Smartbuf: An Agile Memory Management for Shared-Memory Switches in Datacenters

Hamed Rezaei*, Hamidreza Almasi*, and Balajee Vamanan
Department of Computer Science, University of Illinois at Chicago, USA
Email: {hrezae2, halmas3, bvamanan}@uic.edu

***Abstract*—Important datacenter applications generate extremely bursty traffic patterns and demand low latency tails as well as high throughput. Datacenter networks employ shallow-buffered, shared-memory switches to cut cost and to cope up with ever-increasing link speeds. End-to-end congestion control cannot react *in time* to handle bursty, short flows that dominate datacenter traffic and they incur buffer overflows, which cause long latency tails and degrade throughput. Therefore, there is a need for *agile*, *switch-local* mechanisms that quickly sense congestion and provision enough buffer space *dynamically* to avoid costly buffer overflows. We propose *Smartbuf*, 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.**

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

As we migrate data to datacenters and cloud, we increasingly rely on online services (e.g., Web Search) to retrieve data in a timely fashion—the services run in datacenters and often lie in the critical path of interactive, end-user experience. Owing to their large datasets, online services distribute data among a large number of servers; such a distributed architecture also provides good fault tolerance. Their data distribution and interactive nature have two key implications: (1) Every user query must fetch data from a large number of servers and responses to user queries cannot complete without fetching data from all or most servers. Therefore, the overall response time depends on the *tail* (e.g., 99^{th} percentile) flow completion times [1]. (2) Because the user query naturally synchronizes responses from several servers (i.e., 10–100s), the underlying network experiences a highly bursty traffic pattern called *incast* [2]. The challenge of achieving a low tail latency in the presence of incast is well known.

Today's datacenters use shared-memory switches with *shallow* buffers, which further exacerbates the problem [3]. Because incast is a synchronized burst of flows destined to the same output port of a switch, buffers build up and overflow; the resulting packet drops and the accompanied waiting for TCP timeout adversely affects tail latency. Recent congestion control algorithms address the incast problem. The efficacy of congestion control approaches, however, is limited by feedback latency (i.e., round trip time) and algorithms' convergence times, which range from a few to several tens of round trips. While this is not a serious limitation for long flows, recent measurements from datacenters show that incast flows are typically short (e.g., < 1 KB [4,5]) and last only a handful of rounds; further, most incasts last shorter than typical Round Trip Times (RTT). Packet scheduling approaches [6] also do not help because the flows that contend for bandwidth during an incast have identical priority (e.g., size, deadline).

We can avoid the costly TCP timeouts if we can provision enough buffer space to absorb incasts. In general, the amount of buffer required per port is proportional to bandwidth-delay product. The buffer size of Internet routers can be reduced because of statistical multiplexing among flows that share the link and the lack of synchronization among flows [7]. However, the preceding observations do not hold true in datacenters—a much smaller number of flows share a link and applications produce synchronized flows by design [8]. With network bandwidth scaling faster than memory (Moore's law) and delay mostly bounded by the speed of light, large buffers for today's extremely fast datacenters is not a practical design choice [9]. Fortunately, not all ports are likely to experience synchronized flows (i.e., incasts) at the same time and their buffer demands change over time. This diversity among ports provides an opportunity to *dynamically* allocate memory to ports. Existing proposals for dynamic memory allocation identify congested ports and allocate either a portion [10] or all [11] of the memory *equally* among congested ports— while existing proposals detect congested ports, they do not estimate the *extent* of congestion experienced by each port. Therefore, if two or more ports experience congestion (e.g., incast) to varying degrees, they would allocate an equal amount of memory to all congested ports and their performance would suffer. We observe this phenomenon in our experiments (Section IV).

In this paper, we propose *Smartbuf*, an agile memory management scheme for efficiently sharing memory among switch ports by *accurately predicting the buffer demands of ports* and to allocate memory proportional to their demands. As compared to the binary decision of identifying whether a port is congested or not, estimating buffer demands with

