FB: A Flexible Buffer Management Scheme for Data Center Switches

Maria Apostolaki  Vamsi Addanki  Manya Ghobadi  Laurent Vanbaver

Abstract

Today’s network devices share buffer across queues to avoid drops during transient congestion. While cost-effective most of the time, this sharing can cause undesired interference among seemingly independent traffic. As an intuition, unresponsive or malicious flows on uplink low-priority queues can deteriorate the performance of a downlink high-priority queue. The cause of this perhaps unintuitive outcome is that today’s buffer sharing techniques are unable to prevent excessive buffer utilization i.e., buffer pressure, or react to abrupt changes in the traffic demand. To address this issue, we designed FB, a novel buffer sharing scheme that offers minimum-buffer and burst-tolerance guarantees without sacrificing throughput. Importantly, FB does not require modifications to the TCP/IP stacks or network switches. First, we precisely approximated FB in P4 and implemented it on a Intel Barefoot Tofino. Second, we show how any commodity device (e.g., using Broadcom Trident*, Nexus devices, etc.) can approximate FB’s behavior. Finally, we show that FB achieves up to 121x better burst absorption, while not sacrificing throughput and tail Flow Completion Time.

1 Introduction

To reduce cost and maximize utilization, the on-chip buffer of a network device is shared across its queues. A buffer management algorithm [3, 31, 47] is then responsible for dynamically splitting buffer space across queues. The most commonly-used algorithm today is Dynamic Thresholds (DT) [6, 30, 31, 38, 55]. DT allocates buffer proportionally to the unoccupied buffer space across queues. The most commonly-used algorithm today is Dynamic Thresholds (DT) [6, 30, 31, 38, 55]. DT allocates buffer proportionally to the unoccupied buffer space across queues. The most commonly-used algorithm today is Dynamic Thresholds (DT) [6, 30, 31, 38, 55]. DT allocates buffer proportionally to the unoccupied buffer space across queues. The most commonly-used algorithm today is Dynamic Thresholds (DT) [6, 30, 31, 38, 55]. DT allocates buffer proportionally to the unoccupied buffer space across queues. The most commonly-used algorithm today is Dynamic Thresholds (DT) [6, 30, 31, 38, 55]. DT allocates buffer proportionally to the unoccupied buffer space across queues. The most commonly-used algorithm today is Dynamic Thresholds (DT) [6, 30, 31, 38, 55]. DT allocates buffer proportionally to the unoccupied buffer space across queues.

Despite its wide deployment, DT does not meet the performance requirements of today’s multi-tenant datacenters for two key reasons. First, DT is unable to prevent excessive buffer utilization (a phenomenon also known as buffer pressure [14]). As a result, the performance of (even high priority) traffic is dependent on the instantaneous load on all queues sharing the same buffer chip. For instance, unresponsive or malicious flows on uplink low-priority queues can deteriorate the performance of a downlink high-priority queue. Second, DT is unable to react to abrupt changes in the traffic demand, as it incautiously keeps the buffer highly utilized. As a result, DT cannot reliably absorb bursts, which are of paramount importance for application performance [28, 59].

To limit DT’s shortcomings, datacenter operators and researchers have come up with three strategies: (i) splitting the buffer space to private and shared regions i.e., statically assign a part of the buffer to the queues and allow the rest to be dynamically shared; (ii) use ECN-based transport schemes that react to congestion signals; and (iii) use bufferless transport schemes. Unfortunately, none of these approaches can practically solve DT’s shortcomings for today’s multi-tenant datacenters.

First, while statically allocating buffers protects queues from starvation, it practically keeps precious buffer idle most of the time. This strategy becomes even less effective as on-chip buffers are becoming disproportionately smaller relatively to link speeds [21]. The drop in buffer size per Gbps is so dramatic that can prevent full throughput [21]. Second, while ECN-based transport schemes [14, 52, 66, 69], decrease buffer utilization, effectively leaving more space for bursts, they are unable to operate with tiny buffers [21] or unresponsive flows which occur in multi-tenant datacenters [54]. Third, while relying on (almost) bufferless transport schemes [29, 37, 51, 58] alleviates the need for buffer, they (i) require non-trivial modifications to switch hardware and/or network stacks [21], neither of which might be practical for multi-tenant datacenters [46] or (ii) assume non-oversubscribed networks with perfect load balancing [37].

In this paper, we aim at developing an approach that offers provable burst-tolerance and minimum-allocation guarantees without requiring any modification to the host network stacks, to the switch hardware or architecture. To that end, we present Flexible Buffer (FB) a practical buffer management scheme. Unlike existing buffer management schemes that operate in a best-effort manner, FB’s operations lead to predictable ranges of buffer occupancies independently of the incoming traffic. FB achieves this by considering both the content and the...
temporal characteristics of the occupied buffer space to decide on the possible future buffer allocation. In particular, FB (i) prevents any set of queues from monopolizing the buffer by guarding the distribution of buffer usage across multiple dimensions (e.g., per priority); and (ii) makes the buffer available to more traffic by favoring queues that free-up their used buffer faster. Unlike bufferless and ECN-enabled transport schemes, FB does not require privileged access to the chip or control of the end-hosts. Instead As a proof-of-concept, we show that a network operator can approximate FB’s buffer allocations by implementing it on a Barefoot Tofino Wedge 100BF-32X [12] or directly use its logic to guide DT’s periodic reconfiguration. DT is supported by all commodity devices e.g., those with Broadcom chips, or Nexus device etc.

Our evaluation shows that FB’s benefits in burst absorption (measured as the percentage of queries that did not experience drops) over practical configurations of DT varies from 1.58x for relatively smaller bursts up to 121x for large ones. We demonstrate that while ECN-based schemes such as DCTCP can rapidly improve the performance of DT, it only approaches FB’s performance for small bursts. Importantly, FB achieves superior burst absorption without sacrificing throughput or Flow Completion Time (FCT) of short flows. We show that FB can be implemented on existing devices by periodically re-configuring DT’s key parameter at coarse-grained intervals (every 1 sec) to reap FB’s benefits with minor loss of performance. Finally, we show that FB is useful in various deployment scenarios and compatible with in-network traffic classification.

Our main contributions include:

• The first analysis of DT (the most widely-used buffer management algorithm today) in a multi-queue setting. Our analysis reveals DT’s limitations both analytically (§2) and experimentally (§6).

• A novel approach for buffer management that can provide performance guarantees without statically allocating buffer (§4).

• A novel hardware design and implementation of an approximation of FB on a Barefoot Tofino switch [12] that demonstrates its practicality (§5.1), together with another approximation of its behavior that extends its deployability to more-commonly-used devices (§5.2).

• A comprehensive evaluation demonstrating that FB outperforms state-of-the-art buffer management algorithms even when combined with DCTCP (§6).

2 Background & Motivation

In this section, we first describe our model, namely the network device architecture we consider (§2.1). Next, we explain how Dynamic Thresholds (DT), the most commonly-used buffer management algorithm, works (§2.2). Finally, we reveal DT’s core limitations (§2.3).

2.1 Switch Model

Consider an output-queued shared-memory packet switching chip depicted in Fig. 1. Its Memory Management Unit (MMU) selectively enqueues packets to an output queue and buffers them on-chip until they can be dequeued. The MMU buffers an incoming packet if the queue length it is mapped to is shorter than the queue’s dynamically calculated threshold. Thresholds are calculated according to the buffer management algorithm such as DT (§2.2). Note that the MMU cannot buffer every incoming packet as buffer space is limited and buffered packets cannot be removed [21].

We assume that traffic is distinguished in (multiple) high and low priority classes to allow for bandwidth isolation. Each class uses a single queue at each port [8]. Priorities allow operators to prioritise certain classes in times of high load (e.g., traffic subject to Service Level Agreements (SLAs)). In Fig. 1 the MapReduce class is of high priority, while all other classes are of low priority. Note that this prioritization concerns the use of the shared buffer and does not affect scheduling. We assume weighted round robin scheduling.

While round robin scheduling, grouping traffic into high or low priority classes and the existence of multiple queues are common in datacenters, they are not hard requirements to use FB (see §4.1).

