Protean: Adaptive Management of Shared-Memory in Datacenter Switches

Hamidreza Almasi
Department of Computer Science
University of Illinois at Chicago
USA
halmas3@uic.edu

Rohan Vardekar
Department of Computer Science
University of Illinois at Chicago
USA
rvarde2@uic.edu

Balajee Vamanan
Department of Computer Science
University of Illinois at Chicago
USA
bvamanan@uic.edu

***Abstract*—Datacenters rely on high-bandwidth networks that use inexpensive, shared-buffer switches. The combination of high bandwidth, bursty traffic patterns, and shallow buffers imply that switch buffer is a heavily contended resource and intelligent management of shared buffers among competing traffic (ports, traffic classes) becomes an important challenge. Dynamic Threshold (DT), which is the current state-of-the-art in buffer management, provides either high bandwidth utilization with poor burst absorption or good burst absorption with inferior utilization, but not both. We present *Protean*, which dynamically identifies *bursty* traffic and allocates more buffer space accordingly—*Protean* provides more space to queues that experience transient load spikes by observing the gradient of queue length but does not cause persistent unfairness as the gradient cannot continue to remain high in shallow buffered switches for long periods of time. We implemented *Protean* in today's programmable switches and demonstrate their high performance with negligible overhead. Our at-scale ns-3 simulations show that *Protean* reduces the tail latency by a factor of *5* over DT on average across varying loads with realistic workloads.**

***Index Terms*—computer networks, datacenters, buffer management, burst absorption**

I. Introduction

Datacenters perform the core computation in most of the Internet-scale applications that we have come to rely on a day-to-day basis such as social media, web search, video streaming, and gaming. Datacenter operators require both *high utilization* and *low, predictable latency* from the network while relying on low-cost *commodity* hardware [1]. As network bandwidth keeps growing at a rapid pace while network delays cannot reduce substantially due to hard physical limits, the amount of on-chip memory (i.e., SRAM) needed for packet buffers also grows proportionally [2]. However, designing large on-chip memory that also operates at high speed is an engineering challenge and is disproportionately expensive, modern datacenter routers (switches) operate with *shallow* buffers that are shared among ports, as opposed to large buffers that are dedicated per port [3]. Therefore, how to share the buffer space among competing ports has become one of the important research challenges for the networking community.

The current state-of-the-art switches use *Dynamic Threshold* (DT) for determining whether to allow a packet or drop it based on the current per-port queue size and spare capacity [4]. More precisely, DT allows a port (queue) [1] to grow proportional to spare capacity— as spare capacity reduces, each port can only have a proportionally smaller queue and thus favors ports with small queues while penalizing ports with long queues. DT has a parameter that allows operators to tune its performance to achieve a trade-off between utilization and burst absorption but our experiments show that it is hard to achieve a good trade-off. At one extreme, DT behaves very close to *greedy* complete sharing with good utilization but some ports can potentially monopolize the buffer space. In practice, we observe that long-lasting flows tend to occupy most of the buffer space and hurt high-performant short flows (e.g., incast flows) that arrive sporadically. On the other extreme, DT can be configured to reserve a good fraction of the buffer unoccupied to be used for bursts but such reservation hurts network utilization because it exacerbates the shallow buffer problem further. Note that the primary function of packet buffers is to act as *shock absorbers*—avoid overflows during periods of link over-subscription and avoid underflows during periods of under-subscription, thus smoothing out the spikes to provide close to maximum throughput. Because shallow buffered switches are already at a disadvantage when it comes to the function of smoothing out spikes to provide high throughput (utilization), reserving more space is not an attractive option. Indeed, today's operators set this design parameter to optimize utilization (i.e., first extreme) but suffer from its pitfalls.

We propose *Protean*, which resolves DT's shortcomings. We make the *key observation* that DT does not discriminate between ports (queues) based on traffic dynamics. As a result, DT is forced to make a *static* choice between utilization and burst absorption, *globally* for *all* ports. However, in reality, we observe that some ports (or queues) experience transient bursty traffic (e.g., incast) while others experience smoother traffic (e.g., long-lasting flows). Therefore, if we can selectively enable some ports to absorb such transient bursts while allowing non-bursty traffic to use as much buffer capacity as possible at other times, we can achieve both high burst tolerance and high utilization.

[1]It may appear as if we assume only one queue per port, but our idea generalizes to multiple (priority) queues per port. Our design and evaluation use multiple queues per port. We simplify our exposition for ease of readability.

A straightforward way to identify ports that experience bursty traffic is to observe their queue lengths. However, queue length is a lagging indicator. Therefore, we rely on the first derivative of queue length, which is a leading indicator. Our idea is to enhance DT such that we allow a queue (port) to grow proportional to the product of spare capacity and the gradient of its queue length. Thus, ports that experience bursty traffic can grow their queues at the expense of other queues during periods of over-subscription (i.e., during an incast). In practice, *Protean* allocates more buffer space selectively to some ports during periods of transient load spikes when links get over-subscribed. By relying on the gradient, which is unlikely to stay high for long periods to time due to shallow buffers, *Protean* ensures that long-term fairness is not negatively impacted. In other words, we do not let some ports monopolize buffer space at the expense of other ports over long periods of time. Our experiments clearly demonstrate that *Protean* is able to achieve low tail latencies (less latency variation) without worsening network utilization (goodput).

An attractive feature of *Protean* is that it lends itself to easier hardware implementations. As proof of concept, we implemented *Protean* on Intel's *Tofino* switches in P4 [5]–[7]. We implement *Protean* as a packet admission logic that compares queue length to a threshold. We periodically generate probe packets that convey queue lengths from the egress pipeline to the ingress pipeline.

We summarize our contributions as follows:

- Identify the key problem in DT: because DT does not discriminate between ports (queues) based on traffic dynamics, it is forced either to keep a large headroom for *all* ports or to greedily allocate space based on demand on a first-come, first-served basis
- Propose *Protean*, which quickly identifies ports that experience transient load spikes and allocates space to absorb those spikes without causing long-term unfairness and starvation
- Present a feasible design and show proof-of-concept implementation in today's programmable switches
- Demonstrate the benefits of our proposal using exhaustive evaluations: analyze the real implementation as well as study the performance/overhead in realistic scenarios *at scale* using ns-3 simulations.

To highlight key results, *Protean* improves tail latency (i.e., 99.9^{th} percentile flow completion time) by a factor of 5x on average across loads (up to a factor of 6x at high loads) over the current state-of-the-art (DT) while achieving similar or better network utilization (goodput). Further, our implementation on the Tofino switch demonstrates the feasibility of *Protean*'s deployment and incurs negligible overheads— *Protean* uses ten stages with an average of 3% SRAM and 3.5% TCAM usage.