*Equal contribution

978-0-7381-3207-5/21/$31.00

high accuracy is challenging. We make *three key observations* that enable us to accurately predict buffer demands at switch ports: (1) Ports require large buffers during incasts and the fan-in (incast) degree is a stronger predictor of buffer demands than other factors such as flow sizes and congestion control algorithms; (2) The gradient of queue length at an output port of a switch when sufficiently smoothed out (averaged) serves as a reliable *proxy* for fan-in/incast degree; (3) The peak buffer demands of incasts change at relatively longer timescales (e.g., seconds–minutes). Our first observation follows from the fact that most incasts involve small flows (i.e., flow size is not a key factor) and the flows do not last enough for congestion control algorithms to make a significant difference. The second observation follows from the observation that the gradient of queue length reflects the aggregate incoming rate, which is largely determined by incast degree (from (1)). The last observation follows from the fact that workload characteristics change when workloads are added/removed or the nature of the workload changes over time (e.g., software upgrades). Thus, workload characteristics change at relatively longer timescales and allow dynamic buffer adjustments to be effective. The preceding observations enable us to reliably predict future demands based on past allocations (*history*) and to use *local information* (i.e., gradient of queue length) as a *key* to our history, which we maintain as a key-value store.

Motivated by these observations, we present an *online algorithm* (Algorithm 1) for estimating buffer demands based on the gradient of queue length. Further, because buffer demands depend only on fan-in and not on ports, ports can learn from each other. By allocating buffer space proportionally based on demands, *Smartbuf* avoids some ports from needlessly monopolizing buffer space and starving others. This results in improved fairness as well (Figure 4). Finally, multiple incasts can collide and create a larger incast, by accident. However, such occurrences are rare and our algorithm, which is a variant of *K-nearest neighbors*, naturally weeds out statistical outliers.

We summarize our contributions as follows:

- **Novelty:** We show that it is possible to accurately predict buffer demands of switch ports based only on switch-local information without requiring application knowledge or global coordination.
- **Preliminary system design:** We present our complete algorithm for buffer allocation that accurately predicts buffer demands using only a small amount of state.
- **Promising results:** Our preliminary evaluation on ns-3 shows that *Smartbuf* accurately predicts buffer demands within an average error margin of 6% and improves tail latency by a factor of 8x over existing state-of-the-art mechanisms at high loads without sacrificing fairness among ports.

II. Background and motivation

Bursts in high fan-in scenarios such as incast can cause packet drops that are only detected by timeouts. Therefore, achieving the low-latency goal in datacenters translates to avoiding excessive and persistent packet drops caused by periodic spikes in queue occupancy when buffer-hungry protocols are used [6]. While switch buffer overflow and congestion are the most important reasons for packet drops [12,13], they both are caused by incast of short flows.

There are two main solutions for incast problem: (1) congestion control methods that use a combination of in-network signals (e.g., ECN notification) and end-host rate control mechanisms (e.g., adjusting the size of sending window) to detect and react to congestion. (2) Employing dynamic memory management schemes at the switches that grow the share of a specific port without waiting for endhosts' reaction, whenever a port becomes overloaded. Below we discuss both methods in details.

There is a large body of work on congestion control in the past decade [8,14]. However, these methods are ineffective when flows are extremely small, which is the case in modern datacenter networks [4]. A recent study on Facebook's datacenter reveals that almost 70% of the flows are below 1 KB in size [4]. While most congestion control methods are able to solve moderate forms of congestion that last for a few RTTs, they are ineffective in handling incast of extremely short flows that mostly finish in less than one RTT.

Therefore, the preferred approach to handle incast of short flows is to efficiently manage switch buffers. We now describe the state-of-the-art memory management schemes that determine how the switch buffer is shared among ports, and their effect on controlling incast congestion. In the simplest case called *complete partitioning* [15], each port is allowed to use up to a fixed amount of shared buffer space and the sum of these amounts is equal to the total memory. In the other extreme, we have *complete sharing* in which a packet is accepted at the switch if any space remains in the whole shared pool of switch memory for it. Intuitively, complete sharing allows one port to monopolize most of the memory if it is highly utilized compared to the other ports. When the load on the ports is balanced, complete partitioning works well. However, if one or more ports experience severe congestion and require more space, albeit for a short duration, complete sharing will provide space and avoids long tails that occur from the consequent packet drops.

Complete sharing loses its superiority under such traffics in which both short incast flows and large flows exist at a switch, as reported in recent studies such as [16]. In this case, if there is a transient burst among some ports while other ports are busy with transmitting large flows, there would be packet drops because complete sharing assigns a large fraction of the buffer to large flows (i.e., TCP's slow start doubles the sending window after each transmission), which leaves little to no space for short-lived, incast flows. We observed this in our experiments and we will provide the results of this experiment later in Section IV.