2.2 DT’s workings

The Dynamic Thresholds (DT) is the most common buffer management algorithm in today’s devices [6, 16, 38, 49, 55]. DT [30] dynamically adapts the instantaneous maximum length of each queue, namely its threshold \( T_c(t) \) according to the remaining buffer and a configurable parameter \( \alpha \), as we see in Eq.(1). DT’s per-queue threshold is directly proportional to the remaining buffer \( (B - Q_c(t)) \) and to a parameter \( \alpha \) often configured per class.\(^2\)

\[
T_c(t) = \alpha_c (B - Q_c(t))
\]  

\(^1\)We describe the mapping of our model to RMT architecture in §5.  
\(^2\)While \( \alpha \) can be configured per queue, it is often configured per class.
Single queue in the buffer
Another queue needs buffer
New packets to q1 are dropped to allow...t2
q2 is dropped
Threshold q1 equals q1 length
Threshold q2 below q1 length
q1 in the buffer

\[ \alpha \]
(resp. lower)

\[ q \]
Transient state is the state during which at least one queue in the buffer is the low-priority yellow class: q1 of colored in red. The operator has configured the solution of the length of two queues and their thresholds over one of high and one of low priority. Fig. 2 illustrates the evolution of traffic classes. Despite its importance, there is no systematic way to configure the two classes. Different vendors and operators reportedly use different \( \alpha \) values. For instance, Yahoo uses \( \alpha = 8 \) [38] while Cisco uses \( \alpha = 14 \) [6] and Arista \( \alpha = 1 \).

The buffer alternates between steady and transient states. Steady state is the state during which all queues sharing the buffer are shorter or equal to the threshold that DT calculates. Transient state is the state during which at least one queue in the buffer is longer than its threshold.

**DT in action.** To understand how the buffer alternates between steady and transient states, we walk through an example scenario. Consider a switch with a 60-packet buffer shared across multiple queues mapped to two distinct traffic classes: one of high and one of low priority. Fig. 2 illustrates the evolution of the length of two queues and their thresholds over time, illustrated in solid and dotted lines, respectively. Queue q1 belongs to the low-priority class and is colored in yellow. The other queue q2 belongs to the high-priority class and is colored in red. The operator has configured \( \alpha = 1 \) for the class of q1 and \( \alpha = 2 \) for the class of q2.

Before time \( t_0 \), the only congested (non-empty) queue in the buffer is the low-priority yellow class: q1. During this time, the buffer is in steady-state, meaning all queues’ length is lower or equal to DT’s thresholds. Indeed, q1’s length is 30 packets, the remaining buffer space is equal to 30 packets; thus from Eq. (1) q1’s threshold is \( 1 \times 30 = 30 \). While q1 might experience drops while the buffer is in steady-state, these are not critical as the queue has already occupied its fair share of the buffer.

At time \( t_0 \), a burst of packets belonging to the high-priority class arrives at the switch. The burst causes q2’s length to increase in the time frame \( (t_0-t_2) \) (solid red line).

At time \( t_1 \), q2’s length increase is inhibited: q2 continues to grow but at a lower rate as its length starts being controlled by q2’s threshold (red dashed line), which DT calculates. q1’s threshold decreases due to q2’s growth in the buffer, which reduces the overall remaining buffer. Thus, during the time frame \( (t_1-t_2) \), some of the packets mapped to q2 are dropped. These drops are more critical compared to the steady-state drops of q1 as they occur before q2 occupies its fair-share of the buffer. In the time frame \( (t_0-t_1) \), the reduction of the remaining buffer also causes q1’s threshold to decrease. Notably, q1’s threshold decreases at a rate higher than its length does. In the time frame \( (t_0-t_2) \), the buffer is in transient-state, as q1’s length is higher than its threshold.

At \( t_2 \), the buffer reaches steady-state again. This time, the remaining buffer is 15 packets, q1 occupies 15 packets, resulting from q1’s threshold \( T_{q1} = 1 \times 15 \) packets, while q2 occupies 30, as \( T_{q2} = 2 \times 15 \).

**To sum up,** the high-priority burst was dropped before the buffer had reached steady-state. These drops are critical as they could have been avoided if (i) there was more available buffer when the burst arrived (steady-state allocation); or (ii) the buffer could have been emptied faster to make room for the burst (transient-state allocation).

### 2.3 DT’s limitations

DT is fundamentally unable to reliably control the buffer occupancy during steady- or transient-state. As a result, DT cannot offer any minimum-buffer guarantee nor any burst-tolerance guarantee. The former is only affected by DT’s steady-state allocation, while the latter by both steady- and transient-state allocation, as we explained in §2.2. Next, we illustrate these limitations via comprehensive examples. To the best of our knowledge we are the first to uncover these limitations and justify them analytically.

**DT offers no minimum buffer guarantee.** DT allows the operator to preferentially treat a queue or class over the others via the static configuration of the parameter \( \alpha \) for example, \( \alpha_{q1} \) for a high priority class and \( \alpha_{q2} \) for a low-priority class. Yet, \( \alpha \) offers no guarantee as the actual per-queue threshold depends on the overall remaining buffer (Eq. 1), which can reach arbitrarily and uncontrollably low values even in the steady-state.

\[ 4 \]
An operator can statically allocate space for each queue; in this case, though, the buffer space will be wasted unless used by the corresponding queue.

---

\[ 1 \] For simplicity, we assume a single buffer-chip per device.
The queue of port 1 belongs to a high-priority class and is colored in red. The three other queues, mapped to ports 2-4, belong to a low-priority class and are colored in yellow. Notably, all queues are mapped to a single port (port 2). DT allows each queue to occupy 10 packets, as they are configured with $\alpha = 1$ and the remaining buffer space in the steady-state is 10. In Fig. 4c, a high-priority 5 : 1 incast occurs at port 1, meaning 5 incoming ports simultaneously send to port 1. Due to DT’s prior allocations, though, the buffer cannot keep up with the incoming traffic rate. Concretely, the buffer has not enough unoccupied space, nor can it be emptied fast enough to make room for the burst. As a result, the high-priority burst starts to experience drops while it only occupies 8 packets in the buffer. Observe that the high-priority queue’s steady-state allocation (fair share) would be $\approx 17$ packets. Thus, the high-priority queue is experiencing drops in the transient-state, which could have been avoided if the buffer could reach steady-state faster. The reason of this slowdown is that the 5 low-priority queues share the dequeue rate of a single port. Notably, DT has no way to distinguish between 5 queues attached in a single port from 5 queues attached to separate ports.

DT offers no burst-tolerance guarantees. In addition to the unpredictability of its steady-state allocation, DT’s transient-state allocation is uncontrollable. Thus, its burst absorption capabilities for any class or priority depends on the instantaneous length of all congested queues. The main reason for this limitation is that DT perceives buffer space as a scalar quantity ignoring its expected occupancy over time.

To better understand this limitation, consider the same 60-packet buffer before and after the arrival of a burst shown in Fig. 4b and Fig. 4c, respectively.

In Fig. 4b, the buffer is shared across five congested queues of low-priority classes (all in yellow colors). Notably, all queues are mapped to a single port (port 2). DT allows each queue to occupy 10 packets, as they are configured with $\alpha = 1$ and the remaining buffer space in the steady-state is 10. In Fig. 4c, a high-priority 5 : 1 incast occurs at port 1, meaning 5 incoming ports simultaneously send to port 1. Due to DT’s prior allocations, though, the buffer cannot keep up with the incoming traffic rate. Concretely, the buffer has not enough unoccupied space, nor can it be emptied fast enough to make room for the burst. As a result, the high-priority burst starts to experience drops while it only occupies 8 packets in the buffer. Observe that the high-priority queue’s steady-state allocation (fair share) would be $\approx 17$ packets. Thus, the high-priority queue is experiencing drops in the transient-state, which could have been avoided if the buffer could reach steady-state faster. The reason of this slowdown is that the 5 low-priority queues share the dequeue rate of a single port. Notably, DT has no way to distinguish between 5 queues attached in a single port from 5 queues attached to separate ports.

3 Overview

In this section we describe FB and illustrate how FB surpasses DT’s limitations (§2.3) by showcasing its allocation in the same example scenarios we used in §2.3.

FB dynamically bounds the buffer allocation in steady-state. FB prevents traffic of any priority and class from monopolizing the buffer by dynamically bounding buffer usage. This bounding is not static, thus it differs from complete partitioning i.e., statically assigning space per queue and from application pools i.e., carving the buffer to sets of pools.

