can be easily provided by switch vendors. Modern datacenter switches are equipped with a large (4MB to 50MB) shared buffer that is shared among all ports. Reserving 20KB of a 4MB buffer contributes to only 0.5% of the total buffer, which is almost negligible. Thus, ResQueue's overhead is minimal, and our performance is robust for a range of loads and workloads.

4.4 Superways

In this section we present our evaluation methodology, including details of the topology and the workload, and then we show the results of our experiments, including simulation and real testbed.

4.4.1 Experimental methodology

We use ns-3 [47] simulator to simulate a datacenter network. We implemented all previously discussed topologies including leaf-spine, Bcube [51], Jellyfish [52], and Subways [34] in our simulations. In these topologies, we connect 400 servers through 20 leaf (outer switches in Jellyfish) switches and 10 spine (inner switches in Jellyfish) switches. In tree topologies (e.g., leaf-spine, Subways, and Bcube), each of leaf switches have 20 downlinks to servers and 10 uplinks to spine switches. Also, the network fabric interconnects leaf and spine switches in a full mesh manner. In Jellyfish, however, we connect 20 outer switches to 10 inner switches as per [52]. The link speed between servers and leaf (outer) switches is 10 Gbps and links between leaf switches and spine (inner) switches are operating at 40 Gbps. The longest end-to-end

Round-Trip Time (RTT) across the fabric is 80 microseconds, which is close to that of real datacenter networks. Also, congestion control mechanism in all topologies is DCTCP, and per port buffer size in leaf and spine switches is 240 KB (Superways ns3 code is publicly available at [53]).

4.4.2 Workload and Traffic

We use the workload reported in recent studies on Facebook's datacenter network [5, 8] to create a realistic traffic in our simulations. Our short flows' sizes are in the range of 1 KB to 10 KB, while nearly 70% of the flows are 1 KB in size. Also, as reported in [8], our long flows' sizes are in the range of 100 KB to 10 MB, while half of the long flows are 1 MB or below. All servers are spread across the network uniformly randomly, and 20 of the servers are incast aggregators (i.e., 5% of all servers). Also, we used a combination of short and long flows to produce incast. In our experiments, incast degree varies in the range of 48 to 96 among all incast applications, and incast inter-arrival times match the reported numbers in [5].

4.4.3 Superways Results

In this section, we present the results of our evaluation using (1) ns-3 [47] simulations (4.4.3.1–4.4.3.2), (2) a real implementation using CloudLab [54] (4.4.3.5), and (3) a cost analysis (4.4.3.3). Our performance evaluation consists of 5 parts:

- **Flow Completion Time (FCT)**: In datacenter networks, 99^{th} percentile FCT is the most important metric for measuring short flows' performance [10]. Thus, we do not provide detailed results of median flow completion times. However, our experiments show

that when Superways is implemented on top of the aforementioned topologies, it results in up to 21% improvement in median flow completion time, on average, for all loads.

- **Throughput**: We measure the average throughput of long flows (including those that participate in incast) when Superways is implemented on existing datacenter topologies to evaluate the effectiveness of Superways on throughput of long flows.
- **Cost analysis**: We analyze cost of implementing Superways and Subways in a small scale datacenter. We only compare to Subways because while is similar to Superways, it is the most recent proposal as well.
- **Why connecting to spines**: We will discuss the reasons that convinced us to connect the extra links to spine switches rather than leaf switches.
- **Real testbed**: Finally, we will verify our simulation results in CloudLab [54]. We show the performance of Superways in a real Web search workload, using Apache solr text indexing servers.

4.4.3.1 Flow Completion Time

Figure 35 shows a comparison between 99^{th} percentile flow completion times of regular leaf-spine, Subways, Jellyfish, Bcube, and Superways/leaf-spine. We show the flow completion times along Y-axis versus load on X-axis. As we see in Figure 34, when load increases, tail FCT degrades for all topologies. Longer queuing delays and more packet drops are the main reason for this performance degradation. Compared to regular leaf-spine (shown in blue in Figure 34), Superways/leaf-spine (shown in green in Figure 34) reduces tail flow completion time

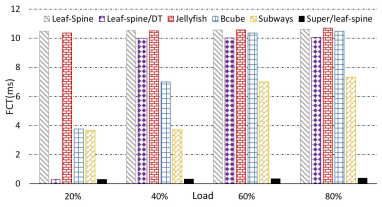

Figure 34. 99^{th} percentile flow completion times

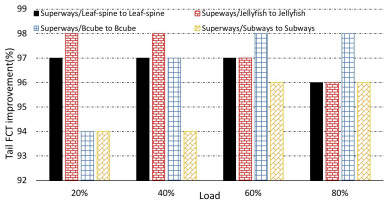

Figure 35. Reduction in 99^{th} percentile FCT

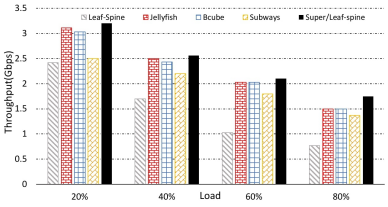

Figure 36. Long flows' average throughput

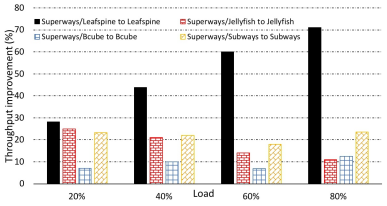

Figure 37. Speedup in long flows' throughput after implementing Superways

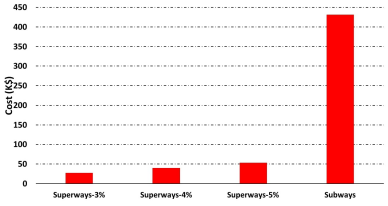

Figure 38. Cost comparison between Subways and Superways

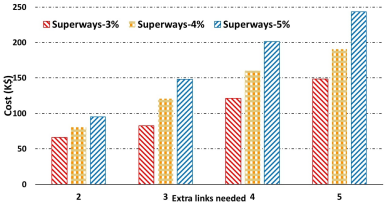

Figure 39. Cost of Superways with high incast degrees and shallow buffers

by about 96% at the loads higher than 40%. Although regular leaf-spine performs worst among the compared topologies, Superways/leaf-spine significantly outperforms other topologies. It shows how efficient Superways is in absorbing incast packets. Subways performance is closer to Superways/leaf-spine because it also provisions extra links for servers. However, since it adds

a static number of extra links for *all* servers, it may not meet all incast applications' buffer demand in case of incast.

Our experiments show that Superways significantly reduces the tail (99^{th} percentile) flow completion times of short flows, when it is implemented on top of other datacenter topologies. We implemented Superways on top of a leaf-spine topology to see how performance of leaf-spine will improve compared to its regular form. We see the results of this experiment in Figure 34. We show the average tail flow completion time of short flows (among various incast degrees) along Y-axis, versus load on X-axis. As we see in the figure, when Superways is implemented on top of a leaf-spine topology, it not only outperforms normal leaf-spine, but it significantly outperforms other complex and expensive datacenter topologies as well. Also, Figure 34 shows the performance of DT, which is the state-of-the-art shared buffer management technology in existing datacenter switches. As we see in the figure, when DT is implemented on all switches in a leaf-spine topology, it works well at lower loads only, which is not the case in many modern datacenter networks.

We implemented Superways on top of other datacenter topologies to see how their performance improves. The result of this experiment is shown in Figure 35. We show the reduction in tail flow completion time along Y-axis versus load on X-axis. As we see in this figure, by implementing Superways on top of Jellyfish (Superways/Jellyfish), flow completion time of short flows will reduce by 97%, on average, for all loads, compared to regular Jellyfish. Also, tail flow completion times in Superways/Bcube and Superways/Subways will reduce by 96% and 95% on average, for all loads, compared to their regular topology. Although Subways and

Bcube both provide additional links for servers, their tail flow completion time reduction rate is close to that of leaf-spine and Jellyfish; which shows Bcube and Subways fail to solve incast problem.

4.4.3.2 Throughput

Superways forwards incast packets on a handful of dedicated links so that they do not collide with other non-incast flows. In other words, long flows will use those links that are not congested, and therefore, their throughput will not be affected by colliding with a burst of short flows.

In Figure 36, we show the average (among various incast degrees) throughput of long flows along Y-axis versus load on X-axis. As we see in the figure, by implementing Superways on top of simple leaf-spine, throughput of long flows improves by several orders of magnitude such that it achieves higher performance compared to all other topologies at higher loads (60%-80%).