The dynamic threshold (DT) [10] policy takes a middle ground between the complete partitioning and complete sharing extremes. In DT, a port is allowed to use a fraction, not all, of the unused buffer space. If $T(t)$ is the buffer threshold

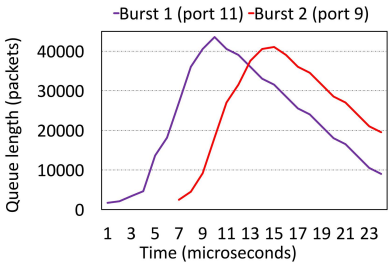

Fig. 1. Bursts in two different ports within a short time

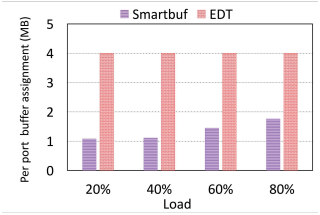

Fig. 2. Maximum buffer allocation in *Smartbuf* vs EDT

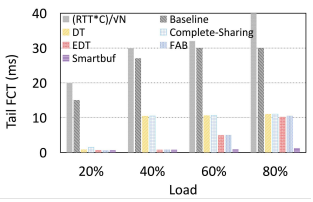

Fig. 3. 99^{th} percentile flow completion time

(i.e., maximum size of the buffer that any one port can use) at time t, then DT allocated the threshold as follows:

$$T(t) = \alpha \times (B - Q(t)) \quad (1)$$

In Equation (1), B is the total buffer space, $Q(t)$ is the sum of all the queue lengths, and α is a configurable parameter that determines the aggressiveness of DT (i.e., as $\alpha = 1$ increases, we gradually move from complete partitioning to complete sharing). DT robustly adapts to changes in the traffic conditions such as load. A load change causes the system to go through a transient state. For example, when the load of one port suddenly increases, its queue will build-up and the remaining buffer will decrease. This causes a decrease in the threshold and the set of queues exceeding this new threshold will stop accepting new packets temporarily until they drain and provide more buffer space for the newly loaded queue. DT is fair because it allocates an equal amount of buffer to the ports that concurrently need some. Variants of DT are used in today's switches. For instance, in a shared-buffer switch like Broadcom Trident with 12 MB of buffer space, each output port receives a small amount of dedicated space while the remaining is dynamically allocated and a single congested port can consume up to 10.5 MB [17]. As more ports become congested, each port receives a smaller share. However, DT does not perform well in datacenters. While microbursts (incasts) of short flows are common and the tail latency of short flows is supremely important in datacenters, DT would not increase the buffer share of ports that are experiencing incast and would fail to absorb microbursts. Thus, DT does not respond to incasts in a timely fashion and suffers from packet drops.

The enhanced dynamic threshold (EDT) [11] temporarily relaxes the fairness constraint and allows a port to temporarily use most/all of the unused buffer size. However, there is one major drawback. *While EDT predicts whether a port is experiencing incast, it does not estimate the port's actual demand for buffer space. Because EDT does not estimate buffer demands of ports, when more than one port experiences incast, EDT cannot proportionally allocate the unused buffer space.* The above limitation is serious if we consider the nature of datacenter workloads. Specifically, we know that load is relatively unbalanced and simultaneous bursts drive buffer usage non-linearly. For example, for data mining and offline analysis workloads like Hadoop, there are times when all ports in use are utilized more than 50%. This also holds to a lower extent for web search and in-memory cache workloads [4]. We simulated the web search workload in a typical datacenter topology (see Section IV-A for our evaluation methodology) to check whether multiple ports experience congestion around the same time and whether such occurrences are common. Figure 1 shows how the queue lengths of two different ports evolve over time. We clearly see two incasts occurring closer in time in different ports. Indeed, we observe this behavior commonly (we do not show a long time window because such a plot will not be legible). The preceding observation suggests that allocating the whole buffer to one port would not only be unfair but also may not help the tail FCT in typical datacenter workloads.

We already discussed the inefficiency of static partitioning, DT, and EDT in lowering tail flow completion times and predicting the actual buffer demand for each port. But why *Smartbuf* would be fair? *Smartbuf* is fair because it predicts the buffer demand of each port, and based on that, puts a cap on the maximum buffer size that each port will receive. Therefore, no port will usurp the *whole* buffer and there will be enough space available for other ports that could become overloaded at the same time. While *Smartbuf* provides this flexibility, EDT assigns the whole buffer to a single port upon detecting a burst. We designed an experiment to see the difference between *Smartbuf*'s maximum buffer assignment, versus that of EDT when a burst arrives at a port. Figure 2 shows the result of this experiment. While *Smartbuf* assigns less than half of the whole buffer to a single burst in all loads, EDT dedicates the entire buffer to a burst regardless of load and activity of other ports. As a result, *Smartbuf* is always fair because it leaves considerable amounts of buffer for other ports. Note that we did not observe any packet drops when *Smartbuf* was employed in this experiment. That being said, the amount of buffer that *Smartbuf* predicted was very accurate and there was no need to reserve the whole buffer for a single port.