To illustrate the difference between the steady-state allocations of FB and DT we use Fig. 5a, which shows FB’s allocation under the same scenario we used for DT in Fig. 4a. Unlike DT, which decreases the buffer space occupied by high-priority traffic proportionately to the number of low-priority queues, FB bounds low-priority (yellow) queues to 15 packets on aggregate, equally shared across the three queues. As a result, the high-priority (red) queue can use 30 packets of buffer. In §4, we show more formally how FB makes the buffer that high-priority occupies independent of the number of congested low-priority queues. Importantly, no allocation is static: if the high-priority queue (red) does not use/need
we use Fig. 5b and Fig. 5c which show FB’s allocation before and after the arrival of a burst. We again consider the same example as in Fig. 4b and Fig. 4c for DT.

In Fig. 5b, FB detects that the aggregate dequeue rate is inevitably low (queues share a single port) and limits each low-priority queue to 6 packets, effectively leaving an incoming burst enough free buffer to be stored. As a result, when the 5:1 high-priority burst arrives (Fig. 5c), the buffer can reach steady state fast enough to avoid transient drops.

4 Design

Having showcased FB’s high-level properties through intuitive examples (§3), in this section we describe FB in detail. First, we describe FB’s threshold calculation considering both multiple and a single queue per port (§4.1). Next, we explain how the operator can configure FB to obtain performance guarantees (§4.2).

4.1 FB’s workings

FB on multiple queues per port: The formula for calculating the per-queue threshold considering multiple queues per port is given in Eq. 4. The threshold equals the product of: (i) an $\alpha$ value assigned by the operator to the class that the queue belongs to: $\alpha_c$; (ii) the inverse number of congested (i.e., non-empty) queues of the priority (low or high) that the class belongs to: $\frac{1}{n_{p(t)}}$; (iii) the per-port-normalized dequeue rate of this queue: $\gamma_q(t)$; and (iv) the remaining buffer space: $B - Q(t)$. Similar to DT, the operator only configures $\alpha_c$ for

its maximum buffer occupancy, the low-priority (yellow) will get more buffer.

**FB makes bursts first-class citizens in the buffer by minimizing transient-state drops.** FB is able to offer burst-tolerance guarantees by allocating buffer such that there is always a combination of (i) enough unoccupied buffer space; and (ii) adequately high aggregate dequeue rate (i.e., the buffer can be emptied fast enough) to accommodate a given burst. Both these factors are critical for a burst to be absorbed. On the one hand, an incoming burst can be absorbed independently of the free buffer space at its arrival, iff the aggregate dequeue rate of the allocated buffer space is at least as high as the enqueue rate. 6 On the other hand, an incoming burst can also be absorbed independently of the aggregated dequeue rate of the buffer at its arrival, iff the unoccupied buffer is sufficient to directly accommodate it. FB achieves a balance between the two extremes as we explain in §4, which allows it to achieve high throughput while providing burst-tolerance guarantees.

To illustrate the difference in FB’s allocation to that of DT we use Fig. 5b and Fig. 5c which show FB’s allocation before and after the arrival of a burst. We again consider the same example as in Fig. 4b and Fig. 4c for DT.

---

6 We are, of course, referring to a burst with a size smaller than the total buffer.
each class, while all other factors are known to the MMU at run time. Unlike DT, the $\alpha_c$ configuration has provable consequences to buffer allocation. Thus, the operator can configure $\alpha_c$ according to the guidelines we provide in §4.2 to obtain performance guarantees.

$$T_c(t) = \alpha_c \cdot \frac{1}{N_p(t)} \cdot \gamma_c(t) \cdot (B - Q(t))$$  \hspace{1cm} (4)

$N_p(t)$: number of congested queues of priority $p$ at time $t$

$B$: remaining buffer

$\gamma_c(t)$: per-port-normalized dequeue rate of $q_c$

**FB on a single queue per port:** While we have designed FB for a multiple-queue deployment, we can adapt Eq.4 for a deployment where only a single queue is available per port. To do so, $\gamma_c$ will always be 1 and $N_p(t)$ will correspond to all congested queues in the buffer. Thus, the threshold of a packet of class $c$ destined to port $i$ will depend on the $\alpha_c$, the total number of congested queues and the remaining buffer space. In essence, FB applies different thresholds for packets that are mapped to the same queue. If traffic is not classified/tagged FB can leverage information from a classifier in the device.

FB’s threshold calculation differs from that of DT (Eq. 1) by two factors: $N_p$; and $\gamma_c$. Next, we explain how each of those contributes to FB’s buffer allocation.

**$N_p(t)$ bounds steady-state allocation.** FB divides per-queue thresholds with $N_p$: the number of congested queues of the priority that the class belongs to, as seen in Eq.4. The consequence of this factor to FB’s allocation is twofold: (i) it bounds per-class and per-priority occupancy; and (ii) it allows weighted fairness across classes of the same priority.

First, dividing by $N_p$ prevents any single class, and any single priority from monopolizing the buffer. As more queues of the same class (or priority) use the buffer, the threshold of each of them decreases, effectively setting an upper bound to the per-class occupancy to $\frac{\alpha_c}{1 + \alpha_c}$ of the total buffer and an upper bound to the per-priority occupancy to $\frac{\alpha_c}{1 + \alpha_c}$ of the total buffer, where $\alpha_p$ is the highest alpha of the priority. As a result, the overall buffer occupancy $Q(t)$ is also upper-bounded, as shown in Eq.5 where $\alpha_H$ and $\alpha_L$ are the maximum $\alpha$ values of the classes of high ($H$) and low ($L$) priorities, respectively. We formally prove the buffer occupancy bound in Appendix A. Unlike DT, the maximum aggregate buffer allocation of FB is independent of the number of congested queues. Consequently, the minimum buffer available for a high-priority class is also independent of the number of queues or low-priority classes in the buffer and vice versa. Finally, the bound in Eq.5 implies that, in steady-state, there will always be a part of the buffer that is empty, effectively serving as headroom in the transient state. The part can be made arbitrarily smaller if needed by increasing $\alpha$ values.

Second, dividing by $N_p$ offers weighted fairness across classes of the same priority. Particularly, the buffer occupied by a priority is split into classes proportionately to their $\alpha$ values. As a result, if the operator wishes to favor a traffic class among those that belong to a single priority, she can do so by assigning a relatively higher $\alpha$ to this class.

$$Q(t) \leq B \cdot \frac{\alpha_L + \alpha_H}{1 + \alpha_L + \alpha_H}$$  \hspace{1cm} (5)

where $\alpha_L = \max_{c \in L}(\alpha_c)$ and $\alpha_H = \max_{c \in H}(\alpha_c)$.

$\gamma_c(t)$ reduces transient state’s duration. FB allocates buffer to each queue proportionately to its dequeuing rate ($\gamma$). The $\gamma$ factor, combined with the upper bounds, minimizes the duration of the transient state. Indeed, given some amount of buffer per priority, FB splits it into queues proportionately to their service rate, effectively minimizing the time it takes for the buffer to be emptied. In effect, FB reduces the time needed to transition from one steady-state allocation to another. Notably, this factor also contributes to increased TCP throughput, which depends on the capacity of the bottleneck link [32, 35, 48].

**4.2 FB guarantees**

Unlike DT, which works in a best-effort manner, FB allows the operator to obtain performance guarantees through appropriate $\alpha$ configuration. Notably, even with a generic configuration (i.e., not tailored to the network’s needs), FB outperforms DT as we show in §6.

**FB can guarantee high throughput for any priority.** Owing to the bounded allocation, FB can provably protect any priority from starvation. In effect, the operator can secure high throughput for low (or high) priority without statically allocating buffer. To do so, the operator would leverage her insights on the aggregate buffer requirements, say $B_p$, of the used congestion control algorithm (or just of the offered SLA), and configure $\alpha_L$ and $\alpha_H$ to satisfy Eq.6. Due to space constraints, we defer the formal derivation of Eq. 6 to Appendix A.

$$\alpha_L \geq \frac{B_p \cdot (1 + \alpha_H)}{B - B_p}$$  \hspace{1cm} (6)

For example, if the operator wishes to guarantee a minimum of 10% of the total buffer $B$ to the low priority and chooses $\alpha_H = 10$ for the high priority, then $\alpha_L$ must be set to 2.75 according to Eq. 6.

