FAB: Toward Flow-aware Buffer Sharing on Programmable Switches

Maria Apostolaki[1], Laurent Vanbever[1], Manya Ghobadi[2]

[1]ETH Zürich [2]MIT

ABSTRACT

Conventional buffer sizing techniques consider an output port with multiple queues in isolation and provide guidelines for the size of the queue. In practice, however, switches consist of several ports that *share* a buffering chip. Hence, chip manufacturers, such as Broadcom, are left to devise a set of proprietary resource sharing algorithms to allocate buffers across ports. This algorithm dynamically adjusts the buffer size for output queues and directly impacts the packet loss and latency of individual queues. We show that the problem of allocating buffers across ports, although less known, is indeed responsible for fundamental inefficiencies in today's devices. In particular, the per-port buffer allocation is an ad-hoc decision that (at best) depends on the remaining buffer cells on the chip instead of the type of traffic. In this work, we advocate for a flow-aware and device-wide buffer sharing scheme (FAB), which is practical today in programmable devices. We tested FAB on two specific workloads and showed that it can improve the tail flow completion time by an order of magnitude compared to conventional buffer management techniques.

1 INTRODUCTION

Few network engineers would expect network devices with low buffer occupancies to ever drop packets [21, 24]. Surprisingly though, this often happens in practice, begging the obvious question: ***why?!***

We investigated this behavior by experimenting with actual data center switches coming from two switch vendors and equipped with totally different ASIC designs. Our experiments confirmed this apparently counter-intuitive behavior: both devices were indeed dropping traffic despite the abundance of unoccupied buffer space. Our experiments also revealed the culprit: the buffer allocation algorithm, which decides how to split the available shared buffer across different ports. Our devices, as many modern network devices [16], indeed rely on shared buffer memory across ports.

In both devices, the key issue was that the buffer allocation algorithm used by the manufacturer did not allow a port/queue to occupy more than a fraction of the shared buffer, even if there was no other active competing queue. This behavior resembles a dynamic buffer management technique which limits the amount of buffer each queue can use to a fraction of the unused buffer size [15, 26].

While configurable, setting this fraction is very challenging. On the one hand, if the fraction is too small, then in an incast scenario, multiple packets of very short flows will be dropped, while the buffer stays mostly empty. Drops on short flows significantly affect their flow completion time [4, 30] and, consequently, the application performance. On the other hand, if the fraction is too large, then longer flows will gradually occupy most of the buffer space, with no throughput benefits [5, 7, 17, 20]. In practice, the two aforementioned cases often tend to be entangled in the same device and even in the same queue.

As one possible answer, we advocate that buffer allocation decisions should be *device-wide* and *flow-aware*. We show that doing so provides significant improvements over existing allocation schemes. In addition, we show that these decisions can be made in existing programmable devices, *i.e.,* they are practically relevant.

Our algorithm, called Flow Aware Buffer (FAB), splits buffer cells among ports of a device while taking into consideration the utilization of all ports, and the expected benefit of buffering for each flow. With FAB, a device can allocate larger amounts of buffer space to bursty short flows while limiting the buffer space allocated to long flows, and this, independently of their respective destination ports.

Concretely, FAB extends a conventional dynamic buffer management technique [15] that allocates a *single* fraction of the remaining buffer to each port, by using multiple fractions per port. While the exact number of such fractions is flexible, a handful corresponding to different priorities suffice in practice. Next, FAB maps packets dynamically to a fraction, based on flow information or priority, *e.g.,* flow size. Thus, two packets destined to the same destination port might be treated differently (dropped or buffered) depending on the flow they belong to. By doing so, FAB can absorb large bursts of short flows, possibly as big as the entire buffer, on any port, while preventing misbehaving or long-lived flows to consume the buffer. Our preliminary results show that FAB can decrease tail Flow Completion Time (FCT) by one order of magnitude even in the presence of buffer-hungry long flows and without the end-hosts' support.