Next, we studied the impact of Superways on throughput, when it is implemented with other datacenter topologies. The result of this experiment is shown in Figure 37. While Superways improves long flows throughput in Bcube and Subways by 9% and 20% respectively, leaf-spine topology benefits a lot from deploying Superways. Long flows' throughput in Superways/leaf-spine improves by about 55%, on average, for all loads. Superways improves long flows' throughput in Jellyfish by about 17% on average, for all loads, which shows the throughput efficiency of Superways in both tree and graph topologies.

4.4.3.3 Cost analysis

In this section, we study cost of implementing Superways and Subways on top of a leaf-spine topology. Similar to previous studies on cost comparisons (e.g., [55]), we use price quotes from different vendors to get a clear picture of the equipment cost. We used price quotes from vendors such as FS [56], Dell [57], and retailer websites such as Amazon [58], Compsource [59], and Cablewholesale [60]. We consider vendors such as Dell [57], Juniper [61], Cisco [62], Brocade [63], and Huawei [64] for switches, and intel [65], HP [66], Rosewill [67], and Arista [68] for our NICs. Although prices vary among different vendors, we considered those equipment that satisfy the minimum requirements.

Our studies show that a 48 port 10 Gbps leaf switch would cost nearly $5K, and a 32 port 40 Gbps spine switch costs nearly $11K, on average. Also, we found the value of $80 for a single-port 10 Gbps NIC. Bellow we use these prices to provide an exhaustive analysis on cost of implementation of Superways and Subways.

(1) **Cost analysis (Subways):** We envision a datacenter network similar to the one in [34]. We consider a small datacenter network with 48 leaf switches and 24 spine switches that are connected in a leaf-spine topology. We assume each leaf switch is connected to 48 servers and each rack contains 48 servers plus the leaf switch that connects to them. We implement Subways type-1 (p=2), which requires an additional link from every server to its neighboring rack. Assuming a 42U rack with 6.5 ft height and cold aisle of 3.9 ft, on average, each server requires a 5.2 ft cable to connect to the neighbouring rack's leaf switch. Our price analysis on cable cost shows that a 10 ft long cat7 cable will cost about $5, which shows that we need to pay

$11520 (i.e., $48 \times 48 \times \$5$) for the extra cables. Although Subways proposes a new way of wiring that decreases the cable cost, it does not significantly reduce the overall implementation cost, because cabling costs contribute to a small fraction of total cost of implementation. Subways needs more leaf switches to keep the over-subscription ratio of the network unchanged, and therefore, we need to pay $48 \times \$5,000 = \$240,000$ for extra leaf switches. Finally, assuming Subways type-1 with only one extra link per each server, we need to buy one more NIC for all servers, which costs $48 \times 48 \times \$80 = \$184320$. In total, we need to pay extra $435,840 to implement Subways on a leaf-spine topology with 2304 servers.

(2) **Cost analysis (Superways):** Superways' cost is highly dependent on number of incast applications, incast degrees, spines' link speed, and buffer sizes at leaf and spine switches. Bellow, we analyze Superways' cost in different scenarios when each incast application requires different number of extra links to absorb all incoming packets. Assuming that 3% of all servers host incast applications (70 out of 2304), and each incast application requires *one* more link to absorb all packets, we need to provide 70 extra links to connect incast servers to spine switches. Assuming 42U racks and a rectangular server room with two rows of racks and the rack containing spine switches located at the end of one of the rows, we need 50 ft long cables to connect the incast servers to spine switches (worst case scenario). Our price analysis on cable cost shows that a 50 ft cat7 cable will cost about $20, which shows that we need to pay $70 \times \$20 = \$1400$ for extra cables. Since we need to add one extra NIC for those servers that hold incast applications, total cost of purchasing new NICs is $70 \times \$80 = \$5600$. In the extreme case that existing spine switches do not have enough available ports, we need to buy two more

spine switches to connect to incast servers. The total cost of two extra spine switches is nearly $22K. Therefore, if each incast server requires one extra link, the total implementation cost of Superways will be around $28820.

Figure 38 shows the cost comparison between Superways and Subways type-1. This figure shows the total cost of Superways for 3 different cases when 3%, 4%, and 5% of all servers are incast applications (aggregators). Note that performance of all four schemes is the same because each server (each incast server in Superways) has only one extra link. As we see in Figure 38, even if 5% of all servers are hosting incast applications, which is a relatively large number, Superways is still about 9x cheaper than Subways. Also, Figure 39 shows different cases of Superways when more than one extra link is required per each incast server. This is the case when incast degrees are extremely high, link speeds of both leaf and spine switches is the same (e.g., all are operating at 10 Gbps, which is not high), and buffers are shallow. Although having too many links for each incast server is a rare case, even with high number of extra links, Superways still costs much less than Subways, which is due to its heterogeneous design that focuses on one area of the network only. In other words, Superways might need more than one link for each incast server depending on the burstiness of traffic, but, since it does not need too many extra switches and it only requires *some* servers to have extra links, the total cost of implementation remains low compared to other topologies with redundant links.

4.4.3.4 Why connect to spine switches?

There are both performance and cost related factors involved in Superways' design. By connecting to spine switches, which are more powerful in terms of processing speed and band-

width, the amount of time that incast packets stay in a port's queue will reduce compared to leaf switches, and therefore, their average and tail FCT improves. Moreover, by providing such dedicated links from spine switches to incast applications, long flows will no longer collide with a batch of short incast flows on the spine switches, and therefore, they will not suffer delay specially when packet prioritization is enabled in the network (i.e, those short incast flows will always get the highest priority). Finally, due to large difference in link bandwidth of spine and leaf switches, the number of links, number of server side NICs, and total number of required switches will significantly reduce, which significantly reduces the overall cost of implementation if we connect the extra links to spine servers.

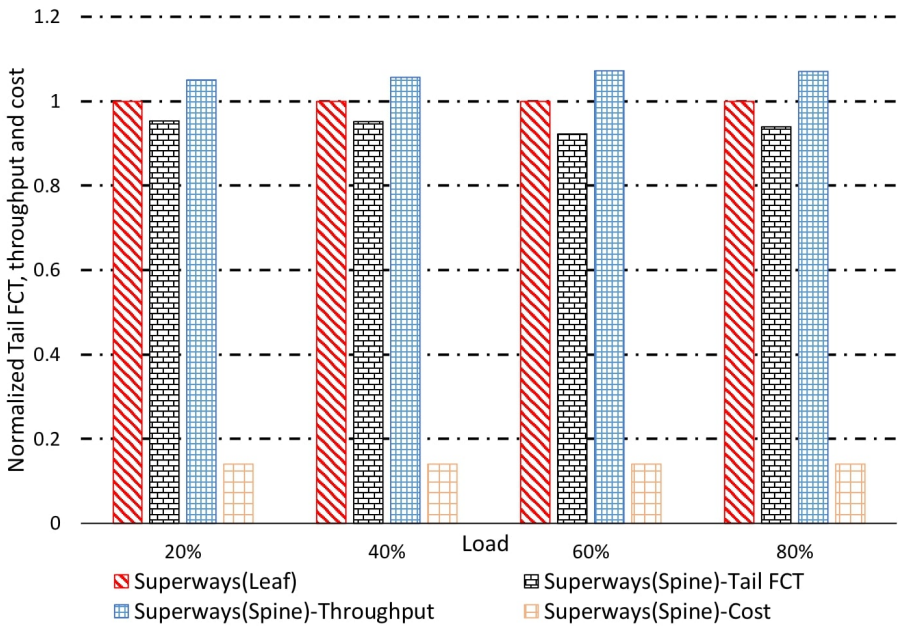

Figure 40. Normalized tail flow completion time of short flows

Figure 40 shows a comprehensive picture of comparing the overall performance of Superways when we connect the extra links to spine switches versus connecting to leaf switches. As we see from the figure, while connecting to spine switches improves both throughput and tail FCT by up to 8%, it reduces the cost of implementation by about 7x, which is a considerable amounts of money in medium to large scale datacenters. Thus, connecting the extra links to spine switches would be the best option as it improves performance and reduces cost of implementation, while adding a little complexity to the routing mechanism (i.e., we can easily update the routing table on the SDN controller through OpenFlow messages).