The paper is organized as follows: We start with a brief background on *Dynamic Threshold* (DT) and present opportunity studies to demonstrate its pitfalls in Section II. Then, in Section III, we unveil our key ideas and dive into design and implementation details. Section IV discusses our evaluation methodology and our key findings. Section V enumerates related work in this area and Section VI concludes our paper with closing remarks and future work.

II. Background and motivation

High fan-in traffic patterns are prevalent in datacenters and impose challenges on network protocol and architecture design. Incast is an example of such a scenario where a burst of (usually short) flows synchronously arrives at a switch port connected to the receiver. Large incasts cause congestion and buffer overflow that lead to packet drops that are only detected by timeouts [8], [9]. Achieving low-latency goal while maintaining high throughput is tightly coupled to avoiding persistent packet drops. There are two main schemes that provide control over packet drops and help achieve this goal in datacenters. (1) End-to-end congestion control protocols like DCTCP [3] and TIMELY [10], are able to infer the extent of congestion based on in-network signals such as ECN and RTT variation. End-hosts will then apply rate control mechanisms such as adjusting the size of sending window to ease the congestion. (2) Dynamic shared-memory management schemes can provide a cushion during transient periods by reserving a fraction of the buffer. This reduces packet drops when a port suddenly becomes overloaded because its share can grow temporarily before reaching a steady-state [4].

Although congestion control methods try to maintain short queue lengths at switches while keeping the throughput high, they have no control over the relative allocation of the shared-memory pool between incast and non-incast ports. Also, these methods are effective when congestion episodes last a few RTTs but they cannot help much if there are many short flows that last less than an RTT, which is the *common case* in today's datacenters [11]. Buffer provisioning schemes, though, can have more insight into how the shared-memory is allocated among contending ports and how bursty the arriving traffic at each port is. Therefore, they can respond to congestion events earlier than the delayed end-host reactions and are preferred for providing burst absorption.

We now explain the state-of-the-art approaches for managing the shared-memory in datacenters. In the most intuitive approach, called *complete partitioning (CP)* [12], the total buffer is statically divided among the ports. Therefore each port is allowed to use up to a fixed amount of the shared buffer space and the sum of these amounts is equal to the total memory. CP is fair but it provides no sharing mechanism. It wastes a lot of the shallow buffers prevalent in datacenters which is a scarce and expensive resource. Also, large bursts create load imbalance at ports and since they can occur at any port, they would need more than the static per-port cap determined by CP to be absorbed. Therefore, CP does not provide burst absorption.

The other intuitive extreme is *complete sharing (CS)* where the entire buffer is shared among ports and a packet is admitted if there is any space left for it in the memory. CS is good for burst absorption if the buffer can accommodate the burst size but it has no mechanism to prevent a highly utilized port from monopolizing most of the buffer. In a scenario where multiple ports are busy transferring long flow bytes, the remaining buffer space may become smaller as TCP increases the sending window size. This could easily reach a point where an incoming burst cannot fit in the buffer and experiences packet drops. As such, long-lasting TCP flows, by construction, will fill up all the available buffer space and starve other flows.

Dynamic Threshold (DT) is the current state-of-the-art and is deployed by several switch vendors. When a packet destined for an output port queue arrives, the queue length is compared with a threshold and if it's larger, the packet is dropped. At time t, the control threshold $T(t)$ in DT is calculated as:

$$T(t) = \alpha \cdot (B - \sum_{i} Q^{i}(t)) \tag{1}$$

where α is a static parameter, B is the total buffer size, and $Q^{i}(t)$ is the length of the output queue i at time t. If the port loads change, DT will go through a transient state but eventually, all queues will reach their steady-state allocation which is less than or equal to a new control threshold and it robustly adapts to the change. If Q is the total amount of buffer occupied in the steady-state, $Q = S \cdot T + \Omega$ where S is the number of very active queues, T is the threshold in the steady-state and Ω is the amount of buffer occupied by those queues that are less than the threshold in length, also known as uncontrolled queues. The steady-state allocation of controlled queues in DT is:

$$Q^{i} = T = \frac{\alpha \cdot (B - \Omega)}{1 + \alpha \cdot S} \tag{2}$$

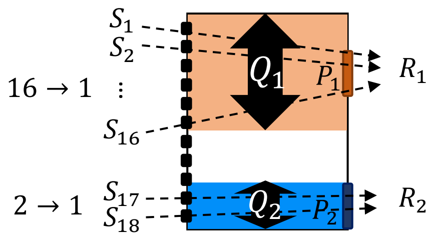
If there are different α values for queues with different priorities, the allocation will be proportional to their α values. In this case if α^{i} is the alpha associated to queue i, equation 2 will generalize to:

$$Q^{i} = \frac{\alpha^{i} \cdot (B - \Omega)}{1 + \sum_{j \in C} \alpha^{j}} \tag{3}$$

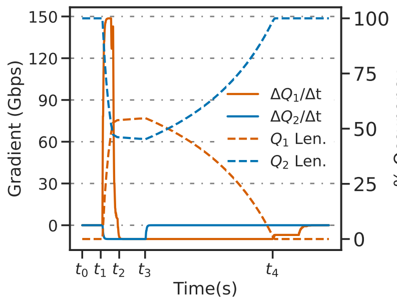
where C is the set of controlled queues. As more ports become congested, each port is allocated a smaller share. DT is simple to implement and variants of it are used in today's switches [13]–[15] but it is not efficient in datacenters because the threshold only depends on the statically configured α parameter and the remaining buffer size. Indeed, although bursty traffic is frequent in datacenters and the tail latency of incast flows is important, the threshold calculated by DT is not sensitive to how bursty the arriving traffic to a queue is and therefore is unable to dynamically set the threshold to absorb it. Moreover, fixed alpha values cannot differentiate between buffer demands of bursts with different severities. This becomes critical in datacenters because the load is unbalanced and simultaneous bursts use the buffer non-linearly. There are workloads that utilize *all* ports of the switch more than 50% [11]. In these settings, it is important to set the thresholds proportional to the burstiness observed at ports. This is what *Protean* does when it detects some ports would need a larger share of buffer since they drastically build up the queues in a short time. *Protean* first monitors how fast the queues build up on arrival of bursts. It then uses this measure as the burst signature to proportionally set a higher threshold for larger ones when they need more buffer space to be absorbed. For large bursts that were previously admitted but are not anymore building up their queue, *Protean* relies on DT's threshold to provision more buffer for other bursts on their onset.