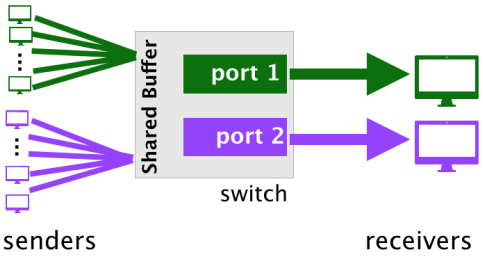


Figure 1: Example in which a switch faces long-lived (resp. transient) congestion on port 1 (resp. port 2).

2 OVERVIEW

We now illustrate the problems behind traditional buffer management approaches and how FAB manages to solve them. We use the scenario depicted in Fig. 1 in which a switch forwards traffic from multiple senders (left) to two receivers (right) via ports 1 and 2. We assume that the traffic towards port 1 (resp. 2) is mainly composed of long (resp. short) flows. Since we are interested in studying buffer allocation, we assume that the incoming rate is higher than the outgoing one resulting in a queue build-up at both output ports. We consider that the switch buffer space is shared across ports and can contain up to 180 packets. Upon congestion, a buffer management technique decides how many packets get buffered for each port.

In the following, we first describe the buffer allocation computed by three conventional buffer management techniques, namely: *Complete partitioning*, *Complete sharing*, and *Dynamic sharing*. We then describe the allocation computed by FAB and how it manages to overcome the limitations of the previous methods. We depict the queue occupancies obtained by each scheme in Fig. 2.

Complete partitioning. This technique simply allocates a static amount of buffer space to each queue. Complete partitioning is ideal for balanced traffic; namely, scenarios in which *all* ports deal with similar load [8]. In all other (and more practical) cases, though, Complete partitioning tends to lead to unnecessary packet drops as the amount of buffer space available to each port ends up being quite small. Thus, packets belonging to a transient burst will be inevitably dropped even if the buffer is almost empty. This is the exact situation depicted in Fig.2a: we see that only a few packets are buffered from each port, while the remaining ones are dropped even though the buffer occupancy is low.

Complete sharing. This technique allows queues to grow arbitrarily large in the shared buffer until it is full. Complete sharing is ideal for imbalanced traffic among ports, namely workloads in which very few ports need buffer. Similarly to Complete partitioning, it is also very simple and, therefore, easy to implement in hardware [8]. If multiple ports are simultaneously competing for buffers though, the excess traffic for these ports will be unrestrictedly and possibly unnecessarily buffered. This is problematic if one of those output ports carry long-lived and high-rate flows as they use buffer space with little to no throughput benefits [7]. If at the same time another output port carries short-lived flows, those will be unable to gain space and will be dropped with significant consequences to their completion time.

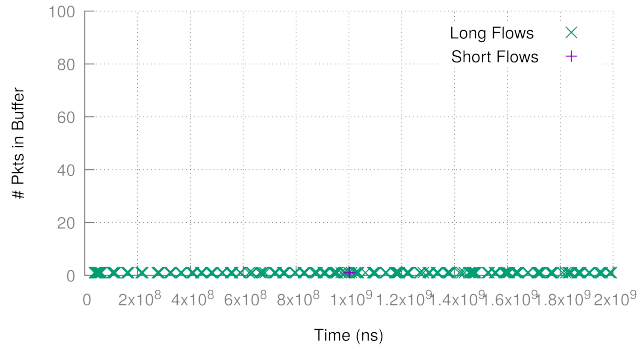
In our example, long flows destined to port 1 monopolize the buffer, leaving almost no space left for the more-bursty port 2 (Fig. 2b). Observe that, unlike Complete partitioning which leaves the buffer mostly idle (Fig. 2a), Complete sharing utilizes it as much as possible, yet not in a way that is necessarily beneficial.

Dynamic sharing. This technique allows each queue to grow up to a dynamically-assigned threshold. This threshold is computed as the product of the remaining buffer with a predefined parameter α [15]. Dynamic sharing is the current state-of-the-art approach, used by multiple vendors, including Broadcom [26]. Unlike the previous two techniques, dynamic sharing allocates buffer space proportionally to the load on the device. Dynamic sharing is also fair as queues that concurrently need buffer will get an equal amount of cells (in the steady-state).