4.4.3.5 Real testbed

We implemented a leaf-spine topology in CloudLab [54] to validate our simulation results and to evaluate metrics such as CPU usage and network I/O utilization, which cannot be done on a simulator. Our real testbed emulates 16 servers, 4 leaf switches, and 2 spine switches. The over-subscription ratio in the leaf-spine topology is 2:1, and the leaf switches are connected to spine switches in a full mesh manner. Also, using the same number of servers and switches, we implemented Bcube, Subways, and Jellyfish to validate the efficiency of Superways when it is implemented on top of these topologies. Bcube, Subways, and Jellyfish are implemented as per [51], [34], and [52]. Our servers run Ubuntu 16.04 (kernel version 3.3) and our switches run Open vSwitch version 2.31. We rate limit all switches to 1 Mbps, and, also, we set the transmit queue size of spine and leaf switches to 10KB.

To create a realistic workload, we used the reported numbers in [8] and [5]. Our short and long flows are 4 KB and 1 MB in size, and our incasts are a mix of short and long flows. While

TABLE III

Improvements in flow completion time and throughput - Real testbed and simulations

Topology	Testbed results		Simulation results	
	Tail FCT	Throughput	Tail FCT	Throughput
Super/leaf-spine	85.12%	38.4%	96.75%	50.7%
Super/Jellyfish	81.4%	16.3%	97.25%	17.1%
Super/Bcube	83.7%	7.1%	96.75%	11.5%
Super/Subways	80.04%	15.5%	95%	20.5%

we used iperf3 to produce all-to-all traffic between random servers, we installed *Apache Solr* [69] version 8.2.0 on certain servers to emulate a real Web search traffic (including incast). These Apache Solr servers generate light weight queries that ask other servers about a specific event that they already saved in their database. We created incast traffic so that all incast senders send synchronous replies to the Apache Solr servers.

4.4.3.6 Flow completion time and throughput

We provisioned three incast applications (running Apache Solr server v8.2.0) with incast degrees of 9, 3, and 3 that are co-located on one of our servers according to calculations described in chapter 3. We repeated the experiments 10 times to accurately measure the performance of Superways in all topologies. Table III shows the improvement in 99^{th} and 50^{th} percentile flow completion times of short flows when Superways is implemented on top of other topologies. As

we see from the table, our testbed results are close to simulation outputs, which endorses the efficiency of Superways in improving tail FCT of short flows. We measured the throughput of long flows as well. Since long flows will no longer collide with bursts of short flows at the network core, their throughput improves by up to 38.4%, as we see in Table III.

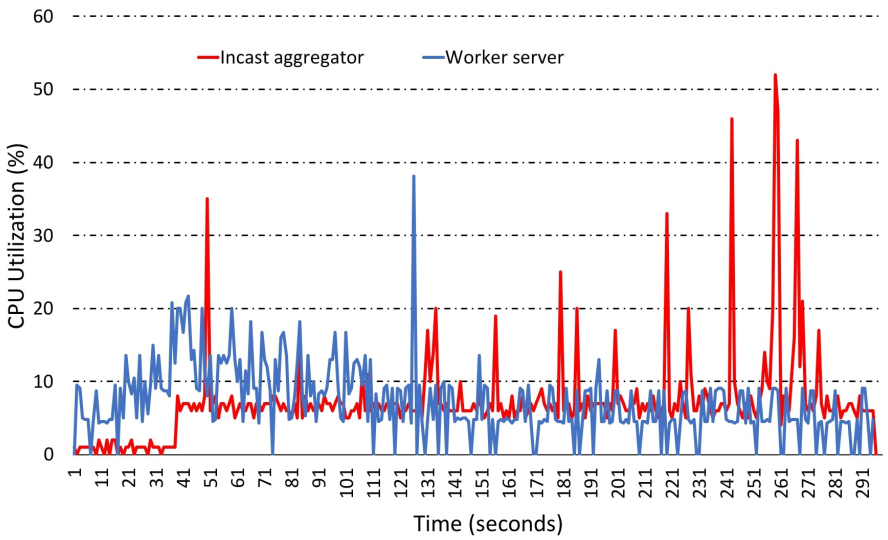

Figure 41. Normalized tail flow completion time of short flows

4.4.3.7 CPU utilization vs bandwidth utilization

Co-locating more than one incast application on a single server may arise concerns about CPU usage of that particular server. We designed an experiment to see how CPU utilization varies among those servers that participate in all-to-all traffic, and that of incast aggregators.

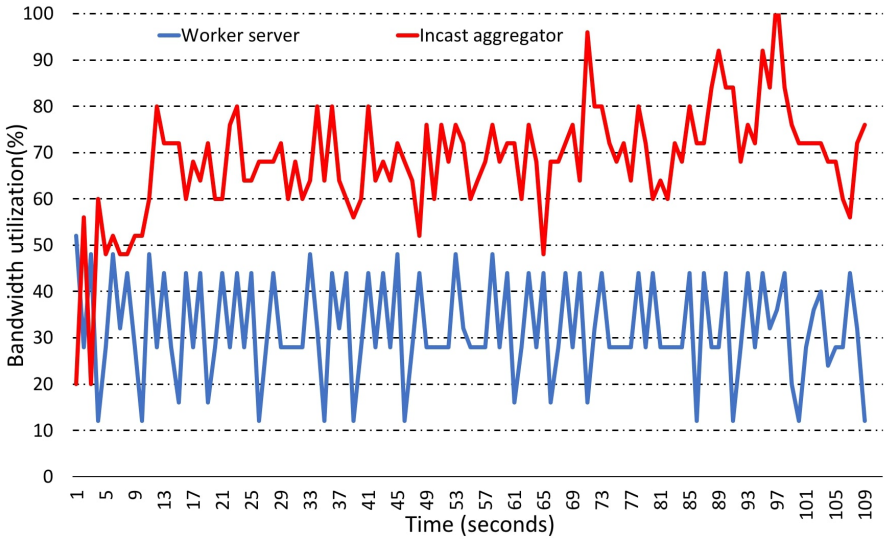

Figure 42. Normalized tail flow completion time of short flows

Figure 41 shows the result of this experiment. The red line shows the CPU utilization of a server with Apache Solr V8.2.0 installed on it (incast application) and the blue line shows CPU utilization of a random server that transmits both short and long flows using iperf3. As we see in the figure, CPU utilization of both servers is almost the same, so that the average CPU utilization of the incast aggregator is 8.19%, while this value is 7.08% for the other server. Therefore, co-locating incast applications on a single server will not over-utilize the CPU as incast applications do not run compute-intensive tasks. On the other hand, we claimed throughout the paper that incast aggregators require more bandwidth compared to other servers due to their high number of incoming packets. To verify this assumption, we measured the bandwidth utilization for an incast aggregator and a random server during a short period of

time. Figure 42 shows this comparison. As we see in this figure, the incast aggregator utilizes bandwidth much more than the other server so that while average bandwidth utilization is 17% for an ordinary server, this value is 68% for an incast aggregator. Thus, incast applications are indeed network bound and providing more bandwidth for them is necessary to guarantee the highest performance.

4.5 Smartbuf

We conduct experiments to evaluate three different aspects of our approach: (1) performance including tail latency for short flows, throughput/fairness for long flows, (2) overhead of our algorithm, and (3) accuracy and parameter sensitivity of our buffer-demand estimation.