**FB can guarantee the absorption of a burst.** A burst is characterized by its incoming rate $r$ and duration $t$ [30]. Whether a burst will be absorbed depends on: (i) its incoming rate ($r$); (ii) the steady-state of the buffer at its arrival (steady-state); and (iii) the buffer’s ability to dequeue fast (transient state). FB provides two types of guarantees to adapt to the operator’s knowledge of the network requirements.

First, if the operator only knows the worst-case in-cast of high priority traffic, i.e., the rate at which hosts can send to a single queue, she can configure FB to guarantee that the
burst will occupy its fair steady-state buffer space before experiencing any drops. To that end, the operator only needs to configure the maximum \( \alpha \) of the low-priority classes according to Eq. 7, where \( BW \) is the port bandwidth assuming symmetric ports. \(^7\) For instance, if the operator wishes to guarantee burst absorption for arrival rates \( r < 10 \times BW \), then \( \alpha_L \) must be set to at most \( \frac{1}{8} \) according to Eq. 7.

\[
\alpha_L \leq \frac{BW}{r - 2 \cdot BW} \quad (7)
\]

Second, if the operator knows the worst-case incoming rate \( r \) and the duration \( t \) of the burst she wants to protect, she needs to configure \( \alpha_H \) and \( \alpha_L \) i.e., the maximum \( \alpha \) values of high and low priorities respectively, according to Eq. 8. Intuitively, FB cannot guarantee that for any burst. For instance, the buffer cannot absorb a burst that lasts arbitrarily long. Still, FB’s strategic steady and transient-state allocations allow the operator to guarantee that reasonably-sized burst can always be absorbed, while the buffer is not wasted waiting for that burst to arrive.

\[
\alpha_L \leq \frac{B}{(r - 2 \cdot BW) \cdot t} - 1; \quad \alpha_H > \frac{1}{\frac{B}{(r - BW) \cdot t} + \alpha_L} - \frac{\alpha_L}{r - BW} \quad (8)
\]

**Sketch of proof.** While we moved the full proof to [19], we include the key intuition below. The proof is centered around the time at which a hypothetical burst \((r, t)\) will begin to experience drops, say \( t_1 \), \( t_1 \) plays a critical role in burst absorption as \( t_1 \) must be greater than the duration \( t \) of a burst for the latter to be absorbed. We first express \( t_1 \) as a function of \( \alpha_L \), \( \alpha_H \) and the arrival rate \( r \) and derive an upper bound on \( \alpha_L \). Indeed, the use of \( N_p \) and \( \gamma_L \) in FB’s allocation eliminates the dependency on the state of the buffer. As a result, we are able to obtain guarantees.

As an example, say the operator specifies \( r = 10 \times 10 \text{Gbps} \) and \( t = 500 \mu s \) with overall buffer size of \( B = 9 \text{MB} \) and port bandwidth \( 10 \text{Gbps} \). In this case, the maximum \( \alpha_L \) must be less than 0.8 according to Eq. 8. Based on the chosen \( \alpha_L \), the operator can then compute \( \alpha_H \) according to Eq. 8 in order to guarantee burst absorption. Notice that by specifying the arrival rate and the duration of the burst, \( \alpha_L \) (< 0.8) can be better configured for the operators requirements on burst absorption including minimum buffer. When only the arrival rate \( r \) is known, \( \alpha_L \) (< 0.125) tends to be much lower in order to guarantee both burst absorption and minimum buffer.

5 **Hardware Design**

Having explained the benefits of FB, in this section, we demonstrate two ways of approximating FB in today’s devices. In particular, we explain how we can approximate FB on a protocol-independent switch architecture (PISA) switch,\(^8\) \((§ \text{5.1})\) and on a more-commonly-used device \((§ \text{5.2})\). The two approximations differ in accuracy, implementation complexity and applicability. In particular, the PISA approximation, namely PISA FB is more accurate than the commodity one as all operations are done in the data plane. Importantly, PISA FB works for both multi and single-queue scenarios. The commodity approximation, namely FBA (for Approximate FB) is significantly easier to implement as, unlike Tofino, the corresponding chips expose the required buffer metrics to the control plane \((\text{e.g., Broadcom}) [1], \text{Cisco}) [5], \text{Mellanox}) [4]\). Yet, FBA only works in the multi-queue scenario.

5.1 **FB on top of PISA**

In principle, FB should be easily implementable on the Traffic Manager (TM) of a PISA switch e.g., Intel Tofino. Indeed, the TM has access to queue lengths which essential all that is needed for calculating FB’s thresholds. In practice, implementing FB in the TM is not possible as the TM is, to-date, not programmable [64]. To circumvent this limitation we approximated FB in P4, therefore using the ingress and egress pipelines. This choice creates three challenges:

\textbf{C1} Deciding whether a packet should be buffered requires comparing the corresponding queue length with a threshold. Yet, queue lengths are only available in the egress pipeline; thus, after a packet has been buffered [64].

\textbf{C2} Calculating FB’s thresholds requires aggregated metrics over multiple queue lengths, e.g., remaining buffer. Yet, accessing multiple values of a single memory block is not possible in PISA switches [23, 27].

\textbf{C3} Calculating FB’s thresholds requires floating-point operations, which are not supported by PISA.

Next, we describe FB’s high-level PISA design and a packet’s journey before we describe how it addresses each of the aforementioned challenges.

**FB’s high-level design** We built FB upon six main components: four register arrays and two Match & Action (M&A) tables as shown in Fig. 6. We use the register arrays to keep the required state for deciding whether a packet should be
buffered or dropped. This decision needs to be taken in the ingress pipeline, i.e., before the TM accesses the packets. The aforementioned state includes the instantaneous remaining buffer (Remaining), the number of congested queues per port and priority (N_Port, N_Priority), as well as the queues’ length (Qlength). We use a M&A table (Routing) to map a packet based on its destination and priority tag to a port and queue for transmission as well as to multiple FB-specific fields. Finally, we use another M&A table (Shift) to approximate the required floating-point operations.

Packet’s journey Upon arrival, a packet’s destination and priority tag are matched against the Routing table to multiple action parameters: an α value and three indexes. These indexes are used to read the relevant information about the state of the buffer (e.g., corresponding queue length from the Qlength array or the number of congested queues of a specific priority from the N_Priority, etc.) (2). This information is used to find the required number of shifts (3) to apply to the remaining to calculate the threshold of the corresponding queue (4). If the threshold is higher than the corresponding queue’s length (5), the packet is enqueued (6). While being at the queue, the packet writes the length of its queue to the Qlength array in the egress (7).

C1: Queue lengths available to the ingress pipeline The length of any given queue is only available as a metadata field to packets that have been enqueued, thus while they traverse the egress pipeline. Yet, FB needs the queue lengths in the ingress. To address this, we transfer queue lengths from the egress to the ingress in two steps. First, we create a register array in the egress pipeline which stores the length of every queue in the device. Each index in the array corresponds to a queue, namely a pair of port and traffic class. Each packet traversing the egress pipeline triggers an update on the value corresponding to the length of the queue it belongs to. Second, we maintain a copy of this register array in the ingress to make it available to FB’s logic. To keep the copy up-to-date, we asynchronously generate specially crafted packets, namely SYNC packets. These packets read the queue lengths from the egress register, re-enter in the ingress pipeline via recirculation, and copy the read values to the ingress register array, as shown in Fig. 6. Due to the PISA constraints, which prevent accessing multiple values of a register array in a single pipeline pass, copying all values in one pipeline pass is not possible. Instead, each SYNC packet recirculates as many times as queues there are in a device.

Notably, there is, in practice, only marginal overhead from the SYNC packets. Concretely, the SYNC packets create at most <30Mbps even assuming maximum number of ports and priority queues in Tofino and an update interval of 1us. That is the case because SYNC packets are created by a traffic generator within the device and their number is static as it depends solely on the number of queues and update interval. Moreover, the new Tofino generation (Tofino2) provides egress queue information at the ingress pipeline, removing the burden of recirculation altogether. Finally, potential lag in the syncing will not significantly affect performance. In § 6.2, we show that FB is beneficial even considering an update interval of 1s.