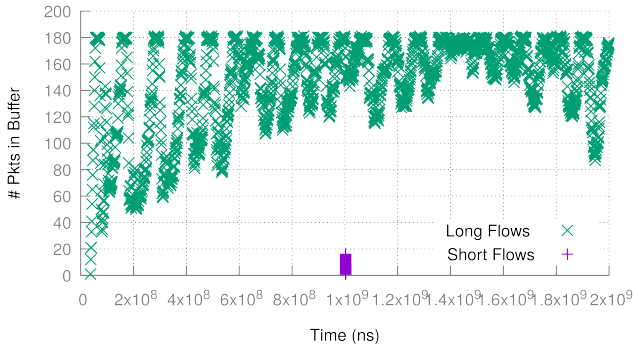
Still, Dynamic sharing has two crucial disadvantages. First, it always leaves some part of the buffer unused, leading to drops that could (and potentially should) have been avoided. Second, Dynamic sharing allocates the same amount of buffer space to ports that concurrently need buffer, regardless of the type of traffic they see or the duration they occupy the buffer. This essentially allows long flows to keep their share continuously occupied while preventing microbursts from using an excessive amount of buffer shortly.

These short-comings are also illustrated in our example (Fig. 2c) in which we configure $\alpha = 0.5$. First, observe that, even when only one port is using the buffer, only a third of the buffer ($\approx$ 60 packets) can effectively be used. This is because $60 \approx .5\times(180-60)$. Even when the burst happens, the overall buffer occupancy is only increased to $\approx$ 100 packets, while the spare buffer is not used to absorb the burst. Second, the fact that port 1 is allowed to continuously occupy 1/3 of the buffer, further reduces the buffer space that is avaliable for the burst, as the buffer consumed by port 1 reduces the remaining buffer size upon the arrival of the burst. Indeed, if the burst had arrived in an empty buffer, it could have used $\approx$ 60-packets, but in our example, it ended up using only $\approx$ 30 and dropping the rest.

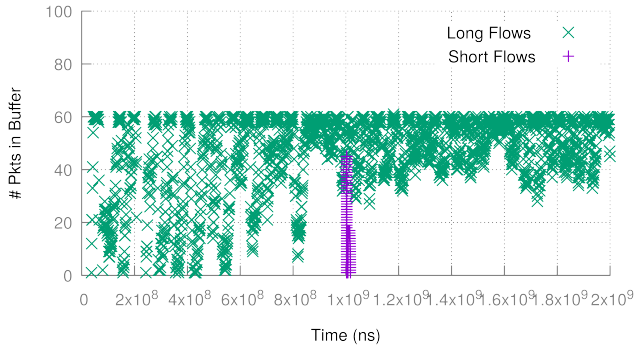
Our solution: FAB. FAB is a generalization of Dynamic sharing, with the addition that FAB allocates buffer space



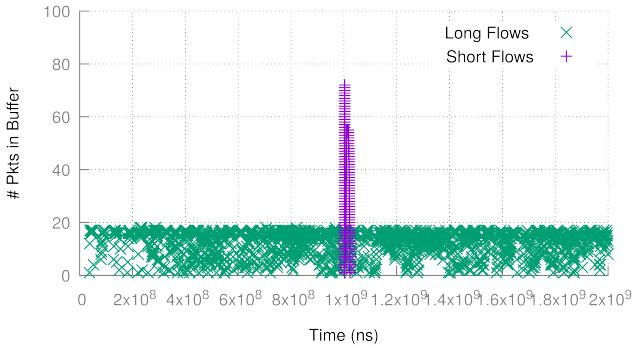
(a) Complete partitioning limits both ports too much.



(b) Complete sharing uses all buffer but unfairly.



(c) Dynamic sharing gives same buffer to short & long flows.



(d) FAB gives short flows as much buffer as they need.

Figure 2: Resulting queue occupancies, under different buffer management schemes

to ports in a flow-aware manner, proportionately to the expected benefit of buffering. Similarly to Dynamic sharing, FAB buffer allocation is dynamic and depends on the remaining buffer space. Unlike Dynamic sharing, though, FAB decisions also depend on the actual traffic seen by each port. As an intuition, FAB gives relatively less buffer space to long flows as these flows benefit less from buffering than short flows.