4.5.1 Methodology

We simulate a leaf-spine datacenter topology in which 20 leaf switches are each connected to 20 servers. The leaf switches are connected upstream to 10 spine switches, thus creating an over-subscription factor of 2. We implement Smartbuf in ns-3 [47] and run it on leaf switches while spine switches are running the traditional dynamic threshold policy. While ns-3 does not support shared memory switches, we created models of shared memory switches with a shared buffer size of 4 MB. We model a web search workload [10] with short flows in the range of 1 KB to 32 KB and long flows varying from 1 MB to 10 MB in size. While we vary the overall network load in our experiments, we set the long flows to contribute to 80% of the load [6]. We set the parameters used for port buffer initialization according to [35]. while link speed of leaf switches and spine switches is 10 Gbps across the network, we set the network round

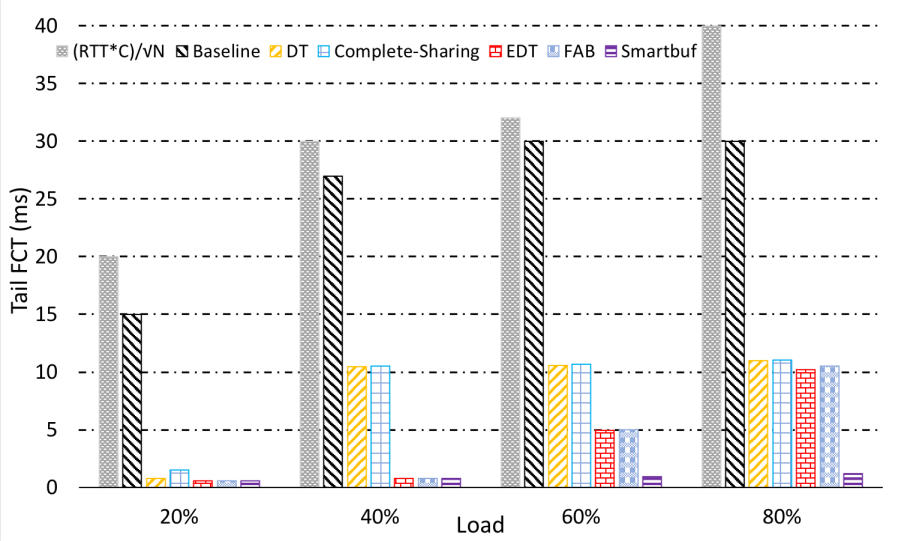

Figure 43. 99^{th} percentile flow completion time

trip-time to 80 microseconds, which is very close to that of modern datacenters. Also, we use DCTCP [10] as our congestion control method in which the retransmission timeout is set to 10 ms. In our experiments, we compare Smartbuf to various policies including Static partitioning, DT [35], Complete sharing, EDT [36], and a recent work called FAB [70]. FAB [70] extends DT by using multiple parameters per port (see α in [35]) depending on flow priority (e.g., flow size). While FAB assigns a larger fraction of buffer to high priority flows, it does not estimate actual demands and therefore suffers from similar shortcomings.

4.5.2 Smartbuf's results

First, we compare the 99^{th} percentile flow completion times (FCT) under different loads for the following schemes: complete (static) partitioning with an equal per-port share as the

baseline, partitioning with a per-port allocation of size $(RTT \times C)/\sqrt{N}$ as in [71], Complete sharing, which allows queues to grow arbitrarily large until the buffer is full, DT, EDT, FAB, and Smartbuf. Parameters used in DT, EDT, and FAB are chosen according to [35], [36], and [70] respectively. Figure 43 shows their 99^{th} percentile flow completion times (FCT) vs. load.

As expected, complete partitioning is highly susceptible to transient bursts which is reflected in its much higher FCT. The per-port buffer space cap in [71] (i.e., $(RTT \times C)/\sqrt{N}$), derived for the Internet, does not seem enough for datacenters with highly synchronized short flows that cause incast. Also, as we see in Figure 43, complete sharing is not able to eliminate incast induced packet drops because it will be fooled by large flows. In other words, complete sharing allows large flows to occupy up to the whole buffer, and therefore, there will be no space left for short-lived incast flows.

While DT allows for sharing memory between ports, it always keeps a fraction of the buffer unallocated at any point and therefore does not absorb incasts; this observation was also made in other papers [36, 70]. In these conditions, the uncontrolled state in EDT temporarily sets the port threshold to the whole buffer size, and then in the controlled state when the burst is gone, it reverts to DT. This helps the tail FCT because microbursts are expected to last for a short time only. Recall from chapter 3, although EDT accurately detects bursts, it does not predict their actual buffer demands and therefore cannot *proportionally* allocate buffer space to ports based on their demands. AT low loads, EDT performs well. However, at high loads, when there is a higher likelihood of multiple bursts on different ports closer in time, EDT

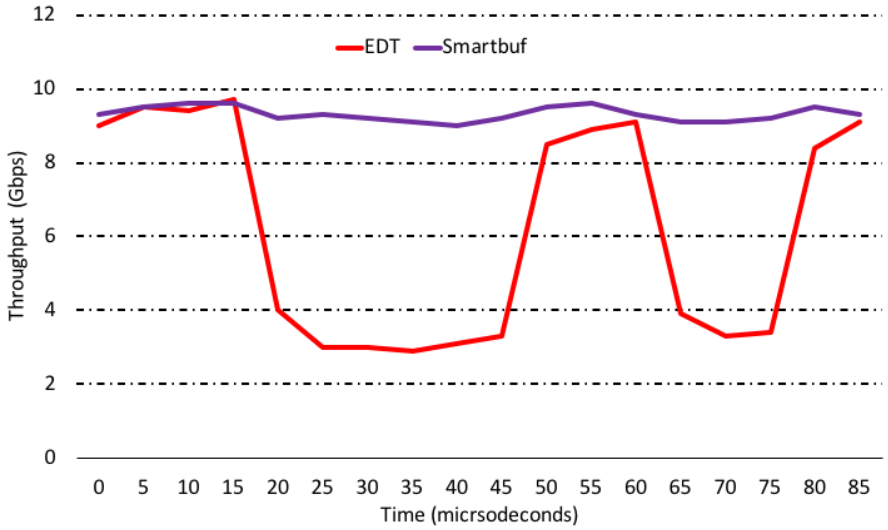

Figure 44. Non-burst flows' buffer share in EDT vs. Smartbuf

makes sub-optimal allocations and suffers from longer tails. As we see in Figure 43, Smartbuf outperforms all the other schemes across all loads. Note that retransmission timeout (RTO) is 10 ms in our experiments, and even at the highest load, Smartbuf nearly eliminates packet drops and improves tail FCTs by a factor of 8. Because FAB's performance is very close to that of EDT, we only consider EDT in our experiments form now on.

We conducted experiments with varying buffer sizes and found that as long as the buffers are not too small (no opportunity) or too large (no buffer contention), Smartbuf's relative performance remains robust. Although long flows benefit less from buffering than short flows, aggressively depriving them of buffer space during the bursts degrades their throughput and

causes fairness issues. For fairness, we designed an experiment in which a long flow on one port competes with an incast on another port. We plot the throughput of this long flow for EDT and Smartbuf in Figure 44. We see a sudden drop in EDT's long flow throughput as most of the buffer space is allocated to the incast port, irrespective of its demand. In contrast, Smartbuf's proportional buffer allocation helps alleviate incast without adversely affecting the throughput and fairness for long flows.

4.5.2.1 Parameter sensitivity

In this section, we discuss the sensitivity of our algorithm to two main factors that we mentioned in Algorithm 2 β and σ. Also, we discuss the sensitivity of our scheme to different values of k in the k-nearest neighbors algorithm. Figure 45 shows the sensitivity of Smartbuf's performance to k for a high load case (other loads not shown due to brevity). As expected, smaller values of k impose less overhead but suffer from inaccurate extrapolation of buffer demands. We achieve the best trade-off between overhead and accuracy at $k = 3$ and further increase in k results only in diminishing returns.

Later, we show that our algorithm's predicted buffer demand closely matches actual demands at $k = 3$, thus providing a desired high accuracy (Figure 48).

The parameter β $(0 < \beta < 1)$ is a threshold on buffer occupancy at any of the output ports, which determines whether the current burst has ended. If we determine that the burst has ended, we will insert updated values for gradient and buffer occupancy into our database. Thus, smaller values of β may mislead us to miss some bursts (i.e, lower performance), whereas larger values may cause us to record one large burst as several smaller bursts (i.e., high state