C2: Calculating aggregated metrics Other than the length of the queue of interest, calculating FB’s thresholds requires visibility over: (i) its normalized dequeue rate; (ii) the number of congested queues of the same priority; and (iii) the remaining buffer space, as seen in Eq. 4. These metrics need to be dynamically calculated based on all queues’ instantaneous lengths. Doing so is challenging for three reasons. First, the dynamic calculation requires accessing multiple values in the same array e.g., the number of active queues per port. Second, it requires accessing selective indexes of the array, namely those corresponding to the subset of queues of interest e.g., number of controlled queues per priority. Third, the result of this calculation needs to be available in the ingress pipeline. We addressed these challenges again using SYNC packets, which read a subset of the indexes of the egress register array, recirculate and write the aggregated results in the ingress register arrays. In particular, we use three types of such packets. First, SYNC packets copy queue lengths from the egress to ingress (as described above) and update the Remaining register array as they anyway traverse all indexes. Second, SYNC packets count the congested queues per port, which is equivalent to the normalized dequeue rate per queue given the scheduling algorithm. Each SYNC packet updates a single port’s count with the number of queues above a threshold. Finally, SYNC packets count the congested queues per priority. All SYNC packets contain in their custom header the indexes from which they start and finish reading from Qlengths, the index at which they write and their pivot (indexes they skip).

C3: Approximating floating-point operation Even after having all required information available, FB needs to multiply the remaining buffer with other factors (i.e., the reverse of the number of congested queues per port and per priority and the α) to calculate the thresholds. Yet, PISA does not allow floating-point operations. To address this issue, we shift the remaining space value so many times as the logarithm of two of the product of all the factors mentioned above. The calculation of the number of required shifts is not done in the data-plane. Instead, we pre-calculate it for all possible values and store all the results using match-action rules which match on three values α, N_Port, and N_Priority. Observe that all three values are discrete and bounded, so the number of required rules is manageable. As an intuition, n is in the range of 2 – 8; there are only a couple of possible α (8 for Tofino), and a few tens of congested queues at most.

5.2 FB on top of DT (FBA)

We can approximate FB’s behavior (FBA) by periodically reconfiguring α per queue according to the instantaneous buffer occupancy and FB’s formula. Recall from Eq. 4 that FB’s formula deviates from DT’s α, say α_{dt}, such that
\[ \alpha_{\text{b}} = \alpha_c \cdot \frac{1}{N_p} \cdot \gamma^c(t) \]. Both the number of congested queues \( N_p \) and the normalized dequeue rate (\( \gamma \)) change over time; thus, we need to monitor both variables to calculate \( \alpha \). Fortunately, chip manufacturers (e.g., Broadcom) expose the queue lengths and the remaining buffer (at least as watermarks). Thus, we can build FBA by using a software controller that (i) periodically pulls the queue lengths and the remaining buffer; (ii) calculates the number of congested queues; and (iii) infers the per-queue dequeue rate considering the scheduling algorithm per port and the number of active queues. The periodicity of the \( \alpha \) updates depends on the capabilities of the device in terms of monitoring queues and updating \( \alpha \). In §6, we consider the update period of 1 s which already brings considerable benefits. Intuitively, as we increase the frequency of the updates, FBA will approach the performance of FB.

Despite its benefits, FBA cannot approximate FB if only a single queue is available per port. As FBA relies on DT it cannot use different thresholds for packets of the same queue. In this case, FBA will have the same performance as DT. Still, provided that traffic is stable, FB’s insights can be used to configure static \( \alpha \).

6 Evaluation

We evaluate FB in simulation aiming at answering four main questions:

**Q1** Is FB able to absorb large burst even with non-optimal \( \alpha \) configuration?

**Q2** Can today’s hardware chips approximate FB’s allocation to reap the benefits of FB?

**Q3** Can ECN-based congestion schemes (such as DCTCP) alone alleviate the problem of buffer pressure?

**Q4** Does FB sacrifice throughput or over-penalize tail FCT of short flows in favor of bursts?

We demonstrate that: (Q1) FB outperforms all buffer management schemes in terms of burst absorption even with identical \( \alpha \) configuration. The improvement in burst absorption capabilities (measured as the percentage of queries that did not experience drops) increases as the burst size increases and ranges between 1.58x and 121.2x. (Q2) FBA (i.e., FB’s approximation which is deployable to all commodity devices) precisely approximates FB’s allocations achieving on-par performance with FB in practice. (Q3) DCTCP significantly improves the performance of DT, but its effectiveness decreases as the burst sizes increase. (Q4) FB does not deteriorate throughput even when paired with TCP for various loads. FB cannot minimize tail FCT without DCTCP, yet it still performs 34.1% better than DT with TCP.

Next, we elaborate on our methodology (§6.1) and we describe our detailed results (§6.2).

6.1 Methodology

We elaborate on the topology, traffic mix, priority assignment, and deployment scenarios we used.

**Topology.** We evaluate FB’s performance in a Leaf-Spine topology [13] with two leaves, two spines, and four links connecting each pair of leaf and spine. We use per-flow ECMP to load-balance traffic across uplinks. Each switch has 48 ports of 10Gbps bandwidth and a buffer of size to 9MB to emulate the Broadcom Trident+ switch [9]. Each leaf switch is connected to 40 servers. We set the propagation delay on all the links to 25\( \mu \)s, which results in a minimum RTT of 200\( \mu \)s and a bandwidth-delay product of 250KB. Our results are not sensitive to analogous increases of bandwidth and buffer size.

Next, we elaborate on our methodology (§6.1) and we describe our detailed results (§6.2).

**Deployment Scenarios.** We compare multiple buffer management schemes considering two deployment scenarios: multi-queue and single-queue, which differ on the number of priority queues and the marking scheme.

In the multi-queue scenario, we assume five queues per port and Round-Robin scheduling. Incoming packets are mapped to a queue according to the tag they carry. The query traffic is marked with a high-priority tag. In effect, packets belonging to a burst carry a high-priority tag and are mapped to a separated priority queue in the multi-queue deployment scenario. The web-search traffic is marked with four equally-low-priority tags. Tags are added by the servers uniformly at random.

In the multi-queue scenario, we compare FB with five other
buffer management schemes: (i) Dynamic Thresholds (DT) configured according to [7] i.e., with a private part which is statically-allocated and a public part which is allocated according to DT § 2.1; (ii) DT with DCTCP; (iii) Cisco Intelligent Buffer (IB) [2] which combines DT, approximate fair dropping mechanism [57], and an additional priority queue for short flows; (iv) FB’s approximation (FBA) as we described in § 5.2; and (v) FBA with DCTCP. We run DCTCP (instead of other newer schemes e.g., DCQCN [71], TIMELY [50] or HPCC [50]) for three main reasons: (i) DCTCP has seen the most deployment to-date; (ii) DCTCP does not require changes in the TCP/IP stack (kernel bypass); and (iii) DCTCP would still be superior to newer schemes (e.g., DCQCN and TIMELY) if we were to consider a kernel bypass setting [45]. As DCTCP requires control over the end-hosts we run it on the buffer management schemes that require no change in the network device i.e., DT and FBA. Finally, we stress that FB is compatible with any congestion control scheme.

In the single-queue deployment scenario, we assume a single queue per port. Traffic is not marked with priority tags by end-hosts, but a flow classifier is available to all buffer management schemes. The availability of the classifier allows us to compare against the recently proposed algorithm FAB [16]. Thus, we also compare against FAB [16] which requires a classifier and cannot use the pre-marked traffic. In this scenario, we do not evaluate Cisco Intelligent Buffer (IB) as it requires an additional priority queue or to FBA as FB cannot be approximated with DT on a single queue as we explain in § 5.2.

Configuration. We configure DT according to [7,11] i.e., statically allocating 18.8% as private space. The private space is equally distributed across all queues. The remaining space is dynamically shared according to DT. This configuration, prevents queues from starvation. In the single-queue scenario, we set DT’s $\alpha = 0.5$. In the multi-queue scenario, we set $\alpha_l = 0.5$ and $\alpha_h = 20$ for low-priority and high-priority traffic, respectively. We configure FB with $\alpha_l = 0.5$ for low-priority (or long flows) and and $\alpha_h = 20$ for high-priority traffic (or short flows) respectively in the single-queue scenario. We configure IB with $\alpha = 0.5$. IB uses headroom for short flows. IB also uses Dynamic Packet Prioritization [2] which prioritizes short flows of all classes to a separate high priority queue. When DCTCP is used, all queues are RED [34] with min and max thresholds set to 65 ($K_{min} = K_{max} = 65$) following the recommendations in [14]. When TCP is used, all queues are DropTail, except for IB. TCP minRTO is set to 100ms.

We use the ns-3 Simulator, version 3.33.7 We perform 10 experiments and report average values.