FAB uses multiple parameters α per port: $\alpha_1, \alpha_2, \ldots, \alpha_n$, (s.t. $\alpha_1 > \alpha_2 > \cdots > \alpha_n$) instead of the single one used by Dynamic sharing. Each incoming packets p is mapped to such a parameter α_x and is buffered if the queue length of its egress queue is shorter than α_x times the instantaneous remaining buffer upon p's reception. FAB maps packets to parameters based on some notion of priority, which depends on the number of packets that the corresponding flow has already transmitted. Thus, packets of short flows will be mapped to larger α parameters and see higher queue limits than long flows even if destined to the same output port and while the buffer is equally utilized. Choosing the number of different parameters as well as their value is challenging. As we will show in §3, FAB starts being useful even with two α parameters: *(i)* one relatively small for long flows; and *(ii)* one arbitrary large for short ones. Indeed, FAB can be augmented by using different queues per port or by using ECN instead of dropping. Likewise, the decisive factor for which flows to prioritize can be different from flow size and dynamically decided by the switch, or set by the end-hosts.

Coming back to our example, let us assume FAB is used to allocate buffer space with $\alpha_1 = 10$ and $\alpha_2 = .1$. Packets belonging to long flows (after the first few ones) will be mapped to α_2 due to the number of packets those flows have transmitted. As such, they would be allowed to buffer up to 18 packets, as shown in Fig. 2d. On the contrary, short flows will manage to finish transmission before they get degraded to α_2. Thus they will assume their limit is 10 times the remaining buffer. Essentially, short flows are allowed to take as much of the buffer is not taken. Of course, as they grow in the buffer their limit will also decrease, as the remaining buffer will decrease due to their own consumption.

3 PRELIMINARY EVALUATION

In this section, we summarize our preliminary results. FAB improves tail FCT by one order of magnitude compared to Dynamic Sharing for an equally-sized buffer and for the tested workloads.

Methodology. We implement all Buffer Management techniques mentioned in §2 in ns3 [3]. Our simulated environment is composed of a star topology in which 200 leaf nodes are connected to a hub node via 100Mbps links. Leaf nodes establish TCP connections to each other via the hub node. We create fan-in scenarios by sending long or short flows from multiple leaf nodes to one receiver. Short flows carry 2.5KB of data, and long flows carry 25MB.

We measure FCT for short flows at the receivers, while changing the buffer management technique at the hub node. Complete sharing is implemented using Dynamic sharing with a high α, in particular, $\alpha = 1000$. In Dynamic sharing we set $\alpha = 0.5$. Finally, in FAB, we use two alpha parameters, namely $\alpha_1 = 10$ for the first 15 packets and $\alpha_2 = 0.5$ for the next ones. New limits are calculated and enforced periodically every 100ns of simulation time for each technique.

We simulate two scenarios that each lasted 2 seconds and compare FCT and the distribution of packet drops across the different buffer management techniques. In particular, for FCT, we report the 50th, 70th, 90th, and 99th percentile across short flows (shown in blue, orange, green, and red bars, respectively). For packet drops, we report the sum of dropped packets (blue bar), the number of those that belong to short flows (green bar), and the number of those that correspond to the handshake of a connection (orange bar). Flows that had not finished by the end of the simulation are assumed to have a FCT equal to the time difference between their SYN and the end of the simulation.We omit the results for Complete partitioning as most of the flows did not finish due to the restricted buffer space.

FAB is as good as Complete sharing in the absence of long flows. We first run a scenario in which one port receives a burst of flows while no long flow exists. As such, all other ports are idle. We plot the FCT and packet drops in Figs 3a and 3b, respectively. We see that Complete sharing and FAB have similar performance. Neither of them drops any of the packets that belong to the burst. On the other hand, Dynamic sharing allows the bursty port only to occupy a small portion (33%) of the buffer, and thus drops packets that belong to the burst, as shown in Fig. 3b. This has a significant impact on the tail FCT. In particular, Dynamic sharing causes the 90th and 99th percentile of FCT to be an order of magnitude higher, while the buffer is only 33% occupied. On the other hand, by keeping the queue smaller, Dynamic sharing results in lower FCT to more than half of the short flows.