III. Proposed Work

A. High-level idea

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.

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 [4,5], *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 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 Section IV, 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.

B. Smartbuf algorithm for buffer allocation

In this section, we explain the details of *Smartbuf*'s buffer allocation mechanism. As we mentioned in Section I, *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 1. 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 5 of Algorithm 1), and records **all** the information. (2) Main_map, which is a hash map as well, and records the burst-related information **only** (line 6 of Algorithm 1). *Smartbuf* kicks in only if a burst is detected. Thus, when no burst is detected, each port's buffer threshold is calculated by DT [10] (line 7 and line 33 in Algorithm 1), 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 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

Algorithm 1: *Smartbuf*

```
1  Input: instantaneous queue length
2  Learning-phase
3  Initializations:
4  max_buf_seen, Gradient, Gradient_prev ← 0
5  Temp_map[port, [Gradient, max_buf_seen]] = φ
6  Main_map[Gradient, max_buf_seen] = φ
7  Port_buf_threshold = DT threshold      (see [10])
8  for each packet dequeue at port P do
9      Gradient = (Qlen − Qlen_prev)/(T − T_prev)
10     Qlen_prev ← Qlen, T_prev ← T
11     Gradient ← 1/4×Gradient+3/4×Gradient_prev
12     Gradient_prev ← Gradient
13     if (cur_buf_in_use > max_buf_seen)
14         max_buf_seen = cur_buf_in_use
15         Temp_map[P] ← [Gradient, max_buf_seen]
16     else if (cur_buf_in_use < β * max_buf_seen)
17         if (max_buf_seen > (1/σ) * total_buf_size)
18             Main_map ← Temp_map[P]
19         else
20             Gradient, max_buf_seen ← 0
21             Temp_map[P] ← [Gradient, max_buf_seen]
22     else
23         Continue
24 Real-phase
25 at line 17 of learning phase:
26 if (max_buf_seen > (1/σ) * total_buf_size)
27     for i = 1; i < K; i = i + 1 do
28         find i^th larger than current Gradient key
             in Main_map and store it in min(i)
29     avg = (min(1) + ... + min(K))/K
30     Port_buf_threshold = avg
31     Main_map ← Temp_map[P]
32 else
33     Port_buf_threshold = DT threshold      (see [10])
```

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 13–15 of Algorithm 1, 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 16 through 23 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 16), 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 Section IV, 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 17 of Algorithm 1 is about this condition. Similar to line 16, 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 Section IV.

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 27–31 of algorithm 1. 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 31), 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 IV, we discuss the range of values of K that lead to high performance.

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. 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.

IV. Evaluation

We conduct experiments to evaluate three different aspects of our approach: (1) performance (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.

A. 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 [18] 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 [8] 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 [10]. while link speed of leaf switches and spine switches is 10 Gbps across the network, we set the network round trip-time to 80 microseconds, which is very close to that of modern datacenters. Also, we use DCTCP [8] 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 [10], Complete sharing, EDT [11], and a recent work called FAB [16]. FAB [16]

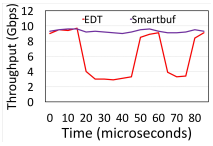

Fig. 4. Non-burst flows' buffer share in EDT vs. *Smartbuf*

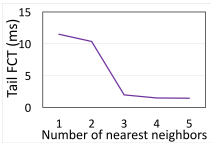

Fig. 5. Sensitivity study of k values

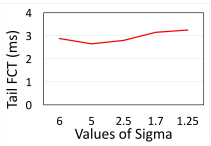

Fig. 6. Sensitivity to σ at 80% load

extends DT by using multiple parameters per port (see α from Equation 1) 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.

B. Performance

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 [7], 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 [10], [11], and [16] respectively. Figure 3 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 [7] (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 3, 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 [11,16]. 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 Section II, 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 makes sub-optimal allocations and suffers from longer tails. As we see in Figure 3, *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 4. 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.