To illustrate the problems with DT, we simulate a simple experiment where several end-hosts are connected to a single shared-memory switch with 3 MB of total buffer size. In Fig. 1(a), 16 senders $S_1..S_{16}$ are going to create a burst of 100 KB flows to receiver R_1 connected to port P_1 with $\alpha_{incast} = 1$ and two other senders S_{17}, S_{18} are sending long ongoing background flows to receiver R_2 connected to port P_2 with $\alpha_{non-incast} = 0.5$. Fig. 1(b) and Fig. 1(c) respectively show how DT and *Protean* allocate the buffer to ports P_1 and P_2. In these figures, the left y-axis shows the gradient of the queue length (with respect to time) for each queue and the right y-axis shows what percentage of the occupied buffer is used by that queue. During the period $t_0 - t_1$ the only existing traffic is the background flows and Q_2 is in its steady-state where it is allocated 1 MB. At the time t_1 the burst arrives at the buffer and Q_1 starts to build up. At this time 2MB of the buffer is unoccupied and the burst (~1.6 MB) could fit, but because DT has a static alpha and is not sensitive to the burst severity, it cannot increase the threshold for Q_1 just high enough to avoid costly packet drops from the burst. Note that the arrival of the burst would introduce a highly utilized port, P_1, and would naturally drop the threshold for the Q_2 at P_2. However, *Protean* admits more burst packets due to setting a threshold proportional to its gradient, and thus the threshold for Q_2 drops more than that of DT. This is because admitting more packets from the burst by *Protean*, would quickly decrease the remaining buffer space and since alpha is fixed for background flows, their threshold would drop even more. In other words, after the burst arrives, *Protean* would drop packets only from the background flows while DT would drop from both the burst and background flows. This difference is reflected at t_2 where the burst is fully absorbed by *Protean* with Q_1 having a higher relative occupancy while it experiences many packet drops with DT leading to a smaller occupancy for Q_1 and a larger one for Q_2. During the period $t_2 - t_3$ both queues are draining their admitted bytes and at time t_3 the background flows recover from their packet losses. During the period $t_3 - t_4$, Q_1 continues to drain more while Q_2 returns back to its steady-state allocation. For $t > t_4$, there are no more Q_1 bytes for *Protean* but for DT, the Q_1 packets that are supposed to be retransmitted are triggered by the timeout at some point.

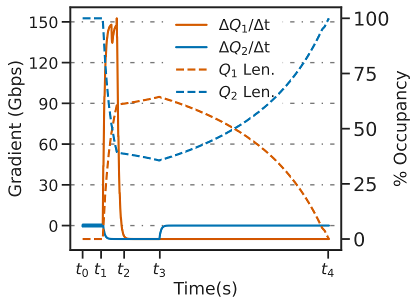
In summary, because background flows do not need a lot of buffer to reach a high throughput, it makes sense to increase the threshold for a short incast which needs a lot of the



(a) Scenario



(b) DT



(c) Protean

Fig. 1: Burst absorption

buffer to be absorbed and hence decrease the threshold for the background flows to free up some space during the time the incast lives. The amount of increase in the threshold for the queue that experiences an incast should be proportional to the severity of the incast. In the next section we describe how we identify the severity of an incast.

III. Proposed work

After explaining DT's inability to dynamically absorb bursts, in this section we first describe high-level insights we used in designing *Protean*. Then we describe *Protean*'s ingress and egress algorithms in detail. Finally, we describe how we realized *Protean* in hardware using programmable switches.

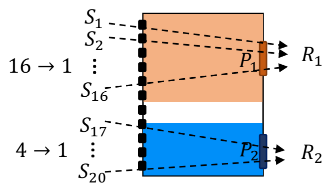
A. High-level Idea

In datacenters, ports that are connected to incast aggregators are more likely to receive large bursts and should be allocated more buffer space. However, predicting the time a burst arrives is not trivial. Our high-level idea is to detect a burst by measuring the gradient of the burst's queue length and allocate the buffer space proportional to that value.

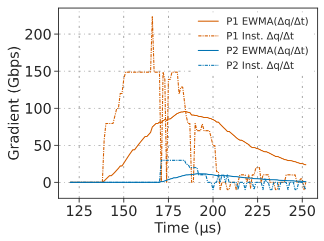
During an incast episode, multiple senders are sending to one receiver. The data from these senders arrive at different input ports of a top-of-rack switch and get forwarded to the same output port. This drastically increases the output port's queue length in a short period of time because the input rate is amplified but the output rate remains fixed. We know that most of the flows in an incast are short flows [11], [16], and they usually have less than $RTT \times BW$ bytes. Therefore, factors like flow size or congestion control protocol are not major predictors of the incast's buffer demand. However, there exists a clear correlation between fan-in (incast degree) and the amount of buffer an incast needs to be absorbed. This observation enables us to allocate buffer space to incasts with respect to their fan-in. Still, fan-in is not known by the network switches because the application in the upper layer divides the query between servers and changing the protocol stack to carry this information in the network layer is difficult.

We make the key insight that the gradient of the queue length can be effectively used as a representative for fan-in. This is because of two reasons: (1) It is proportional to the aggregate input rate, i.e., $average\ sending\ rate \times fan\text{-}in$. (2) It is switch-local information. Therefore, instead of using fan-in, we use the gradient of queue length to set the threshold for large bursts. However, if we directly use this metric, our approach would be susceptible to noisy switch measurements, so we propose to smooth the gradient using exponentially weighted moving average (EWMA) with a weight β to the current gradient sample and $1-\beta$ to the previously calculated average. We use this metric as a coefficient for calculating the threshold for bursts.

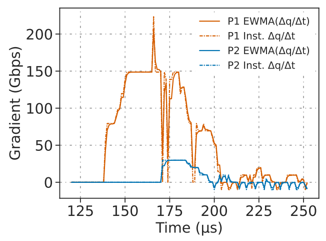
The value of β serves as a knob to tune the smoothness of the gradient. It should be large enough to ensure that the presence of back-to-back packets is captured but also small enough to filter outliers and improve accuracy. To better understand the effect of β, we simulated an experiment with two bursts of different fan-in values sending data to two output ports. The scenario for this experiment is shown in Fig. 2(a). Here, 16 senders $S_1..S_{16}$ are sending to receiver R_1 connected to port P_1 and 4 other senders $S_{17}..S_{20}$ are sending to receiver R_2 connected to port P_2. Each flow is 100 KB, the buffer size is 2 MB, and we use *Protean* for buffer allocation. We measure instantaneous and smoothed gradients over time at port queues with different values of β. Setting β to a small value like in Fig. 2(b), filters outliers and smoothes the gradient but any decision for buffer allocation based on that would be too late. This is because by setting a small β the averaging time window would be too large and bursts are usually short-lived, so *Protean* would miss the burst and a decision for its threshold would lag behind to warrant its absorption. On the other hand, setting β to a large value like in Fig. 2(c) would make the smoothed gradient closely track the instantaneous one and ensure that burst packets would not be missed, but because measurements are noisy, it would be inaccurate. Here around the time $t = 175\mu s$ a similar threshold would be calculated for both bursts during their onset whereas clearly, one burst is larger and more packets should be admitted from it. Finally, a proper value for β like in Fig. 2(d) provides smoothness, accuracy, and agility at the same time and can be considered a reliable signature for burst that can be used in the threshold calculation. Later in Section IV we discuss the robustness of *Protean* to this design parameter.