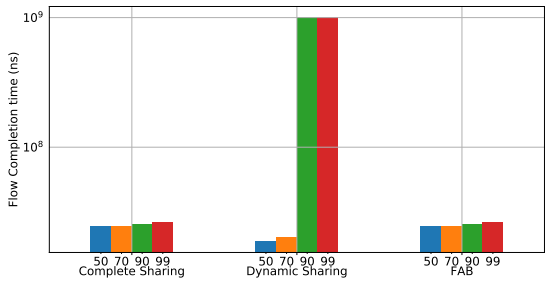
FAB outperforms all approaches when there are both long and short flows. We run the scenario described in Fig.1. Namely, two sets of 100 senders disseminate traffic to two ports. One of the two sets only sends short flows while the other one only long ones. The port with the long flows runs at full capacity with all buffer management techniques. In Figs. 3c and 3d, we see the FCT and the dropped packets' distribution, respectively. In this scenario, Complete sharing is clearly suboptimal. Long flows will quickly consume all buffer, causing starvation (high FCT and drops) to the incoming burst. Dynamic sharing and FAB have similar performance in the 50th and 70th FCT percentiles, with FAB being slightly worse as it allows longer queues to form, and thus causes higher queueing delay. Even so, FAB has lower tail FCT as it manages to avoid drops of short flows. Interestingly, FAB also seems to be fairer among short flows, offering a more predictable FCT.

4 PRACTICALITY

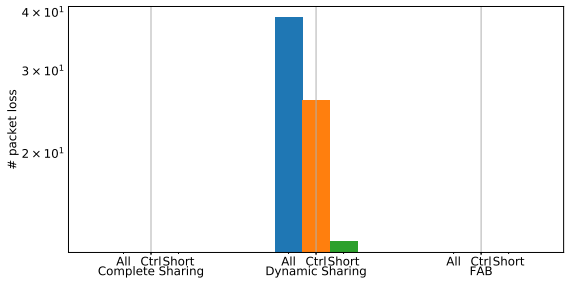
Until recently, implementing a new buffer management technique would have required the vendor's support and, therefore, time. Luckily though, that restriction does not apply for (existing) reprogrammable network devices, such as P4-enabled hardware [12].

One needs to solve three main challenges to implement FAB in programmable data planes, namely, how to: *(i)* know the queue occupancies in the ingress pipeline; *(ii)* ensure packets are dropped exclusively in the ingress pipeline; and *(iii)* handle per-flow state across the device. In the following, we describe how to address each of these challenges.

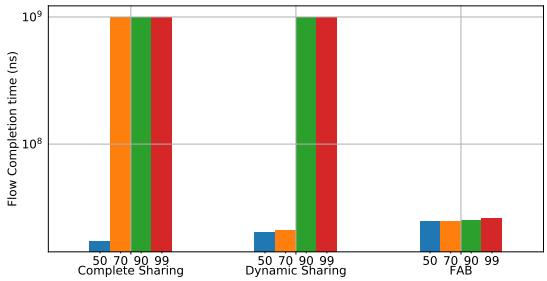
Approximating queue occupancies. A device-wide, flow-aware buffer management requires queue occupancy information to be available at the ingress pipeline (i.e., before the packets hit the traffic manager). However, existing programmable devices usually know the current queue length only in the egress pipeline, that is, after buffering has been made [13]. We propose to approximately calculate the queue occupancy in the ingress by keeping counters per port. Counters are increased as packets arrive, and decreased periodically depending on the known dequeue rate. In practice, this is challenging as the same memory block cannot be accessed twice [11] once for increasing and once for decreasing the counter. To overcome this limitation, we can split queue counters into two memory blocks and, thus, two stages. In this way, upon arrival of a packet, the switch will increase the counter that corresponds to its output port by the packet's



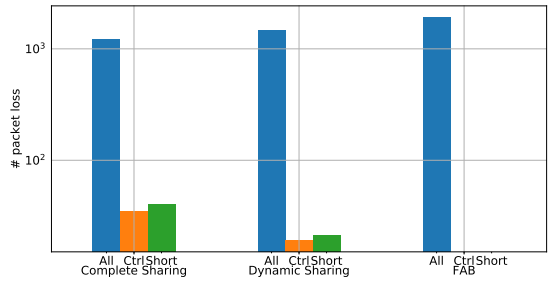
(a) FAB is equivalent to Complete sharing if there are no long flows.