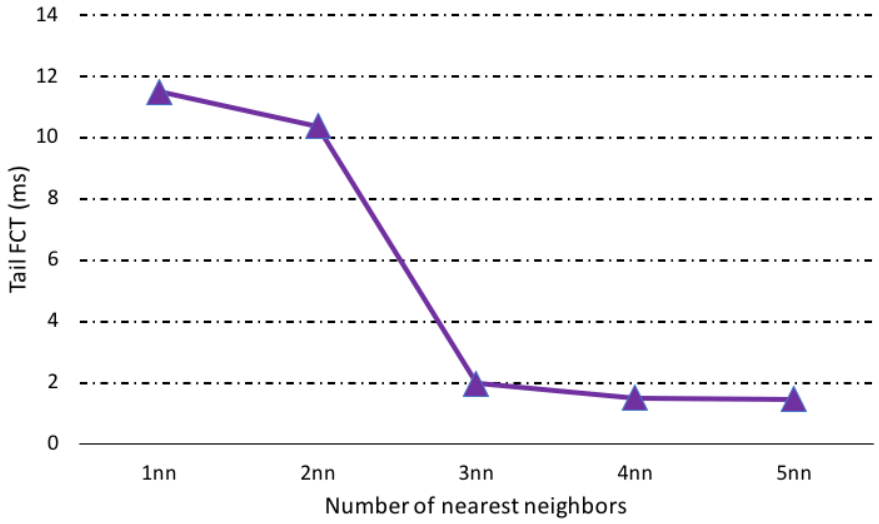

Figure 45. Sensitivity study of k values

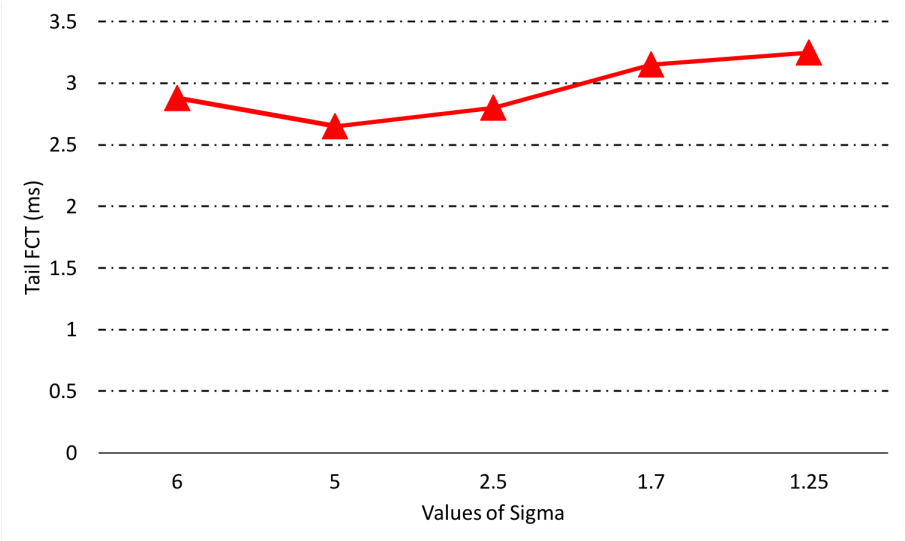

Figure 46. Sensitivity to σ at 80% load

overhead). Our experiments show that if β is chosen between 0.2 and 0.5, it reduces the number of entries in the Main_map by 44% compared to β=0.9, while improving the performance (tail latency) by 9% compared to when β is in the range of 0.01 to 0.19. We observed good, stable performance for β in the range of 0.2 to 0.5.

Finally, we study the sensitivity of our algorithm to different values of σ. σ ($\sigma > 1$) is a threshold that classifies spikes in buffer usage as burst and non-burst. In other words, if buffer occupancy is larger than a certain threshold, switch decides to save this value because this burst has the potential to occupy more buffer and there is a risk of packet drop. Figure 46 shows the result of this experiment. We see that we get better performance for larger values of σ. While larger σ provides high performance, it requires more processing as the size of the database is considerably larger. Therefore, we opt for the smallest value of σ that guarantees high performance (e.g., $2.5 < \sigma < 5$ in Figure 46).

4.5.2.2 Overhead

Next, we analyze the state/memory overhead of our algorithm. Figure 47 shows the maximum number of key-value pairs stored in our map under different loads (in a shared 4 MB buffer). As expected, we observe that the number of entries in our database (i.e., map) scales proportionally to the load. Nevertheless, we see that even in the highest load, we require less than 35 entries, which shows that our design is feasible in practice. Further, as most datacenters operate at loads between 20% and 40%, our tables require less than 20 entries.

If one were to implement our table lookup in hardware, these results imply that the hardware (ASIC) design is feasible. A recent paper, pFabric [6] shows that one can find a minimum of

600 entries ($log\ (N)$ complexity using binary search) within about $log(600) \approx 10$ clock cycles. Assuming the worst case of minimum size 64B packets, each packet dequeue requires 51.2 ns (in a 10 Gbps link), which translates to 40 clock cycles in a modern ASIC. Nevertheless, in our case we need to find 3 nearest neighbors. Finding each nearest neighbor is equivalent to finding a minimum of the distance between the query and each entry. However, the number of entries in our case is quite small (i.e., about 30 entries in the worst case from Figure 47). So, we require about $log(30) \approx 5$ clock cycles for finding one neighbour. Therefore, we require only about 15 clock cycles even for a naive implementation in the worst case, which still leaves about 25 clock cycles for the rest of the algorithm, which is plentiful as the other operations are not computationally intensive and many operations can be done in parallel. As a last point, to scale the table lookup to higher speeds (e.g., 100 Gbps), instead of performing one comparison in each stage of the binary tree, it is feasible to have a combinatorial hardware block to maximize the number of operations in each cycle, thanks to the extremely small footprint of our database (map).

Smartbuf adds a bit more latency when it checks the gradient of queue length and performs all the operations in lines 10 through 30 of the algorithm 2. However, there are less than 20 operations in this block, which means the whole algorithm (including finding 3 nearest neighbors) costs nearly 20 CPU cycles, which translates to 10 nanoseconds in a 2 GHz CPU. Therefore, Smartbuf's latency is 12 times less than packet dequeue delay at the fastest existing link (i.e., 100 Gbps link), which makes it quite feasible to implement in hardware switches.

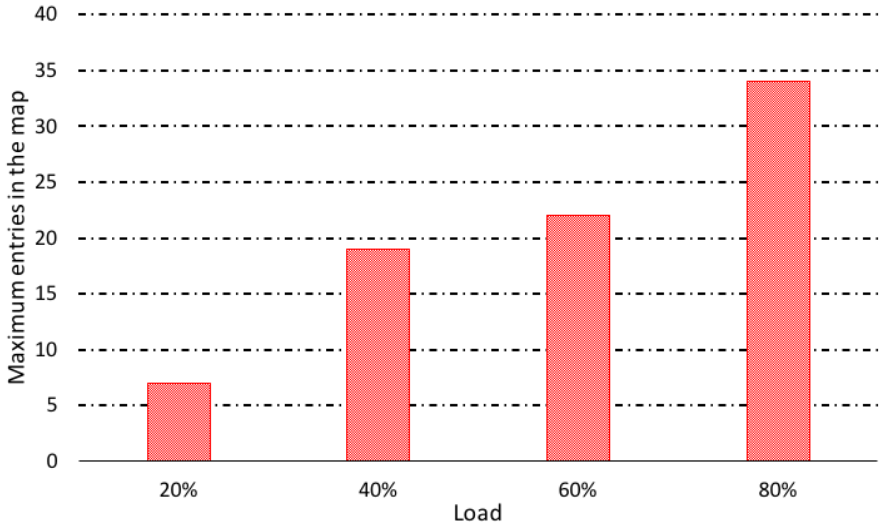

Figure 47. Number of entries in the hash table

4.5.2.3 Accuracy

In this section, we quantify the accuracy of Smartbuf. To quantify how precisely Smartbuf allocates buffer according to flows' requirements, we run an experiment in which we allow flows to take as much memory as they need from a hypothetically unbounded memory with the complete sharing policy and compare it to the amount predicted by Smartbuf. Figure 48 shows predicted demands vs. ideal (oracular) demands for each load level when the difference between the two (error) is **maximum**. Note that while it appears that our predicted demand is always less than the oracular case in the figure, it shows only cases when the error is maximum; in reality, across all bursts, there are cases when we predict a larger demand than what it really is.