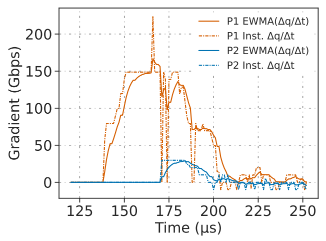
(a) Scenario



(b) $\beta = \frac{1}{32}$



(c) $\beta = \frac{3}{4}$



(d) $\beta = \frac{1}{4}$

Fig. 2: Smoothness vs. Accuracy

Protean classifies flows into two main categories, incast flows and non-incast flows. It considers higher (proportional to the gradient) thresholds for the queues that incast flows use during their life. Non-incast flows are further classified into long and short flows. For long flows, *Protean* falls back to DT and for short flows, because tail latency is also important to them, it falls back to CS as a best-effort allocation attempt.

B. *Protean*

After explaining the insights that led us to *Protean*'s design, we now describe *Protean*'s functionality at switch ingress and egress pipelines. An abstract model for a switch pipe is shown in Fig. 3. *Protean* decides whether a packet should be admitted based on thresholds calculated for each queue. Admission checks are performed in the ingress pipeline. If *Protean* decides to admit a packet, it delivers it to the traffic manager to place it in the right queue, otherwise, it marks the packet for drop. *Protean* calculates the thresholds for each queue at the egress pipeline and feeds this information as input to the ingress.

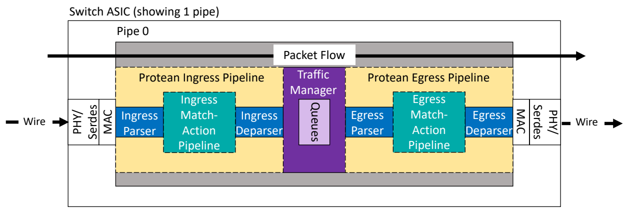


Fig. 3: *Protean*'s abstract switch model

Algorithm 1 shows the details of *Protean* at the ingress. When a packet arrives, we assume that the output port and the output queue within that port for the packet is known based on routing table entries and packet priority. We periodically synchronize the size of the output queue, remaining buffer size, and incast queue threshold from the egress logic with the ingress. Therefore we assume this information is available at the ingress. At line 1 we extract the packet's priority. Line 2, specifies that *Protean*'s admission block at the ingress assumes a packet is inadmissible unless determined otherwise. Lines 3-4 of Algorithm 1 check if the packet belongs to a short flow. In that case, if the remaining buffer space is enough, the packet is admitted. This is the policy CS would take for all packets. Lines 5-6 check if the packet belongs to a long flow. In that case, *Protean* would behave similarly to DT, i.e., if the current output queue length is smaller than DT's threshold (alpha $\times$ remaining buffer size) it would admit the packet. Otherwise, the packet is using an incast queue in port P for which, at lines 7-8, *Protean* checks if the queue length is smaller than the threshold calculated by Algorithm 2 then it admits the packet.

Algorithm 1: *Protean* Ingress

```
Input: Packet: pkt, OutputPort: P,
       QueueSize: Qlen, RemainingBuffer: Rem,
       ShortPriority: $p_s$, LongPriority: $p_l$,
       LongAlpha: $\alpha_l$, IncastThresh: $thr_i[P]$
Output: bool Admit
1 $prio \leftarrow pkt.priority$
2 $Admit \leftarrow False$
3 if $prio = p_s$ and $pkt.size < Rem$ then
4     $Admit \leftarrow True$
5 else if $prio = p_l$ and $Qlen < \alpha_l \times Rem$ then
6     $Admit \leftarrow True$
7 else if $Qlen < thr_i[P]$ then
8     $Admit \leftarrow True$
```

Protean's egress logic is shown in Algorithm 2. Here the total buffer size and total occupied buffer space (derived from probe packets coming out of the traffic manager) are inputs to the algorithm. We consider an input parameter, *buildup_threshold*, that serves as a measure to detect large bursts since not all bursts are incasts. If *buildup_threshold* is positive, it means the queue is building up, i.e., the enqueue rate is greater than the dequeue rate. However, for incasts, there is a sharp increase in queue length. Therefore we set the *buildup_threshold* to a multiple of linerate. The outputs of Algorithm 2 are the threshold set for the queue at port P, as well as the remaining buffer space. To calculate the threshold, whenever a packet is dequeued from an output port, we perform a series of operations. First, we calculate the remaining buffer space at line 2. Then we capture the dequeue time, and the length of the queue at lines 3-5. At lines 6-9 we calculate the instantaneous and smoothed gradients. Note that before the switch starts operating, $Gradient_{prev}$ and T_{prev} are initialized to zero for all queues. Finally, at lines 10-13, if the

Algorithm 2: *Protean* Egress

```
Input: BufferSize: Total, OccupiedBuffer: Occ,
       BuildupThresh: buildup_threshold, Weight: $\beta$,
       IncastAlpha: $\alpha_i$
Output: RemainingBuffer: Rem,
        IncastThresh: $thr_i[P]$ for output port P
1  for each pkt dequeue at port P do
2      $Rem = Total - Occ$
3      $T \leftarrow pkt.timestamp$
4      $prio \leftarrow pkt.priority$
5      $Qlen \leftarrow P[prio].len$
6      $Gradient = (Qlen - Qlen_{prev})/(T - T_{prev})$
7      $Qlen_{prev} \leftarrow Qlen$, $T_{prev} \leftarrow T$
8      $Gradient = \beta \times Gradient + (1-\beta) \times Gradient_{prev}$
9      $Gradient_{prev} \leftarrow Gradient$
10     if $Gradient > buildup\_threshold$ then
11         $thr_i[P] = Gradient \times \alpha_i \times Rem$
12     else
13         $thr_i[P] = \alpha_i \times Rem$
```

smoothed gradient is greater than the *buildup_threshold*, we use it as a coefficient in DT's threshold calculation formula, otherwise we rely on DT's original threshold. We set $\beta = \frac{1}{4}$ and $buildup\_threshold = 2 \cdot Linerate$. Later in Section IV we discuss the robustness of *Protean* to these design parameters.