(b) Dynamic sharing drops packet of small flows although there is space in the buffer.



(c) FAB gives lower FCT in higher percentiles and higher for median.



(d) FAB drops roughly the same number of packets but exclusively from long flows.

Figure 3: Simulation results for short-only flows (3a,3b) and mixed short & long flows (3c,3d)

length and decrease the counter of another queue by an estimated number of bytes that should have been dequeued since the counter was last decreased.

Maintaining per-flow state at scale. To distinguish flows based on their size, the switch would need to store per-flow counters, *i.e.*, per-flow state. Doing so could quickly exhaust the limited memory resources of a programmable device. To address this challenge, we propose two approximations. First, we can approximate flow sizes with flow duration. Indeed, short flows in size will most likely be short in duration. Thus, instead of prioritizing short flows, we prioritize those that have more recently started. Second, to avoid keeping the starting timestamp for each flow, we propose to split time into time windows and store an identifier of the time window at which a flow started. Flows that start in the same window can be easily stored in a Bloom filter corresponding to this window. As such, bloom filters corresponding to older flows can be reset to make up space for new ones, while the flows that are no longer contained in a BF are treated as old.

5 RELATED WORK

Buffer occupancy is affected by algorithms at the port-level (*e.g.*, queue management, scheduling), device-level (*e.g.*, buffer sharing), or host-level (*e.g.*, TCP). These algorithms are often complementary. In this section, we provide an overview of these approaches.

Port-level Buffer Management. Active Queue Management techniques avoid bufferbloat and flow synchronization by controlling the buffer allocation per port, but are oblivious to the overall buffer utilization and practically unable to increase the buffer allocated to a port according to the device-level utilization. Initial proposals of AQM systems were oblivious to the queue content and only considered the queue length or the queue delay. For instance, RED [19] uses the queue length to decide a dropping rate, while Codel [23] distinguishes bad queues using the minimum queue delay per predefined period. Similarly, PIE [27] uses a proportional-integral controller to limit the queuing delay by updating the drop probability per queue but without parametrization. Randomness in dropping packets as well as lack of visibility of the queue content make such approaches prone to punishing "innocent" flows and benefiting unresponsive ones.

More recent AQM systems and patents apply fair queuing and per-flow buffer limits to protect the buffer from aggressive flows. For instance, FQ_codel [22] allocates a queue per flow, which is serviced in a round-robin fashion to prevent head of line blocking. Similarly, the Dynamic Buffer Limiting [10, 18] algorithm protects the buffer allocated to each port using the per-flow buffer occupancy to distinguish non-responsive flows. Finally, [25] moves long flows to a different queue with different limits. While extremely useful, all these techniques cannot change the buffer that is allocated to a port or queue at the device-level, even if the queue is indeed well-behaving and can benefit from extra free space in the device.

Device-level Buffer Management. The classical dynamic thresholds described in [26] and [15] suggest the use of the remaining buffer to set the limits per port. The main issue with this approach is that it applies the same limits to same-length queues of different natures, *e.g.,* a queue containing an aggressive flow and a queue containing a bursty one. Indeed, one could use different parameters per priority, but that could only work if one fully controls end hosts which should tag their flows correctly and respond to congestion notifications. Both requirements are hard to be met in cloud environments. Instead of the remaining buffer, authors in [29] suggest the use of the overall cell arrival rate to gauge queue limits, if the buffer occupancy is above a predefined threshold and adapts it to fit the actual queue size. Finally, [28] leverages packet size to decide whether to drop a packet. None of these approaches change the buffer per port using flow information.

Scheduling. Similar to scheduling techniques such as pFbric [6] and PIAS [9], FAB's objective is to achieve low flow completion times by prioritizing certain (*e.g.,* short) flows. However, scheduling approaches are orthogonal to device-level buffering as they focus on resolving conflicts among flows in the same port, not the same device. Scheduling cannot help when a burst of high priority flows arrive and is larger than the pre-allocated buffer space. Still, special scheduling can and should augment buffer management techniques like FAB to control the queuing delay per port.