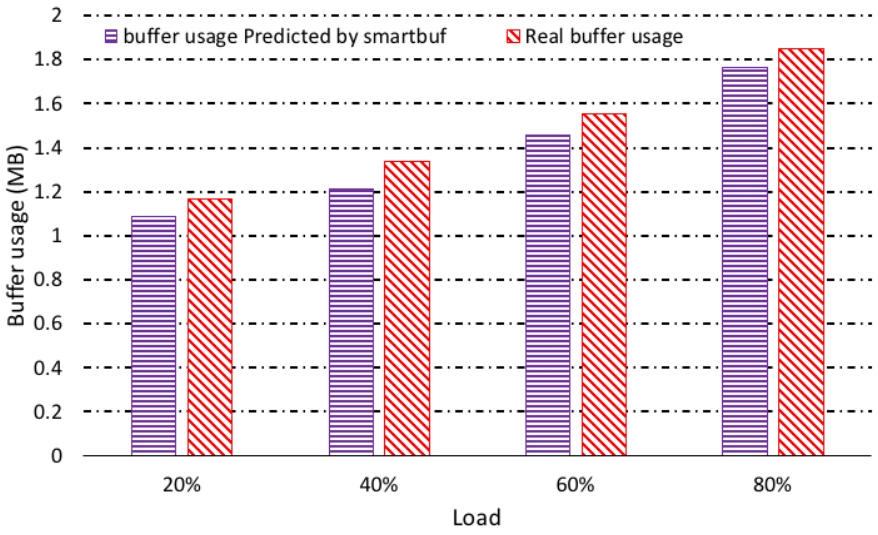

Figure 48. Predicted vs. actual buffer demands: maximum prediction error at each load

The key takeaway is that our predicted demands are very close to the oracular buffer demands (within 9% of the ideal).

Note that depending on the index of the matched key and its nearest neighbors, under-provisioning would not always be the case as shown in Figure 48, but it could happen more often for the largest bursts as their signature is more likely to fall out of the range of our keys and we effectively would pick the $k = 1$ nearest value in that case.

CHAPTER 5

CONCLUSION

We proposed five different approaches for tackling incast problem in datacenter networks. Incast is mainly a many-to-one traffic pattern that exists in most of modern datacenter networks that host Online Data Intensive (OLDI) Applications. These applications tend to collect small portions of data and then represent it to the end user.

First, we presented Slytherin, which identifies tail packets and prioritizes them in the network switches. Unlike prior approaches that emulate SJF scheduling of packets which is well-known to minimize average flow completion times, Slytherin optimizes tail flow completion times, a metric that is more relevant for many online datacenter applications (e.g., Web search, Facebook). Slytherin does not require extensive support for identifying tail packets and leverages already available congestion signals (i.e., ECN). Using realistic workloads on typical datacenter network topologies, we showed that Slytherin reduces tail flow completions by about 18% as compared to existing state-of-the-art schemes. Further, we also showed that Slytherin drastically cuts the queue lengths and speeds up convergence. As data continues to grow at a rapid rate, schemes such as Slytherin that minimize network tail latency will become even more important.

Next, we proposed ICON with a goal of improving tail latency for foreground applications and throughput for background applications, by avoiding queue buildups that happen due to incast at edge switches. While there has been a number of papers that show incast-induced

congestion is common in datacenters, the problem remains largely unsolved. Incast is also inherently tied to the nature of many foreground applications that perform distributed search of small objects in large datasets. Incast cannot be fixed by having large buffers as large buffers cause increased queuing delays and exacerbate the problem even further.

We introduced ICON, which employs fine-grained traffic pacing at the end host's NIC. Fortunately, many vendors are starting to introduce support for packet pacing in NICs, which augurs well for our proposal. In addition to pacing, ICON also leverages application knowledge to arrive at the right sending rate, without having to continuously adjust the sending rates over several RTTs. ICON reduces the tail percentile flow completion times of short flows by *up to* 89% compared to DCTCP. It also improves the throughput of large flows by up to 30% compared to DCTCP. As incast degrees of foreground applications increase to cope up with the exponential growth of data, schemes such as ICON would become attractive.

Next, We presented ResQueue, which identifies delayed packets and prioritizes them in switches. Unlike prior approaches that rely only on flow size to determine priority, ResQueue uses a combination of flow size and packet drop history to infer priority. ResQueue improves tail flow completion times, which is the key metric for a broad class of user-facing datacenter applications. Further, ResQueue does not require hardware changes at switches. The current trend of increasing traffic burstiness in datacenters combined with the emphasize on low tail flow completion times necessitate schemes such as ResQueue that consider packet history in flow scheduling.

In future, we plan to improve ResQueue's accuracy in assigning flow priorities by combining a Shortest Remaining Processing Time (SRPT) scheme with counting the total number of drops in a flow. It would be more powerful scheme as it assigns more realistic priorities to large flows that suffered packet drops.

Next, we presented Superways, a heterogeneous datacenter topology that is designed for incast-heavy workloads. Based on our key observation that in most incast applications a few aggregator processes are bottlenecked by receiver bandwidth, our proposal provides additional bandwidth (links) for a small subset of nodes. Further, we proposed a heuristic for scheduling critical aggregator processes in our heterogeneous topology. Our insight for the scheduler is to greedily find the optimal number of extra links by measuring the residual buffer of the extra links after bin-packing incast aggregators. By connecting incast aggregators to the spine switches, Superways decreases the packet drop rate and improves the tail flow completion times of incast flows. When combined with existing load balancing schemes, Superways distributes the incast flows among multiple receive links, avoiding flow collision between incast and non-incast flows. Furthermore, Superways improves fault tolerance and throughput while reducing the total cost of implementation compared to other schemes such as Subways. As incast-heavy applications become more prevalent, heterogeneous topologies and their associated job scheduling strategies such as our proposal would become increasingly important.

Finally, We present Smartbuf, an agile memory management scheme for shared-memory switches in datacenters that accurately predicts the buffer demands of ports based on switch-local information and allocates the memory proportional to their demands. Our experiments

show that Smartbuf outperforms existing state-of-the-art in tail latency by up to a factor of 8x at high loads without sacrificing fairness among ports. With the advent of programmable data planes, bursty datacenter traffic, and ever-increasing line rate, online learning approaches that rely on only switch-local information will become appealing for resource management.

APPENDIX

COPYRIGHTS

APPENDIX (Continued)

IEEE COPYRIGHT AND CONSENT FORM

To ensure uniformity of treatment among all contributors, other forms may not be substituted for this form, nor may any wording of the form be changed. This form is intended for original material submitted to the IEEE and must accompany any such material in order to be published by the IEEE. Please read the form carefully and keep a copy for your files.

Slytherin: Dynamic, Network-assisted Prioritization of Tail Packets in Datacenter Networks
Hamed Rezaei, University of Illinois at Chicago, United States; Mojtaba Malekpourshahraki, University of Illinois at Chicago, United States; and Balajee Vamanan, University of Illinois at Chicago, United States
2018 27th International Conference on Computer Communication and Networks (ICCCN)

COPYRIGHT TRANSFER
The undersigned hereby assigns to The Institute of Electrical and Electronics Engineers, Incorporated (the "IEEE") all rights under copyright that may exist in and to: (a) the Work, including any revised or expanded derivative works submitted to the IEEE by the undersigned based on the Work; and (b) any associated written or multimedia components or other enhancements accompanying the Work.

Figure 49. Copyright - Slytherin

IEEE COPYRIGHT AND CONSENT FORM

To ensure uniformity of treatment among all contributors, other forms may not be substituted for this form, nor may any wording of the form be changed. This form is intended for original material submitted to the IEEE and must accompany any such material in order to be published by the IEEE. Please read the form carefully and keep a copy for your files.

ICON: Incast Congestion Control using Packet Pacing in Datacenter Networks
Mr. Hamed Rezaei, Mr. Muhammad Usama Chaudhry, Mr. Hamidreza Almasi and Mr. Balajee Vamanan
2019 11th International Conference on Communication Systems &Networks (COMSNETS)

COPYRIGHT TRANSFER
The undersigned hereby assigns to The Institute of Electrical and Electronics Engineers, Incorporated (the "IEEE") all rights under copyright that may exist in and to: (a) the Work, including any revised or expanded derivative works submitted to the IEEE by the undersigned based on the Work; and (b) any associated written or multimedia components or other enhancements accompanying the Work.

Figure 50. Copyright - ICON

APPENDIX (Continued)

IW3C2 Copyright Release Form

Title of work:	**ResQueue: A Smarter Datacenter Flow Scheduler**
Author(s):	**Hamed Rezaei;Balajee Vamanan (University of Illinois at Chicago)**
Description of material:	**WWW Paper**
Title of IW3C2 Publication:	**WWW '20: The Web Conference 2020**

I hereby assign (to the extend transferable, see point B below) to the International World Wide Web Conference Committee (IW3C2) the copyright of this Work for the full period of copyright and all renewals, extensions, revisions and revivals together with all accrued rights of action throughout the world in any form, including as part of IW3C2 and the public Conference Web site, on CD-ROM and in translation, or on videocassette, broadcast, cablecast, laserdisc, multimedia or any other media format now or hereafter known. (Not all forms of media will be utilized.) I accept that IW3C2 will allow the Association for Computing Machinery (ACM) to distribute, make available on the Internet or sell this Material as part of the above-named publication in the ACM Digital Library. Finally, I also accept that IW3C2 will publish this Work under the Creative Commons Attribution 4.0 International (CC-BY 4.0) license, which reserves my rights to disseminate the work on my personal and corporate Web site with the appropriate attribution. Notwithstanding the above, I retain all proprietary rights other than copyright as assigned above, such as patent and trademark rights. For further details, see the Appendix below.