6.2 Results

FB/FBA have better burst absorption capabilities compared to all alternatives when bursts are >60% of the buffer. Fig. 7a, Fig. 7b illustrate the burst absorption capability of the buffer as a function of the total burst size for the multi- and single-queue deployment, respectively. The burst absorption is measured as the percentage of queries that did not experience drops similar to [14]. The background load is fixed at 90%, while the total burst size varies from 0.2 to 0.9 of the total buffer size. We find that FB improves DT’s burst absorption by 1.58x-2.5x in the multi-queue deployment and by 3.64x-121.2x in the single-queue deployment. As an intuition, when FB was managing the buffer 90% of the bursts that arrived did not experience any drop (i.e., were fully absorbed by the buffer), while when DT was managing the

---

8Setting $\alpha = 20$ would approximate absence of buffer management.

9Our implementation of the evaluated buffer-management schemes in ns-3 will be made available online.
buffer only $40 - 50\%$ (dependent on whether DCTCP is used) of the same bursts did not experience drops. FB superiority over DT’s performance in the single-queue scenario stems from FB’s ability to use multiple thresholds for the same queue as we explained in § 4 (combined, of course, with the threshold calculation). At the same time, We also observe that FAB’s prioritization mechanism also allows bursts to grow disproportionately more in the buffer. In effect, FAB performs up to 54.1x better than DT. Still, FAB performs up to 2.3x worse than FB, as it does not consider overall buffer contents.

**FB does not require information about traffic or optimal configuration.** As we show in §4, the operator can configure the FB’s $\alpha$ of all classes to guarantee that the buffer will fully absorb a specific burst while maximizing the space the other queues can use. Yet, since none of the alternative buffer management schemes have such capability, we configure all schemes with the same $\alpha$ values regardless of the burst size (see § 6.1). On the one hand, non-optimal configuration naturally deteriorates FB’s performance (hence the drops in bursts). On the other hand, though, the performance of FB under non-optimal configuration shows that FB is superior to existing buffer management algorithms and can improve the burst absorption without prior knowledge about traffic.

**Infrequent $\alpha$ updates according to FB (i.e., FBA) are beneficial.** FBA only updates the threshold according to FB’s calculations every $1\text{sec}$, effectively serving as a practical approximation of FB in any commodity device that implements DT and is only capable of infrequent $\alpha$ updates. Observe that FBA only slightly reduces the gains of FB compared to DT to 1.58x-2.05x in the multi-queue deployment. The slight reduction is owed to the FBA’s lag in adapting the thresholds of low-priority $\alpha$ values upon the sudden arrival of a burst.

**DCTCP indirectly improves buffer usage but cannot approach FB’s burst-absorption benefits.** DCTCP reduces the buffer that long flows use, indirectly leaving more headroom for bursts. As a result, DCTCP can reach FB’s burst-absorption benefits when the total burst size is smaller than 40% of the buffer in the multi-queue scenario. Still, for larger bursts and for single-queue scenarios, DT with DCTCP displays poor burst absorption. Concretely, as the burst size increases, DCTCP’s buffer savings are insufficient to compensate for DT’s inability to control the aggregate buffer usage. Observe that DCTCP (or any other congestion-control scheme) is orthogonal and beneficial to any buffer management scheme. As an illustration, FBA performs 6.15% better when combined with DCTCP compared to FBA when combined with TCP. Interestingly, FAB achieves better performance than DT with DCTCP in the single-queue scenario. FAB achieves this because it prioritizes short flows i.e., something that DT cannot do by design.

**Tail buffer occupancy correlates with burst absorption capabilities.** The relationship between burst absorption capabilities and buffer occupancy is better illustrated in Figures 7c and 7d where we plot the 99-th percentile buffer occupancy as a function of the burst size. Observe that as the burst size increases, FB’s 99-th percentile buffer occupancy also increases, indicating that FB only allows high buffer occupancy when there are bursts. FBA closely approximates this behavior but uses slightly more buffer, especially when not combined with DCTCP. Interestingly, DCTCP combined with DT displays a similar pattern, namely its 99-th buffer increases with the burst size. DT’s allocations are limited to 80% of the buffer as the rest is statically allocated to all queues. Recall that statically allocating part of the buffer is common practice [7, 11] that protects ports from starvation, since DT cannot offer such guarantees § 2.2. Finally, FAB and Cisco’s Intelligent buffer display almost 100% 99-th percentile buffer occupancy regardless of the burst size, effectively explaining their mediocre performance in terms of burst absorption.

**FB achieves on-par throughput under any background load.** To evaluate the achieved throughput of different schemes, we measure the average aggregate uplink utilization, while varying the load and plot our results in Fig. 8a and Fig. 8b for multi and single-scenario, respectively. All algorithms, including FB achieve on-par throughput. While FB uses less buffer to keep headroom available for bursts, it does not sacrifice throughput. Concretely, FB achieves this because it does not statically allocate any buffer but dynamically assigns it to queues proportionally to their dequeue rate instead. In effect, FB minimizes the time each packet spends at the buffer. TCP achieves a slightly higher throughput compared to DCTCP due to its aggressive buffer utilization.

**FB’s low buffer occupancy unintentionally benefits all short flows (i.e., also those of low priority).** Fig. 8c and Fig. 8d show the 95-th percentile FCT (Flow Completion Times) as a function of the background load. We observe that DCTCP achieves the lowest tail FCT when combined with either DT or FBA. This observation demonstrates the importance of congestion control when short flows share the same queue with long flows. Even when combined with TCP, though, FB keeps the tail FCT relatively low, thanks to its low buffer occupancies. Concretely, FB performs, on average, 34.1% better than DT with TCP.

Besides DCTCP superiority and FB’s benefits compared to DT, Fig. 8c and Fig. 8d display an additional interesting pattern: as the background load increases, the tail FCT increases and then decreases. As the background load increases, the remaining buffer decreases together with the thresholds of all queues. The impact of this trend in the tail FCT is two-fold. On the one hand, the tail FCT of short flows improves (decreases) as the thresholds of their queues increase because they are less likely to experience drops. On the other hand, the tail FCT of short flows deteriorates (increases) as the thresholds of their queues increase because of the added queuing delays. At a load lower than 40% short flows are not experiencing any drops.
In this section, we aim at answering two questions:

Q1: Is PISA FB capable of controlling the buffer allocation directly in the data-plane?
Q2: Does PISA FB fit the tight resource budget of a real device?

We answer both questions affirmatively. Next, we describe our proof-of-concept setup before we elaborate on the findings.

Setup. We implement PISA FB on a Tofino Wedge 100BF-32X in 850 lines of P4 code. We use this code to instantiate the testbed that we illustrate in Fig. 9a. Our testbed contains a Tofino switch and two servers (S1 and S2). S1 connects to the Tofino via port p1, while S2 via ports p2 and p3. We configure 8 queues per port. Among the 8 queues, 1 is used for high-priority TCP traffic and 7 for low-priority UDP traffic. We configure $\alpha = 0.8$ for high-priority traffic and $\alpha = 0.6$ for low.\footnote{The selection of $\alpha$ is affected by design choices specific to Tofino and beyond our control. Still, higher $\alpha$ would make DT’s performance worse.} We limit the aggregate bandwidth and buffer size to $100\, \text{Mbps}$ and the shared buffer $27\, \text{KB}$ if the device is configured with the maximum stateful memory. The memory requirements can grow up to $300\, \text{KB}$ per port and once with FB.

Q1: FB PISA controls the buffer allocation without using/programming the Traffic Manager. Fig. 9b presents the CDF of the FCT of all TCP flows. FB reduces short flows FCT by at least 50ms compared to DT. While the exact FCT improvement is not important here, it shows that FB can in practice limit the buffer occupied by low-priority UDP flows. In effect, short flows of high-priority can occupy more buffer thus experience less drops.

Q2: The resource footprint of PISA FB is low. First, our implementation of PISA FB uses only 6 stages in the ingress and 3 stages in the egress pipeline. This usage is well within Tofino’s limit of a few tens of pipeline stages [25]. Moreover, if FB was implemented in the TM, no pipeline stages would be needed. Second, our implementation uses only $8\, \text{KB}$ of stateful memory. The memory requirements can grow up to $27\, \text{KB}$ if the device is configured with the maximum number of queues and ports. In any case, the memory overhead is insignificant compared to the tens of $\text{MB}$ of available memory [42].