C. Parameter sensitivity

In this section, we discuss the sensitivity of our algorithm to two main factors that we mentioned in Algorithm 1: β and σ. Also, we discuss the sensitivity of our scheme to different values of k in the k-nearest neighbors algorithm. Figure 5 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 8).

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 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 6 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

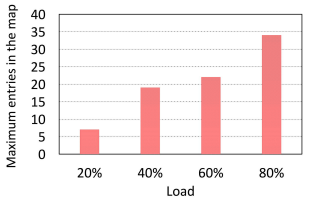

Fig. 7. Number of entries in the hash table

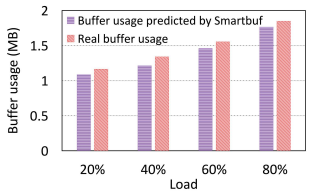

Fig. 8. Predicted vs. actual buffer demands: maximum prediction error at each load

smallest value of σ that guarantees high performance (e.g., $2.5 < \sigma < 5$ in Figure 6).

D. Overhead

Next, we analyze the state/memory overhead of our algorithm. Figure 7 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.

E. 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 8 shows predicted demands vs. ideal (oracular) demands for each load level when the difference between the two (error) is **maximum**. The key takeaway is that our predicted demands are very close to the oracular buffer demands (within 9% of the ideal).

V. Related work

In the preceding sections, we have discussed related work in dynamic buffer allocation such as DT [10], EDT [11], and FAB [16]. In this section, we will discuss related work in congestion control [8,19] and flow scheduling. Reactive congestion control schemes [8] require multiple RTTs and do not quickly respond to incast. Recent approaches [14,19] that speed up the response still require at least one RTT. Packet scheduling approaches [6,20] use local information at switches to prioritize certain packets based on both static (e.g., flow size) and dynamic priority (e.g., queue length). However, their efficacy is limited when during incast of short flows because the contending flows have similar priorities. In such cases, *Smartbuf* effectively uses buffer space from unused ports to accommodate all contenting packets (flows). Buffer capacity is a scarce resource in datacenter switches, which are known to use shallow buffers [3]. The application push toward incast of short flows and technology pull toward shallow buffers favor smart management approaches.

VI. Conclusion and future work

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.

References

[1] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, Feb. 2013.

[2] Y. Chen *et al.*, "Understanding tcp incast throughput collapse in datacenter networks," in *Proceedings of the 1st ACM workshop on Research on enterprise networking*, 2009.

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

[4] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy, "High-resolution measurement of data center microbursts," ser. IMC'17. ACM, 2017.

[5] A. Roy *et al.*, "Inside the social network's (datacenter) network," in *ACM SIGCOMM Computer Communication Review*. ACM, 2015.

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

[7] G. Appenzeller, I. Keslassy, and N. McKeown, "Sizing router buffers," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 4, p. 281–292, Aug. 2004.

[8] M. Alizadeh *et al.*, "Data center tcp (dctcp)," ser. SIGCOMM '10, 2010, p. 63–74.

[9] M. Mathis and A. McGregor, "Buffer sizing: a position paper," http://buffer-workshop.stanford.edu/papers/paper16.pdf, (Accessed on 06/23/2020).

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

[11] D. Shan, W. Jiang, and F. Ren, "Absorbing micro-burst traffic by enhancing dynamic threshold policy of data center switches," in *2015 IEEE Conference on Computer Communications (INFOCOM)*, 2015.

[12] C. Guo *et al.*, "Pingmesh: A large-scale system for data center network latency measurement and analysis," ser. SIGCOMM '15, 2015, p. 139–152.

[13] Y. Zhu *et al.*, "Packet-level telemetry in large datacenter networks," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, p. 479–491, Aug. 2015.

[14] M. Handley *et al.*, "Re-architecting datacenter networks and stacks for low latency and high performance," ser. SIGCOMM '17, 2017.

[15] 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.

[16] 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*, ser. BS '19, 2019.

[17] Y. He, N. Batta, and I. Gashinsky, "Understanding switch buffer utilization in clos data center fabric."

[18] G. F. Riley and T. R. Henderson, *The ns-3 Network Simulator*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 15–34.

[19] H. Almasi *et al.*, "Pulser: Fast congestion response using explicit incast notifications for datacenter networks," ser. IEEE LANMAN'19.

[20] H. Rezaei and B. Vamanan, "Resqueue: A smarter datacenter flow scheduler," in *Proceedings of The Web Conference 2020*, 2020.