Cisco Intelligent Buffering. Cisco Intelligent Buffering [1, 2] is a combination of a flow classifier, an active queue management scheme, and a scheduling technique. Yet, the buffer allocation per port is static. In particular, the flow classifier (Elephant Trap) distinguishes long flows, based on the number of packets a flow sends and a user-defined threshold. An AQM scheme, namely Approximate Fair Drop (AFD), calculates and actuates a dropping rate per flow for all the elephant flows such that each takes its fair share of the bandwidth. In essence, AFD is a flow-aware replacement for WRED. Finally, to avoid increased delay by mixing elephant with mice flows, a scheduling scheme called Dynamic Packet Prioritization (DPP) is used. DPP offers a fast lane to the mice flows. Intelligent Buffering does not change the buffer allocation itself but only better manages the already allocated space per port. Additionally, this solution requires to keep state per flow as the classification is done based on the byte counts of incoming flows for all flows, which translates to a large amount of memory.

Programmable Devices. Snapy [13] can detect bursts using programmable data-planes. Conquests [14] is a practical data-plane technique that for each dequeued packet provides the number of packets of the same flow that were enqueued after it. This information, though, is only available at the egress, namely after the decision to accept it has been taken.

DIBS [31] is a just-in-time congestion mitigation for data centers with a similar goal to FAB, namely avoiding packet drops of short flows. Instead of using the shared buffer, though, DIBS detours excessive packets to neighbor devices.

TCP versions, such as DCTCP [5] require end-host support and are not useful for short flows as they have no time to react to ECN marking. Yet, ECN schemes can augment FAB, allowing to achieve better throughput with smaller buffers for long flows.

6 CONCLUSION

FAB is a new device-wide and flow-aware buffer management scheme that aspires to ripe the benefits of the shared buffers in switches. By applying well-known techniques for prioritizing flows, from port-level to the device-level, FAB solves the short-comings of conventional buffer management techniques. Preliminary results on FAB's performance show an order of magnitude shorter tail flow completion times for short flows in specific workloads. Implementing FAB in programmable hardware would also facilitate new buffer management schemes that are tailored to specific workloads.

REFERENCES

[1] [n. d.]. Cisco Nexus 9000 Series Switches. https://www.cisco.com/c/en/us/products/collateral/switches/nexus-9000-series-switches/white-paper-c11-738488.html. ([n. d.]).

[2] [n. d.]. Cisco Nexus 9000 Series Switches. https://www.cisco.com/c/en/us/products/collateral/switches/nexus-9000-series-switches/white-paper-c11-739134.html. ([n. d.]).

[3] [n. d.]. NS3 Network Simulator. https://www.nsnam.org/. ([n. d.]).

[4] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Francis Matus, Rong Pan, Navindra Yadav, George Varghese, et al. 2014. CONGA: Distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM Computer Communication Review*, Vol. 44. ACM, 503–514.

[5] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2011. Data center tcp (dctcp). *ACM SIGCOMM computer communication review* 41, 4 (2011), 63–74.

[6] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pfabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM Computer Communication Review*, Vol. 43. ACM, 435–446.

[7] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. 2004. *Sizing router buffers*. Vol. 34. ACM.

[8] M. Arpaci and J. A. Copeland. 2000. Buffer management for shared-memory ATM switches. *IEEE Communications Surveys Tutorials* 3, 1 (First 2000), 2–10. https://doi.org/10.1109/COMST.2000.5340716

[9] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Weicheng Sun. 2014. PIAS: Practical information-agnostic flow scheduling for data center networks. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*. ACM, 25.

[10] Andreas V Bechtolsheim and David R Cheriton. 2003. Per-flow dynamic buffer management. (Feb. 4 2003). US Patent 6,515,963.

[11] Ran Ben-Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. 2018. Efficient Measurement on Programmable Switches Using Probabilistic Recirculation. *2018 IEEE 26th International Conference on Network Protocols (ICNP)* (Sep 2018). https://doi.org/10.1109/icnp.2018.00047

[12] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM CCR'14* (2014).

[13] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, and Ori Rottenstreich. 2018. Catching the microburst culprits with snappy. In *Proceedings of the Afternoon Workshop on Self-Driving Networks*. ACM, 22–28.