8 Related Work

FB is related but complementary to algorithms operating at the port level (e.g., queue management, scheduling) and host level (e.g., TCP). First, active Queue Management and scheduling algorithms can facilitate preferential treatment of...
some flows over others, but only if they are mapped to the same output port. Indeed, AQM pro-actively controls individual queues e.g., RED [34], Codel [41], PIE [56] and limits the per-flow buffer or bandwidth [22, 33, 39, 53] at the per-port level. Second, scheduling techniques e.g., pFabric [15] and PIAS [20] allow certain flows to be dequeued faster than others of the same port. Third, congestion control mechanisms, such as [14, 24, 26, 50, 62, 70, 71] reduce the unwanted buffer usage, but cannot detect or react to high overall buffer occupancy, neither can they change the allocated buffer at the device level.

Regarding buffer algorithms more recent works such as EDT [63], FAB [16] and Cisco IB [2] empirically recognize bursts and prioritize them. While intuitive, such algorithms cannot consistently (under various loads) absorb bursts as we show in §6 and cannot be trivially mapped to multi-queue scenarios. On the contrary, FB offers provable guarantees under various scenarios. Other buffer management algorithms such as [36, 43, 60, 61] are not applicable to Call Admission Control (CAC), while pushout-based ones such as [68], [65] are considered impractical [18].

9 Conclusion

In this paper, we demonstrate the limitations of today’s most common buffer management algorithm, Dynamic Thresholds, both experimentally and analytically. We present FB, a novel algorithm that offers provable burst-tolerance guarantees, without sacrificing throughput or statically allocating buffer. We show that FB outperforms all other buffer management techniques even when they are combined with DCTCP. Finally, we show that FB’s design is practical by implementing it on a Barefoot Tofino and evaluating its approximation to commodity hardware.

References


[38] Yihua He, Nitin Batta, and Igor Gashinsky. Understanding switch buffer utilization in clos data center fabric.


[41] V Jacobson and N Kathleen. Controlling queue delay—a modern aqm is just one piece of the solution to bufferbloat. Association for Computing Machinery (ACM Queue), 2012.


A. Analysis

In this section we formally model and analyze the shared memory switch architecture (§2.1) with FB’s buffer allocation scheme. The aim of our analysis is to show FB’s formal guarantees, its properties. Due to space contraints, a full present a complete version of our analysis in [19] including the analysis of parameter setting. We refer the reader to §2 and §4 for the terminology and definitions regarding ports, queues, class and priority.

A.1 Model

For generality, we model a switch with arbitrary but fixed number of ports and queues per port. In particular, each port has only one queue per class as defined in (§2). The switch in our model has a shared memory architecture with $B$ buffer space. We denote the instantaneous occupied buffer at time $t$ as $Q(t)$. Our analysis is based on a fluid model where packet (bits) arrivals and departures are assumed to be fluid and deterministic.

We denote by $\alpha_c$, the parameter used by FB for each class in allocating the buffer. Each class is associated with a separate queue at each port. Additionally, FB requires each class to be mapped to a priority. A Low priority is a set of classes which share the buffer fairly proportionate to their alpha values. A High Priority is simply a set consisting of one class. We denote port indices by $i$, class by $c$ and priority $p$ where $p$ is a set of classes. The number of congested queues of a priority $p$ at time $t$ by $N_p(t)$.
A.2 Formalizing FB’s allocation

As described in §4, the threshold of a queue at port $i$, of class $c$ and belonging to a priority $p$ is calculated based on the alpha parameter $\alpha_c$, the number of congested queues $N_p(t)$, the normalized dequeue rate of the queue denoted by $\gamma_c^i(t)$ and the remaining buffer $B - Q(t)$. Formally,

$$T^i_c(t) = \alpha_c \cdot \beta_p(t) \cdot \gamma_c^i(t) \cdot (B - Q(t)) \quad (9)$$

where, $\beta_p(t) = \frac{1}{N_p(t)}$ is the inverse of the total number of congested queues of priority $p$ (to which the class $c$ belongs to) at time $t$.

Definition 1 (Omega, adaptive alpha parameter). For a queue belonging to a class $c$ of priority $p$, the product $\alpha_c \cdot \beta_p(t) \cdot \gamma_c^i(t)$ in FB’s buffer allocation scheme (Eq. 9) is defined as Omega denoted by $\omega_c^i(t)$ and is viewed as an adaptive alpha parameter.

$$\omega_c^i(t) = \alpha_c \cdot \beta_p(t) \cdot \gamma_c^i(t) \quad (10)$$

Based on the above definition of $\omega$, in the following we derive an upper bound on the sum of $\omega$ values for all the classes of a priority $p$. Later in our analysis, we will see how the sum of $\omega$ values play a key role in FB’s buffer allocation scheme. We will later use this upper bound to derive several properties and formal guarantees provided by FB.

Lemma 1. The instantaneous sum of $\omega_c^i(t)$ over all the classes belonging to a priority $p$ across all the ports is upper bounded by $\alpha_p$, where $\alpha_p = \max_{c \in p} \alpha_c$.

$$\sum_i \sum_{c \in p} \omega_c^i(t) \leq \alpha_p \quad (11)$$

Proof. Using Definition 1 and observing that $\beta_p(t)$ is the number of congested queues of a priority $p$ is same across all the classes $c \in p$, we express the sum of $\omega_c^i(t)$ as follows:

$$\sum_i \sum_{c \in p} \omega_c^i(t) = \sum_i \sum_{c \in p} \alpha_c \cdot \beta_p(t) \cdot \gamma_c^i(t) = \beta_p(t) \cdot \sum_i \sum_{c \in p} \alpha_c \cdot \gamma_c^i(t)$$

Since $\gamma_c^i(t)$ is the normalized dequeue rate, $\gamma_c^i(t)$ is upper bounded by 1. Finally we substitute $\beta_p(t) = \frac{1}{N_p(t)}$. We reduce the sum to an inequality as follows, where the last inequality holds since the average of $\alpha_c$ is less than the maximum.

$$\beta_p(t) \cdot \sum_i \sum_{c \in p} \alpha_c \cdot \gamma_c^i(t) \leq \frac{1}{N_p(t)} \cdot \sum_i \sum_{c \in p} \alpha_c \leq \max_{c \in p} \alpha_c$$

A.3 Steady-State Analysis

We now analyze the steady-state behavior of FB’s buffer allocation scheme. Specifically, we say steady-state when the load-conditions remain stable and a steady buffer occupancy is achieved. Under steady-state, the queue lengths remain stable at less than or equal to their corresponding thresholds. To stress on the worst-case scenarios, we assume that any occupied queue is at the respective threshold. In our steady-state analysis, for simplicity of presentation, we drop the time variable in all the equations.

Under steady-state, we are interested in determining the overall buffer allocation and occupancy denoted by $Q$, the remaining buffer space $B - Q$ and FB’s threshold calculation per queue $T^i_c$.

Lemma 2 (Steady-state allocation). Under steady-state, given a set of congested queues, the overall buffer occupancy $Q$ is given by Eq. 12, the remaining buffer $B - Q$ is given by Eq. 13 and the threshold per congested queue calculated by FB is given by Eq. 14.

$$Q = \frac{B \sum_i \sum_c \omega_c^i}{1 + \sum_i \sum_c \omega_c^i} \quad (12)$$

$$B - Q = \frac{B}{1 + \sum_i \sum_c \omega_c^i} \quad (13)$$

$$T^i_c = \frac{B \cdot \omega_c^i}{1 + \sum_i \sum_c \omega_c^i} \quad (14)$$

Proof. In the steady-state, from the assumption that the queue lengths are equal to their thresholds, we derive the overall buffer occupancy by summation of queue lengths of all the congested queues. Using Eq. 9 and Eq. 10 we express the total buffer occupancy $Q$ as follows and solve for $Q$ leading to the last equality.

$$Q = \sum_i \sum_c \omega_c^i \cdot (B - Q) = \frac{B \sum_i \sum_c \omega_c^i}{1 + \sum_i \sum_c \omega_c^i}$$

The remaining buffer space (Eq. 13) is then straight-forward by substituting $Q$. Finally, the threshold per queue $T^i_c$ (Eq. 14) is obtained by definition from FB’s allocation scheme i.e.,

$$T^i_c = \alpha_c \cdot \beta_p(t) \cdot \gamma_c^i(t) \cdot (B - Q) = \omega_c^i \cdot (B - Q).$$

Using Lemma 2, we now derive the overall buffer occupancy of FB.