C. Implementation on Programmable Switches

Considering the mathematical complexity of the algorithm for programmable switches, *Protean*'s implementation on Tofino Native Architecture (TNA) is challenging due to hardware limitations. For our experiments we used a Tofino Wedge 100BF-32X switch [17]. Since the traffic manager is not programmable, workarounds impose approximations in the implementation. Queue Id and forwarding port information are pushed in form of table entries. Challenges faced in the implementation are as follows:

- **Acquiring the queue length and timestamp in ingress pipeline**: Whether to accept a packet or to drop it, is decided in the ingress pipeline by comparing the queue length to a determined threshold. Queue length is not accessible in the ingress pipeline, but it is available in form of metadata in the egress pipeline. Tofino has a packet generator that we trigger using a timer to create and recirculate customized probe packets which synchronize this information for each queue every RTT. From the egress intrinsic metadata, details such as queue id, queue length and time stamp are filled in the header fields of these custom packets. After recirculation, this information is stored in the registers of the ingress pipeline.
- **Calculating the total buffer occupied in Tofino**: In order to calculate the total buffer occupied, the lengths of different queues (i.e., their contribution to the buffer) should be aggregated. Tofino does not allow accessing multiple indices of a register array at once. Our solution assigns a register for the remaining buffer and uses the probe packets generated for queues in every RTT to subtract their corresponding queue length from the remaining buffer. We also assign another register that counts the number of probe packets accounted for the current RTT and once all queue length contributions towards the buffer are considered, the remaining buffer and the counter are reset in the next RTT.
- **Floating point operations with Tofino**: Multiplication and division are not supported by Tofino. We use logarithm lookup tables to reduce multiplication and division to addition and subtraction and instead of the operands, we use their logarithm and exponent values [18]. Therefore, line 6 of the Algorithm 2 is realized using:

$$Gradient = 2^{\log_2(Qlen - Qlen_{prev}) - \log_2(T - T_{prev})} \tag{4}$$

  Also, since we pick α and β values as powers of two, their multiplication or division into another operand is realized using shift operation which is supported. As an example, line 11 in Algorithm 2 is realized using:

$$thr_i[P] = 2^{\log_2(Gradient) + \log_2(Rem \gg x)} \tag{5}$$

  where x is a constant value we shift by and is chosen depending on the specific power of two we are multiplying. Therefore, we approximate multiplication/division with a mean error of less than 1%.
- **Creating buffer pressure in a small physical deployment**: Our switch has 20 MB of total buffer space designed to be shared among 32 $\times$ 100 Gbps links. Filling the buffer enough to create a buffer pressure for evaluation with a few servers connected to it is challenging. We use the runtime API to shape the output ports inside of the traffic manager in order to fill up the buffer.

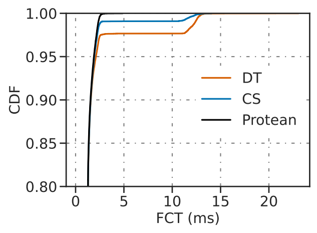
Later in Section IV-C, we describe our physical testbed details and evaluation.

IV. Evaluation

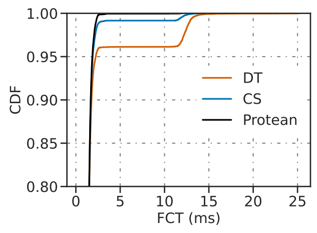
A. Methodology

We evaluate *Protean*'s performance with large-scale simulations as well as a real implementation scenario.

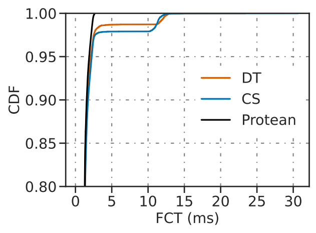
For large-scale experiments, we use ns-3 [19] and simulate a leaf-spine topology with 4 spine switches, 4 leaf switches, and 16 servers connected to each leaf switch. The oversubscription factor is 4 and all links are 10 Gbps with a $10\mu s$ unloaded delay. We set the TCP RTO_{min} to 10ms and set DCTCP ECN threshold to 65 following the guidelines in [3]. Switches have 2 MB of shared buffer space according to a Trident II model [15], [20] proportional to number of used ports. As in Section II, we set $\alpha_{incast} = 1$ and $\alpha_{non-incast} = 0.5$ for



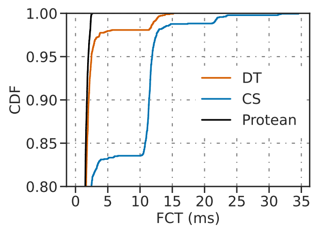
(a) DCTCP, Load=40%



(b) DCTCP, Load=80%

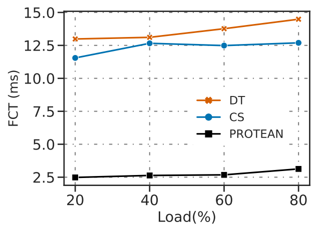


(c) TCP, Load=40%

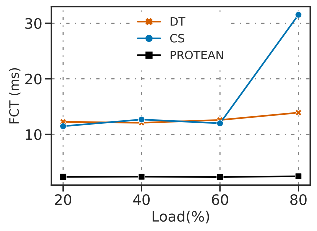


(d) TCP, Load=80%

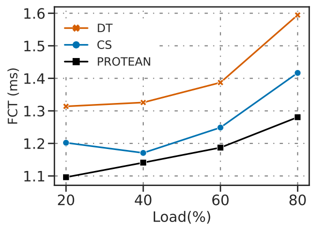
Fig. 4: CDF of incast flow completion times



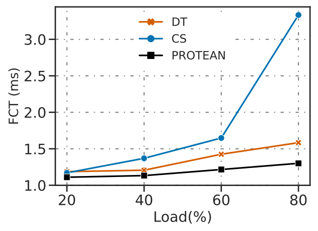
(a) 99.9th %-ile with DCTCP



(b) 99.9th %-ile with TCP



(c) Average with DCTCP



(d) Average with TCP

Fig. 5: Incast flow completion times

both DT and *Protean* according to [4]. We set $\beta = \frac{1}{4}$ and $buildup\_treshold = 2 \cdot Linerate$ following the insights in Section III-A. We generate background flows using the flow size distribution in [3]. For a given load, background flow arrivals follow a Poisson distribution and the source and the destination for each flow is chosen uniformly randomly. We generate the incast traffic following the model in [21] where all 16 servers under a randomly chosen leaf switch send $\frac{1}{16}$ of a given incast size to an incast aggregator under another leaf switch. Incast flows arrival times are according to a Poisson process with an average rate of 1 incast per second per each incast aggregator. Incast size is set to 75% of the buffer size in all experiments unless otherwise specified.