[14] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, and Tzuu-Yi Wang. [n. d.]. Fine-Grained eue Measurement in the Data Plane. ([n. d.]).

[15] Abhijit K Choudhury and Ellen L Hahne. 1998. Dynamic queue length thresholds for shared-memory packet switches. *IEEE/ACM Transactions On Networking* 6, 2 (1998), 130–140.

[16] Sujal Das and Rochan Sankar. 2012. Broadcom smart-buffer technology in data center switches for cost-effective performance scaling of cloud applications. *Broadcom White Paper* (2012).

[17] Amogh Dhamdhere and Constantine Dovrolis. 2006. Open issues in router buffer sizing. *ACM SIGCOMM Computer Communication Review* 36, 1 (2006), 87–92.

[18] F. Ertemalp. 2001. Using Dynamic Buffer Limiting to Protect Against Belligerent Flows in High-Speed Networks. In *Proceedings of the Ninth International Conference on Network Protocols (ICNP '01)*. IEEE Computer Society, Washington, DC, USA, 230–. http://dl.acm.org/citation.cfm?id=876907.881596

[19] S. Floyd and V. Jacobson. 1993. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking* 1, 4 (Aug 1993), 397–413. https://doi.org/10.1109/90.251892

[20] Yashar Ganjali and Nick McKeown. 2006. Update on buffer sizing in internet routers. *ACM SIGCOMM Computer Communication Review* 36, 5 (2006), 67–70.

[21] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: a scalable and flexible data center network. In *ACM SIGCOMM computer communication review*, Vol. 39. ACM, 51–62.

[22] Toke Hoeiland-Joergensen, Paul McKenney, Dave Taht, Jim Gettys, and Eric Dumazet. 2016. The flowqueue-codel packet scheduler and active queue management algorithm. *IETF Draft, March* 18 (2016).

[23] V Jacobson and N Kathleen. 2012. Controlling Queue Delay-A modern AQM is just one piece of the solution to bufferbloat. *Asscociation for Computing Machinery (ACM Queue)* (2012).

[24] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be general and fast. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 1–16.

[25] Aisha Mushtaq, Asad Khalid Ismail, Abdul Wasay, Bilal Mahmood, Ihsan Ayyub Qazi, and Zartash Afzal Uzmi. 2014. Rethinking Buffer Management in Data Center Networks. *SIGCOMM Comput. Commun. Rev.* 44, 4 (Aug. 2014), 575–576. https://doi.org/10.1145/2740070.2631462

[26] Eugene Opsasnick. [n. d.]. Buffer management and flow control mechanism including packet-based dynamic thresholding. *US patent US7953002B2* ([n. d.]). https://patents.google.com/patent/US7953002B2/en

[27] R. Pan, P. Natarajan, C. Piglione, M. S. Prabhu, V. Subramanian, F. Baker, and B. VerSteeg. 2013. PIE: A lightweight control scheme to address the bufferbloat problem. In *2013 IEEE 14th International Conference on High Performance Switching and Routing (HPSR)*. 148–155. https://doi.org/10.1109/HPSR.2013.6602305

[28] V. Rajan and Yul Chu. 2005. An enhanced dynamic packet buffer management. In *10th IEEE Symposium on Computers and Communications (ISCC'05)*. 869–874. https://doi.org/10.1109/ISCC.2005.27

[29] Ruixue Fan, A. Ishii, B. Mark, G. Ramamurthy, and Qiang Ren. 1999. An optimal buffer management scheme with dynamic thresholds. In *Seamless Interconnection for Universal Services. Global Telecommunications Conference. GLOBECOM'99. (Cat. No.99CH37042)*, Vol. 1B. 631–637 vol. 1b. https://doi.org/10.1109/GLOCOM.1999.830130

[30] Selma Yilmaz and Ibrahim Matta. 2001. On Class-based Isolation of UDP, Short-lived and Long-lived TCP Flows. 415–422. https://doi.org/10.1109/MASCOT.2001.948894

[31] Kyriakos Zarifis, Rui Miao, Matt Calder, Ethan Katz-Bassett, Minlan Yu, and Jitendra Padhye. 2014. DIBS: Just-in-time congestion mitigation for data centers. In *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 6.