In the event that any elements used in the Material contain the work of third-party individuals, I understand that it is my responsibility to secure any necessary permissions and/or licenses and will provide it in writing to IW3C2. If the copyright holder requires a citation to a copyrighted work, I have obtained the correct wording and have included it in the designated space in the text.

I hereby release and discharge IW3C2 and other publication sponsors and organizers from any and all liability arising out of my inclusion in the publication, or in connection with the performance of any of the activities described in this document as permitted herein. This includes, but is not limited to, my right of privacy or publicity, copyright, patent rights, trade secret rights, moral rights, or trademark rights.

All permissions and releases granted by me herein shall be effective in perpetuity and throughout the universe, and extend and apply to the IW3C2 and its assigns, contractors, sub-licensed distributors, successors, and agents.

The following statement of copyright ownership will be displayed with the Material, unless otherwise specified: **" © [year] International World Wide Web Conference Committee (IW3C2), published under Creative Commons CC-BY 4.0 License." IW3C2 reserves the right to provide a hyperlink to the author's site if the Material is used in electronic media.**

◉ A. I hereby represent and warrant that I am the sole owner (or authorized agent of the copyright owner(s)) and assign publishing rights

○ B. I do not own some rights to this work: (check applicable)

Assign Rights

☑ I hereby assign rights and agree to publish under Creative Commons.

Third Party Material

☐ I have used third-party material and have permission to do so.

DATE: **02/07/2020** - hrezae2@uic.edu

Figure 51. Copyright - ResQueue

APPENDIX (Continued)

IW3C2 Copyright Release Form

Title of work:	**Superways: A Datacenter Topology for Incast-Heavy Workloads**
Author(s):	**Hamed Rezaei,Balajee Vamanan**
Description of material:	**WWW Paper**
Title of IW3C2 Publication:	**WWW '21: The Web Conference 2021**

I hereby assign (to the extend transferable, see point B below) to the International World Wide Web Conference Committee (IW3C2) the copyright of this Work for the full period of copyright and all renewals, extensions, revisions and revivals together with all accrued rights of action throughout the world in any form, including as part of IW3C2 and the public Conference Web site, on CD-ROM and in translation, or on videocassette, broadcast, cablecast, laserdisc, multimedia or any other media format now or hereafter known. (Not all forms of media will be utilized.) I accept that IW3C2 will allow the Association for Computing Machinery (ACM) to distribute, make available on the Internet or sell this Material as part of the above-named publication in the ACM Digital Library. Finally, I also accept that IW3C2 will publish this Work under the Creative Commons Attribution 4.0 International (CC-BY 4.0) license, which reserves my rights to disseminate the work on my personal and corporate Web site with the appropriate attribution. Notwithstanding the above, I retain all proprietary rights other than copyright as assigned above, such as patent and trademark rights. For further details, see the Appendix below.

In the event that any elements used in the Material contain the work of third-party individuals, I understand that it is my responsibility to secure any necessary permissions and/or licenses and will provide it in writing to IW3C2. If the copyright holder requires a citation to a copyrighted work, I have obtained the correct wording and have included it in the designated space in the text.

I hereby release and discharge IW3C2 and other publication sponsors and organizers from any and all liability arising out of my inclusion in the publication, or in connection with the performance of any of the activities described in this document as permitted herein. This includes, but is not limited to, my right of privacy or publicity, copyright, patent rights, trade secret rights, moral rights, or trademark rights.

All permissions and releases granted by me herein shall be effective in perpetuity and throughout the universe, and extend and apply to the IW3C2 and its assigns, contractors, sub-licensed distributors, successors, and agents.

The following statement of copyright ownership will be displayed with the Material, unless otherwise specified: **" © [year] International World Wide Web Conference Committee (IW3C2), published under Creative Commons CC-BY 4.0 License." IW3C2 reserves the right to provide a hyperlink to the author's site if the Material is used in electronic media.**

◉ A. I hereby represent and warrant that I am the sole owner (or authorized agent of the copyright owner(s)) and assign publishing rights

○ B. I do not own some rights to this work: (check applicable)

Audio/Video Release
* Your Audio/Video Release is conditional upon you agreeing to the terms set out below.

I further grant permission for ACM to record and/or transcribe and reproduce my presentation and likeness in the conference publication and as part of the ACM Digital Library and to distribute the same for sale in complete or partial form as part of an ACM product on CD-ROM, DVD, webcast, USB device, streaming video or any other media format now or hereafter known. I understand that my presentation will not be sold separately as a stand-alone product without my direct consent.

Accordingly, I further grant permission for ACM to include my name, likeness, presentation and comments and any biographical material submitted by me in connection with the conference and/or publication, whether used in excerpts or in full, for distribution described above and for any associated advertising or exhibition.

Figure 52. Copyright - Superways

APPENDIX (Continued)

IEEE COPYRIGHT AND CONSENT FORM

To ensure uniformity of treatment among all contributors, other forms may not be substituted for this form, nor may any wording of the form be changed. This form is intended for original material submitted to the IEEE and must accompany any such material in order to be published by the IEEE. Please read the form carefully and keep a copy for your files.

Smartbuf: An Agile Memory Management for Shared-Memory Switches in Datacenters
Mr. Hamed Rezaei, Mr. Hamidreza Almasi and Mr. Balajee Vamanan
2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)

COPYRIGHT TRANSFER
The undersigned hereby assigns to The Institute of Electrical and Electronics Engineers, Incorporated (the "IEEE") all rights under copyright that may exist in and to: (a) the Work, including any revised or expanded derivative works submitted to the IEEE by the undersigned based on the Work; and (b) any associated written or multimedia components or other enhancements accompanying the Work.

Figure 53. Copyright - Smartbuf

CITED LITERATURE

1. Barroso, L. A. and Hoelzle, U.: *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines* . Morgan and Claypool Publishers, 1st edition, 2009.

2. Barroso, L. A., Dean, J., and Hölzle, U.: Web search for a planet: The google cluster architecture. *IEEE Micro* , 23(2):22–28, March 2003.

3. Raiciu, C., Barre, S., Pluntke, C., Greenhalgh, A., Wischik, D., and Handley, M.: Improving datacenter performance and robustness with multipath tcp. In *Proceedings of the ACM SIGCOMM 2011 conference* , SIGCOMM '11, pages 266–277, New York, NY, USA, 2011. ACM.

4. Dean, J. and Barroso, L. A.: The tail at scale. *Commun. ACM* , 56(2), February 2013.

5. Zhang, Q., Liu, V., Zeng, H., and Krishnamurthy, A.: High-resolution measurement of data center microbursts. IMC'17. ACM, 2017.

6. Alizadeh, M. et al.: pfabric: Minimal near-optimal datacenter transport. SIGCOMM '13. ACM, 2013.

7. Vamanan, B., Hasan, J., and Vijaykumar, T.: Deadline-aware datacenter tcp (d2tcp). SIGCOMM '12, 2012.

8. Roy, A. et al.: Inside the social network's (datacenter) network. In *ACM SIGCOMM Computer Communication Review* . ACM, 2015.

9. Chen, Y. et al.: Understanding tcp incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM workshop on Research on enterprise networking* . ACM, 2009.

10. Alizadeh, M. et al.: Data center tcp (dctcp). SIGCOMM '10, 2010.

11. Wu, H., Feng, Z., Guo, C., and Zhang, Y.: Ictcp: Incast congestion control for tcp in data center networks. CoNEXT'10, 2010.

12. Handley, M. et al.: Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* , SIGCOMM '17. ACM, 2017.

13. Cho, I., Jang, K., and Han, D.: Credit-scheduled delay-bounded congestion control for datacenters. In *Proceedings of SIGCOMM* , pages 239–252, 2017.

14. Almasi, H., Rezaei, H., Chaudhry, M. U., and Vamanan, B.: Pulser: Fast congestion response using explicit incast notifications for datacenter networks. IEEE LANMAN'19, pages 1–6, 2019.