We evaluate *Protean* for various performance metrics, both when an active queue management scheme (e.g., RED [22]) is in place, and when it is not. We assume three queues per-port and use round-robin scheduling for dequeueing. We compare *Protean* with DT and CS when paired with TCP Cubic and DCTCP as congestion control scheme. Here when we mention short flows, we mean non-incast flows with a size smaller than $Linerate \times RTT$.

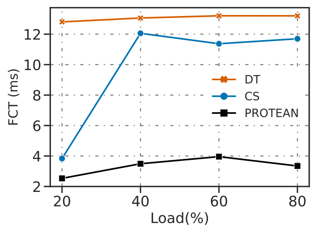
B. Simulation Results

We summarize our at scale evaluation of *Protean* as follows:

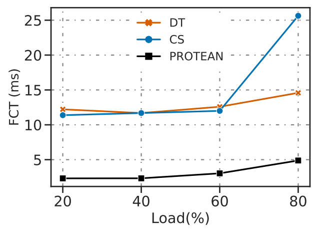
- **Flow Completion Time (FCT):** We compare average and tail (99.9^{th} percentile) flow completion times of *Protean* with DT and CS for both incast and short flows:
  - **Tail Incast FCT:** With TCP, *Protean* improves tail incast FCT by a factor of 5x (on average across loads) compared to DT and 6.9x compared to CS. When paired with DCTCP, it improves tail incast FCT by a factor of 5x on average compared to DT and 4.6x compared to CS.
  - **Average Incast FCT:** With TCP, *Protean* improves average incast FCT up to 1.26x over DT and 2.61x over CS at high loads. With DCTCP, it improves average incast FCT up to 1.25x over DT and 1.11x over CS.
  - **Tail Short FCT:** *Protean* also improves tail FCT of short flows by a factor of up to 5x and 3.6x compared to DT and CS respectively when paired with DCTCP. With TCP, it improves tail FCT of short flows up to 3x and 5x for DT and CS respectively.
  - **Average Short FCT:** For both TCP and DCTCP, *Protean* improves average short FCT up to 1.18x compared to DT, and for TCP, it improves short average FCT up to 1.6x compared to CS.
- **Goodput** We compare network goodput of *Protean* with DT and CS for the long flows. *Protean*'s goodput is always similar to that of DT or CS and in high load with TCP it gets slightly better. In other words, *Protean* does not sacrifice throughput to obtain better FCT.

1) Flow completion times: Fig. 4 compares the CDF of incast flow completion times for different buffer management schemes under two different loads, when TCP and DCTCP are used for congestion control. As we can see both DT and CS have long tails that signifies packet drops leading to timeouts. This is especially true for TCP, because it lacks active queue management (AQM) that reacts to ECN signal. *Protean* on the other side has a short tail and outperforms other approaches in both lower and higher loads under TCP and DCTCP.

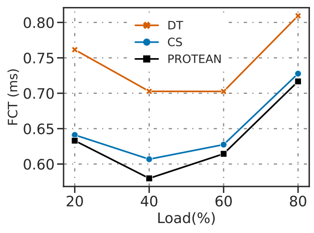
To have a closer look at flow completion times under different loads, we plot incast tail and average FCT across load



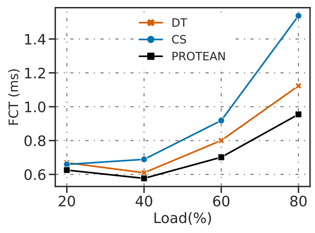
(a) 99.9th %-ile with DCTCP



(b) 99.9th %-ile with TCP



(c) Average with DCTCP



(d) Average with TCP

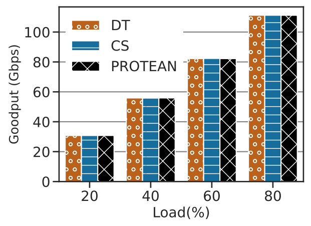
Fig. 6: Short (non-incast) flow completion times

in Fig. 5. As seen in Fig. 5(a) and Fig. 5(b), *Protean*'s tail FCT performance remains almost invariant across various loads for both TCP and DCTCP and it significantly outperforms DT and CS. Fig. 5(c) and Fig. 5(d) study the average incast FCT and show the performance of average incast flow is also better under *Protean*. This is because *Protean* relies on accurately isolating incasts from other traffic and ensuring that they are absorbed by providing buffer proportional to their fan-in.

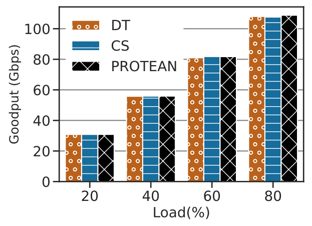
We study the performance of *Protean* for non-incast short flows in Fig. 6. In Fig. 6(a), both DT and CS incur timeouts to short flows at higher loads. For DT, this is due to reserving a fraction of buffer and not being able to accommodate short flow packets. For CS, this is due to letting the long flows occupy as much buffer as they can. However, since DCTCP still tries to maintain the queues short by reacting to ECN, long flows cannot completely monopolize the buffer which benefits short flows and therefore CS performance is slightly better than with DT. With TCP and lack of AQM, CS loses this benefit at high load as depicted in Fig. 6(b). *Protean* does not sacrifice short flows for incasts and tries to adhere to a best-effort policy for them. This can be seen for both tail as well as average FCTs (as seen in Fig. 6(c) and Fig. 6(d)) across all loads.

2) Goodput: Prioritizing incast occupancy in buffer may lead to throughput loss if performed naïvely. Too strict thresholds for long flows and drastic drops from them can easily degrade network throughput. However, *Protean* reduces the threshold for long flows only when it detects a large incast. Because incasts are usually short lived, drastic packet drops from the long flows during that time would not deteriorate *Protean*'s throughput. In fact as soon as *Protean* senses that the gradient is small, it falls back to DT for its threshold and that indirectly helps long flows. This is shown in Fig. 7 where we change the load and measure network goodput for long flows. *Protean* achieves similar goodput to both DT and CS under TCP and DCTCP.

3) Robustness to buffer pressure: To show *Protean*'s robustness to various burst sizes, we run an experiment with loads 40% and 80%. We fix the number of senders as before but we change incast size as a percentage of buffer size and measure the tail FCT. The results are shown in Fig. 8. For both loads, as the incast size increases, there is a point that either DT or CS incur losses leading to timeouts but *Protean*'s

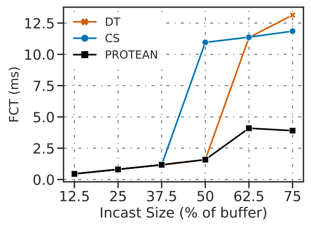


(a) DCTCP

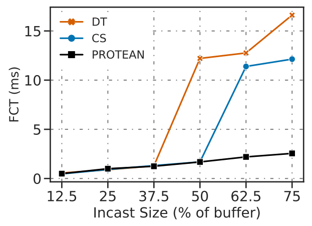


(b) TCP

Fig. 7: Goodput



(a) Load=40%



(b) Load=80%

Fig. 8: The effect of burst size on buffer pressure

FCT increases almost linearly.

4) Robustness to design parameters: We also evaluate *Protean*'s robustness to design parameters, namely β and $buildup\_threshold$. As mentioned in Section III-A, β should be large enough to capture consecutive packets in a burst and small enough to filter out noisy switch measurements. In Section III-B we also mentioned that in order to isolate small bursts from large ones, *Protean* threshold should override that of DT only if the gradient is large enough that we determine by comparing to $buildup\_threshold$. We run an experiment to test the sensitivity of *Protean* to these parameters. In this experiment, we set the load to 50% and change these parameters over a range of reasonable values according to our discussion in Fig. 9 shows the tail FCT remains robust to a combination of values for parameters.

C. Hardware Testbed Evaluation

Following our solutions to challenges listed in Section III-C, we implemented a prototype of *Protean* on a Wedge 100BF-32X switch with Tofino chipset in P4 language [7]. In this

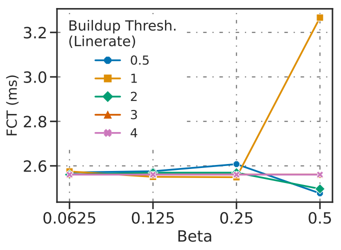


Fig. 9: *Protean*'s sensitivity to its design parameters. Tail FCT for Load=50%

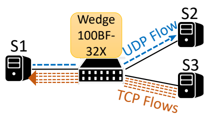


Fig. 10: Hardware testbed

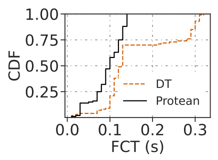


Fig. 11: CDF of TCP flow completion times

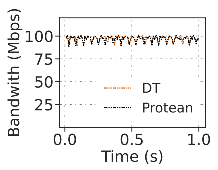


Fig. 12: UDP bandwidth

section we present the overheads and experimental results with the hardware testbed.

We connect the switch to three Linux servers in a star topology as shown in Fig. 10. Each server has 16 cores of Intel(R) Xeon(R) Silver 4108 CPU @ 1.80GHz, 64GB RAM, a Mellanox ConnectX-5 100Gbps NIC, and has Linux 4.15.0-188. We set the total application pool usage of the buffer to 21,000 cells and as mentioned in Section III-C, to fill up the switch buffer we rate limit the output ports to 100 Mbps with the traffic manager API. Tofino supports a limited set of alpha values from which we pick $\alpha_{incast} = 0.8$ and $\alpha_{non-incast} = 0.5$. Using iperf [23], we first start a background UDP flow from S_1 to S_2 and then create 100 short TCP flows, 16 KB each, in parallel from S_3 to S_1 with TCP $RTO_{min} = 200ms$.

1) Results: We measure the FCTs for TCP flows and throughput for the UDP flow under DT and *Protean* and show the results in Fig. 11 and Fig. 12. *Protean* improves *tail FCT by a factor of 2.3x* and *average FCT by a factor of 1.7x* while maintaining the same throughput for the background flow.

2) Switch resource overhead: In Algorithm 2, there is one division (line 6) and one multiplication (line 11) operation that cannot be performed only by shifting. For each of them we use *two ternary match table* lookups to approximate logarithms and *one exact match table* lookup to approximate exponentiation. Ternary matches use TCAM and exact matches use SRAM memory. Overall, with 32 bit operands and with a *mean error of less than 1%*, *Protean* uses less than 8.2 KB of SRAM and less than 18.8 KB of TCAM, i.e., *a total of less than 27 KB of memory specifically for floating point operations*. In summary, our program occupies ten stages with *an average of 3% SRAM and 3.5% TCAM usage per stage*, i.e., a negligible overhead considering that modern commodity switches have a few tens of megabytes of on-chip memory [24], [25].

V. Related work

We discussed CS and DT [4] buffer allocation policies in detail in previous sections. Among other buffer allocation schemes, EDT [26] detects whether a port is experiencing microburst (incast), temporarily relaxes the fairness constraint, and behaves similar to CS during the period microburst exists, i.e., it allows the port to use as much of the buffer as it can. However, it has a major drawback since it cannot differentiate between bursts of different sizes and cannot cap the threshold proportionally when more than one port experiences burst which is frequent in datacenters [11]. FAB [27] maps flows to several α values per port based on their size, with short flows having a higher value and therefore higher threshold. Yet, these α values do not represent burst signatures and cannot be dynamically adjusted for optimal burst absorption. Smartbuf [28] focuses on learning upper bounds for buffer occupancy of bursts, i.e., their demand, and capping the ports that experience similar gradients at those levels. However, it has parameters that need to be tuned when workloads change and its extension for multiple priorities per port is not obvious. *Protean*, on the other hand, dynamically sets the threshold based on burst gradient. It also embeds static (set by the network operator or end-hosts) alphas in its formula for any other notion of priority both across bursts and long flows, while doing a best-effort allocation for non-burst short flows.

Congestion control algorithms [3], [10], [29]–[31] that react to network signals, try to keep the queue lengths small, but still need at least one RTT for response. They also have no visibility into the share of other ports out of total buffer occupancy. Scheduling algorithms [32]–[35] work within the domain of ports and enable preferential dequeueing policies for queues with different priorities. These algorithms provide control over queueing delays and are complementary to buffer allocation policies but they cannot control burst absorption because it needs broader knowledge of switch's other flows. Active queue management schemes [22], [36], [37], work in conjunction with packet scheduling algorithms. Similar to buffer management schemes, they try to reduce the congestion by performing admission control and dropping packets probabilistically but their scope is limited to queues.