15. Chen, L., Chen, K., Bai, W., and Alizadeh, M.: Scheduling mix-flows in commodity datacenters with karuna. SIGCOMM '16, 2016.

16. Bai, W. et al.: Information-agnostic flow scheduling for commodity data centers. In *NSDI* , 2015.

17. Mittal, R., Agarwal, R., Ratnasamy, S., and Shenker, S.: Universal packet scheduling. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation* , NSDI'16, pages 501–521, 2016.

18. He, K. et al.: Presto: Edge-based load balancing for fast datacenter networks. SIGCOMM '15, 2015.

19. Alizadeh, M. et al.: Conga: Distributed congestion-aware load balancing for datacenters. SIGCOMM '14, 2014.

20. Katta, N. et al.: Hula: Scalable load balancing using programmable data planes. SOSR '16. ACM, 2016.

21. Zarifis, K. et al.: Dibs: Just-in-time congestion mitigation for data centers. In *Proceedings of the Ninth European Conference on Computer Systems* , EuroSys '14, 2014.

22. Arista 7050qx series 10/40g data center switches. `https://www.arista.com/assets/data/pdf/Datasheets/7050QX-32_32S_Datasheet.pdf`, 2020.

23. Bai, W., Chen, K., Hu, S., Tan, K., and Xiong, Y.: Congestion control for high-speed extremely shallow-buffered datacenter networks. APNet'17, 2017.

24. Leiserson, C. E.: Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.* , 34(10):892–901, October 1985.

25. Singh, A. et al.: Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Proceedings of SIGCOMM* , pages 183–197, 2015.

26. Kandula, S., Sengupta, S., Greenberg, A., Patel, P., and Chaiken, R.: The nature of data center traffic: Measurements & analysis. In *Proceedings of IMC* , pages 202–208, 2009.

27. Dean, J. and Ghemawat, S.: Mapreduce: simplified data processing on large clusters. *Communications of the ACM* , 51(1):107–113, 2008.

28. Wilson, C., Ballani, H., Karagiannis, T., and Rowtron, A.: Better never than late: meeting deadlines in datacenter networks. SIGCOMM '11, New York, NY, USA, 2011. ACM.

29. Munir, A., Baig, G., Irteza, S. M., Qazi, I. A., Liu, A. X., and Dogar, F. R.: Friends, not foes: Synthesizing existing transport strategies for data center networks. In *Proceedings of the 2014 ACM Conference on SIGCOMM* , SIGCOMM '14, pages 491–502, New York, NY, USA, 2014. ACM.

30. Hong, C.-Y., Caesar, M., and Godfrey, P. B.: Finishing flows quickly with preemptive scheduling. ACM SIGCOMM'12, 2012.

31. Kabbani, A., Vamanan, B., Hasan, J., and Duchene, F.: Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies* , CoNEXT '14, pages 149–160, New York, NY, USA, 2014. ACM.

32. Montazeri, B. et al.: Homa: A receiver-driven low-latency transport protocol using network priorities. SIGCOMM '18, pages 221–235. ACM, 2018.

33. Varga, A.: Omnet++. In *Modeling and tools for network simulation* , pages 35–59. Springer, 2010.

34. Liu, V. et al.: Subways: A case for redundant, inexpensive data center edge links. CoNEXT'15. ACM, 2015.

35. Choudhury, A. K. and Hahne, E. L.: Dynamic queue length thresholds for shared-memory packet switches. *IEEE/ACM Transactions On Networking* , 6(2):130–140, 1998.

36. Shan, D., Jiang, W., and Ren, F.: Absorbing micro-burst traffic by enhancing dynamic threshold policy of data center switches. In *2015 IEEE Conference on Computer Communications (INFOCOM)* , pages 118–126. IEEE, 2015.

37. Postel, J.: Transmission control protocol. 1981.

38. Ramakrishnan, K., Floyd, S., and Black, D.: The addition of explicit congestion notification (ecn) to ip. Technical report, 2001.

39. Benson, T., Akella, A., and Maltz, D. A.: Network traffic characteristics of data centers in the wild. In *Proceedings of IMC* , 2010.

40. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J.: Borg, omega, and kubernetes. *Queue* , 14(1):70–93, 2016.

41. Kumar, J. and Singh, A. K.: Cloud datacenter workload estimation using error preventive time series forecasting models. *Cluster Computing* , 23(2):1363–1379, 2020.

42. Greenberg, A. et al.: Vl2: a scalable and flexible data center network. Number 4 in SIGCOMM '09. ACM, 2009.

43. `https://www.techrepublic.com/article/facebook-fabric-an-innovative-network-topology-`
fb.

44. Association, T. I.: Tia standard ansi/tia-942-a, data center cabling standard amended. 2012.

45. `https://www.cablinginstall.com/data-center/article/16468527/the-data-center-evolution-how-to-overcome-its-cabling-challenges`, Cablinginstall.

46. `https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html`, Intel Tofino.

47. Riley, G. F. and Henderson, T. R.: The ns-3 network simulator. In *Modeling and tools for network simulation* , pages 15–34. Springer, 2010.

48. Cho, I., Jang, K., and Han, D.: Credit-scheduled delay-bounded congestion control for datacenters. SIGCOMM '17. ACM.

49. Vasudevan, V., Phanishayee, A., Shah, H., Krevat, E., Andersen, D. G., Ganger, G. R., Gibson, G. A., and Mueller, B.: Safe and effective fine-grained tcp retransmissions for datacenter communication. In *ACM SIGCOMM* . ACM, 2009.

50. Intel ethernet controller x710.

51. Guo, C. et al.: Bcube: a high performance, server-centric network architecture for modular data centers. *ACM SIGCOMM Computer Communication Review* , 39(4):63–74, 2009.

52. Singla, A., Hong, C.-Y., Popa, L., and Godfrey, P. B.: Jellyfish: Networking data centers randomly. NSDI'12, 2012.

53. `https://github.com/hrezae2/Superways-WWW-21/`, Superways_ns3_code.

54. Ricci, R. and Eide, E.: The cloudlab team. *Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. USENIX* , 39(6), 2014.

55. Popa, L., Ratnasamy, S., Iannaccone, G., Krishnamurthy, A., and Stoica, I.: A cost comparison of datacenter network architectures. In *Proceedings of the 6th International Conference* , page 16. ACM, 2010.

56. `http://www.fs.com`, FS.com.

57. `http://www.dell.com`, Dell.

58. `http://www.Amazon.com`, Amazon.

59. `https://www.compsource.com`, Compsource.

60. `https://www.cablewholesale.com`, Cablewholesale.

61. `https://www.juniper.net`, Juniper Networks.

62. `https://www.cisco.com`, Cisco Systems.

63. `https://www.brocade.com`, Brocade Inc.

64. `https://www.huawei.com`, Huawei Technologies.

65. https://www.intel.com, intel.

66. https://www.hp.com, HP.

67. https://www.rosewill.com, Rosewill.

68. https://www.arista.com, Arista Networks.

69. Apache solr. https://lucene.apache.org/solr/, 2020.

70. Apostolaki, M., Vanbever, L., and Ghobadi, M.: Fab: Toward flow-aware buffer sharing on programmable switches. In *Proceedings of the 2019 Workshop on Buffer Sizing* , BS '19, 2019.

71. Appenzeller, G., Keslassy, I., and McKeown, N.: Sizing router buffers. *SIGCOMM Comput. Commun. Rev.* , 34(4):281–292, August 2004.

VITA

NAME: HAMED REZAEI

EDUCATION: Ph.D., Computer Science, University of Illinois at Chicago, Chicago, Illinois, 2021.

M.Sci., Information Technology, University of Qom, Qom, Iran, 2014.

B.Eng., Information Technology, Payame-Noor Univeristy, Kermanshah, Iran, 2012.

ACADEMIC EXPERIENCE: Research Assistant, BITS Lab, Department of Computer Science, University of Illinois at Chicago, 2015 - 2021.

Teaching Assistant, Department of Computer Science, University of Illinois at Chicago:

- Operating Systems **Fall 2015**
- Program Design - C/C++ **Fall 2018 and Spring 2020**
- Introduction to Computer Networks **Spring 2019**

RESEARCH INTERESTS: Datacenter/Cloud networks, intersection of ML and networking, traffic engineering, programmable DCs, congestion control, NFV/VNF, load balancing, Software defined networking.