VI. Conclusion

We present *Protean*, an adaptive buffer management scheme in shared-memory datacenter switches that detects and absorbs large bursts based on switch-local information while providing low flow completion times for other short flows and high throughput for long flows. We evaluate Protean and show it outperforms the state-of-the-art buffer management policy, dynamic thresholds, as well as complete sharing both with TCP and DCTCP. We also build a prototype for *Protean* on programmable switches and show its implementation feasibility.

References

[1] L. A. Barroso, U. Holzle, P. Ranganathan, and M. Martonosi, *The Datacenter As a Computer: Designing Warehouse-Scale Machines*, 3rd ed. Morgan & Claypool Publishers, 2018.

[2] G. Appenzeller, I. Keslassy, and N. McKeown, "Sizing router buffers," in *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. NY, USA: Association for Computing Machinery, 2004, pp. 281–292.

[3] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *Proceedings of the ACM SIGCOMM 2010 Conference*, NY, USA, 2010, pp. 63–74.

[4] A. Choudhury and E. Hahne, "Dynamic queue length thresholds for shared-memory packet switches," *IEEE/ACM Transactions on Networking*, vol. 6, no. 2, pp. 130–140, 1998.

[5] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, NY, USA, 2013, pp. 99–110.

[6] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, jul 2014.

[7] P4 language specification. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.2.2.pdf

[8] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, NY, USA, 2015, pp. 139–152.

[9] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng, "Packet-level telemetry in large datacenter networks," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, NY, USA, 2015, pp. 479–491.

[10] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats, "Timely: Rtt-based congestion control for the datacenter," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, NY, USA, 2015, pp. 537–550.

[11] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy, "High-resolution measurement of data center microbursts," in *Proceedings of the 2017 Internet Measurement Conference*, NY, USA, 2017, pp. 78–85.

[12] M. Arpaci and J. A. Copeland, "Buffer management for shared-memory atm switches," *IEEE Communications Surveys & Tutorials*, vol. 3, no. 1, pp. 2–10, 2000.

[13] Cisco nexus 3000 series nx-os qos configuration guide. [Online]. Available: https://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus3000/sw/qos/93x/configuration/guide/b-cisco-nexus-3000-nx-os-quality-of-service-configuration-guide-93x/b-cisco-nexus-3000-nx-os-quality-of-service-configuration-guide-93x_chapter_011.html

[14] Cisco nexus 9000 series nx-os quality of service configuration guide. [Online]. Available: https://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus9000/sw/93x/qos/configuration/guide/b-cisco-nexus-9000-nx-os-quality-of-service-configuration-guide-93x/b-cisco-nexus-9000-nx-os-quality-of-service-configuration-guide-93x_chapter_01000.html

[15] packet buffers. [Online]. Available: https://people.ucsc.edu/~warner/buffer.html

[16] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, NY, USA, 2015, pp. 123–137.

[17] Edgecore wedge 100bf-32x. [Online]. Available: https://www.edge-core.com/productsInfo.php?cls=1&cls2=180&cls3=181&id=335

[18] N. K. Sharma, A. Kaufmann, T. Anderson, C. Kim, A. Krishnamurthy, J. Nelson, and S. Peter, "Evaluating the power of flexible packet processing for network resource allocation," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*. USA: USENIX Association, 2017, pp. 67–82.

[19] The ns3 network simulator. [Online]. Available: https://www.nsnam.org

[20] W. Bai, S. Hu, K. Chen, K. Tan, and Y. Xiong, "One more config is enough: Saving (dc)tcp for high-speed extremely shallow-buffered datacenters," in *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, 2020, pp. 2007–2016.

[21] M. Alizadeh and T. Edsall, "On the data path performance of leaf-spine datacenter fabrics," in *2013 IEEE 21st Annual Symposium on High-Performance Interconnects*, 2013, pp. 71–74.

[22] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 397–413, 1993.

[23] iperf - the ultimate speed test tool for tcp, udp and sctp. [Online]. Available: https://iperf.fr/

[24] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proceedings of the 26th Symposium on Operating Systems Principles*. NY, USA: Association for Computing Machinery, 2017, pp. 121–136.

[25] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya, "Incbricks: Toward in-network computation with an in-network cache," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. NY, USA: Association for Computing Machinery, 2017, pp. 795–809.

[26] D. Shan, W. Jiang, and F. Ren, "Analyzing and enhancing dynamic threshold policy of data center switches," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 9, pp. 2454–2470, 2017.

[27] M. Apostolaki, L. Vanbever, and M. Ghobadi, "Fab: Toward flow-aware buffer sharing on programmable switches," in *Proceedings of the 2019 Workshop on Buffer Sizing*. NY, USA: Association for Computing Machinery, 2020.

[28] H. Rezaei, H. Almasi, and B. Vamanan, "Smartbuf: An agile memory management for shared-memory switches in datacenters," in *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*, 2021, pp. 1–7.

[29] H. Almasi, H. Rezaei, M. U. Chaudhry, and B. Vamanan, "Pulser: Fast congestion response using explicit incast notifications for datacenter networks," in *2019 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, 2019, pp. 1–6.

[30] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik, "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*, NY, USA, 2017, pp. 29–42.

[31] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, and M. Yu, "Hpcc: High precision congestion control," in *Proceedings of the ACM Special Interest Group on Data Communication*, NY, USA, 2019, pp. 44–58.

[32] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "Pfabric: Minimal near-optimal datacenter transport," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, NY, USA, 2013, pp. 435–446.

[33] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "Pias: Practical information-agnostic flow scheduling for commodity data centers," *IEEE/ACM Trans. Netw.*, vol. 25, no. 4, pp. 1954–1967, aug 2017.

[34] Z. Yu, C. Hu, J. Wu, X. Sun, V. Braverman, M. Chowdhury, Z. Liu, and X. Jin, "Programmable packet scheduling with a single queue," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, NY, USA, 2021, pp. 179–193.

[35] N. K. Sharma, C. Zhao, M. Liu, P. G. Kannan, C. Kim, A. Krishnamurthy, and A. Sivaraman, "Programmable calendar queues for high-speed packet scheduling," in *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation*. USA: USENIX Association, 2020, pp. 685–700.

[36] K. Nichols and V. Jacobson, "Controlling queue delay: A modern aqm is just one piece of the solution to bufferbloat." *Queue*, vol. 10, no. 5, pp. 20–34, may 2012.

[37] R. Pan, B. Prabhakar, and K. Psounis, "Choke - a stateless active queue management scheme for approximating fair bandwidth allocation," in *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No.00CH37064)*, vol. 2, 2000, pp. 942–951.