Theorem 1. FB’s buffer allocation scheme upper bounds the occupied buffer space $Q$ given by Eq. 15, where $\alpha_p = \max_{c \in p} \alpha_c$.

$$Q \leq \frac{B \cdot \sum_p \alpha_p}{1 + \sum_p \alpha_p} \leq \frac{B \cdot (\alpha_l + \alpha_h)}{1 + (\alpha_l + \alpha_h)} \quad (15)$$
\textbf{Proof.} From Lemma 2, we rewrite the occupied buffer space \( Q \) as follows by expanding the summation over classes \( c \) into a summation over classes \( c \in p \) for each priority \( p \):
\[
Q = \frac{B \cdot \sum_c \omega_c}{1 + \sum_c \omega_c} = \frac{B \cdot \sum_{p} \sum_{c \in p} \omega_c}{1 + \sum_{p} \sum_{c \in p} \omega_c}
\]

We use the above equality and obtain Eq. 15 since from Lemma 1 we have that \( \sum_{c \in p} \omega_c^L(t) \leq \alpha_L \) and the inequality in Eq. 15 holds. \( \square \)

In the following we derive FB’s formal guarantee on the remaining buffer space i.e., irrespective of the number of congested queues, FB offer minimum buffer space based on the \( \alpha \) parameters formally stated in the following theorem.

\textbf{Theorem 2.} In the presence of high-priority classes, FB’s buffer allocation scheme upper bounds the buffer occupancy \( Q \) by the Low Priority set of classes \( L \). The bound is given by Eq. 16, where \( \alpha_L = \max_{c \in L} \alpha_c \).
\[
Q \leq \frac{B \cdot \alpha_L}{1 + \alpha_L + \alpha_H}
\]  
(16)

\textbf{Proof.} From Lemma 2, we rewrite the occupied buffer \( Q \) as follows for the case with both low and high priority classes:
\[
Q = \frac{B \cdot \sum_c \omega_c^L \sum_{c \in \{L,H\}} \omega_c^H}{1 + \sum_c \omega_c^L \sum_{c \in \{L,H\}} \omega_c^H}
\]

Since from Lemma 1 we have that \( \sum_{c \in L} \omega_c^L(t) \leq \alpha_L \) and \( \sum_{c \in H} \omega_c^H(t) \leq \alpha_H \), the inequality in Eq. 16 holds. \( \square \)

\textbf{Theorem 3.} If \( B_p \) amount of buffer is required for the low priority, then \( \alpha_L \) must be chosen as follows:
\[
\alpha_L \geq \frac{B_p \cdot (1 + \alpha_H)}{B - B_p}
\]  
(17)

\textbf{Proof.} The proof is straight-forward from the upper bound of the buffer allocated to low priority even in the presence of high priority. From Theorem 2, the upper bound on the overall occupied buffer is \( Q \leq \frac{B \cdot \alpha_L}{1 + \alpha_L + \alpha_H} \). Then the lower bound on the remaining buffer is \( B - Q \geq \frac{B}{1 + \alpha_L + \alpha_H} \). The total buffer that is allocated to the low priority is then greater than or equal to \( \alpha_L \cdot (B - Q) \). Using this relation and using the requirement that \( \alpha_L \cdot (B - Q) \geq B_p \), we obtain \( \alpha_L \geq \frac{B_p (1 + \alpha_H)}{B - B_p} \). \( \square \)

Theorem 1 states an important property on how the overall buffer occupancy and consequently the remaining buffer scope are bounded. For example, consider that there are only two priorities \textit{Low Priority} and \textit{High Priority} with maximum alpha parameters \( \alpha_L \) and \( \alpha_H \) respectively. The remaining buffer space, irrespective of the number of queues and their dequeue rates, is bounded by \( B - Q \geq \frac{B}{1 + \alpha_L + \alpha_H} \). Further in the absence of high priority packets in the buffer, we can also observe that the remaining is greater than equal to \( \frac{B}{1 + \alpha_L} \) which leaves predictable amount of remaining buffer space for incoming high priority traffic e.g., to absorb a burst without statically reserving buffer space.

Due to space constraints, we only present steady-state analysis in this paper. A full length analysis, including transient-state properties and parameter setting can be found in [19].

\section*{B FAQ}
We address common questions including ones about the production-readiness of our implementation, or the representativeness of our experiments and relation with congestion control.

\subsection*{B.1 Is the FB’s implementation production-ready?}

The goal of this paper is not to find yet another use case for programmable devices. Thus, designing an optimal implementation of FB on Tofino is beyond the focus of our work and would require access to the Traffic Manager. Instead, we suggest various ways of deployment to allow more network operators to benefit from our work. First, \textit{Vamsi: sentence is not clear!} PISA FB (§ 5.1) would be suitable if PISA switches are useful for other purposes or/and the traffic is so dynamic that it requires very frequent updates. FBA (§ 5.2) is suitable for large deployments containing heterogeneous devices and/or deployments in which the hardware switches are not capable of very frequent \( \alpha \) updates. Even very frequent updates e.g., every couple of hours, could be beneficial for a network with less dynamic traffic. Observe that these deployments are possible because FB only requires simple operations and statistics that are already available on any switch.

\subsection*{B.2 Why not run PISA FB on 100G or at the full buffer size?}

Our experiment is representative of FB’s behavior on a PISA switch at scale (i.e., when run with higher bandwidth and larger buffer) for three key reasons. First, the number of pipeline stages and the amount of stateful memory is independent of the number of instantaneous active queues, of throughput, and of the available buffer, as we explain in § 5.1. Indeed, these resources are only dependent on the number of ports and queues. Second, the number of recirculated packets (SYNC) is independent of the incoming load and of the available buffer. Recall that the recirculated packets only need to sync ingress and egress state on a set interval, which defines the quantity of required recirculated traffic. If we consider a 1us interval, then the recirculated traffic does not exceed 30KBps, which only marginally affects throughput. Importantly, the benefits in the performance of FB even for much courser-grained update interval i.e., 1sec (thus even less recirculation) are significant as we show in § 6.2. Third, the new Tofino generation (Tofino2) provides egress queue information at the ingress pipeline [10]. This modification in the
architecture will significantly simplify the design by alleviating the burden of recirculation.

Since PISA FB is not sensitive to increases in throughput or buffer size, a larger-scale experiment would only stress-test Tofino. Stress-testing Tofino is neither within our scope nor feasible with our current hardware resources. Observe that in order to stress the buffer of a Tofino device, we will need to send on multiple ports at full speed (the input rate should be higher than the output for some time). Unfortunately, we do not have access to a traffic generator that can send multiple 100Gbps of traffic.

B.3 Why evaluate with 10G?
We evaluate FB at 10G (which corresponds to days-worth of ns-3 experiments) to allow for easier reproduction of our results. We stress that our experiments are representative of datacenter line rates for three key reasons. First, we have verified that a concurrent increase in the aggregate bandwidth and buffer size will not change our results. In fact, the required buffer size per port typically increases linearly with bandwidth for loss/ECN based TCP protocols [14, 17] given that TCP’s statistical multiplexing does not hold for datacenter traffic [14]. Second, we set the switch parameters (number of ports, aggregate bandwidth, buffer size) according to an existing chipset, namely Trident+ [9]. Finally, our evaluation results are backed-up by provable observations (see Appendix § A).

B.4 Why not evaluate FB with more-advanced CC algorithms than DCTCP?
We include DCTCP in the evaluation (instead of more advanced CC schemes) because it sits in sweetspot between performance and generality. First, DCTCP has been recently shown [45] to out-perform many of the newer congestion control algorithm when evaluated on a common network architecture (i.e., no kernel bypass). Second, DCTCP does not make any assumption about the network capabilities and architecture. Indeed, TIMELY/Swift [44, 50], DCQCN [71], HPCC [45] consider a kernel-bypass network stack, while HOMA [51], NDP [37] and Aeolus [40] consider a non-oversubscribed network with optimal load balancing.

B.5 Would CC algorithms solve all buffer-related problems?
While CC can indirectly control the buffer occupancy, they cannot provide guarantees. For many popular and recently proposed CC algorithms, there are corner cases in which an advanced buffer management scheme is required. To give better intuition, we describe some of these corner cases for well-known CC algorithms.

First, many known CC algorithms [14, 44, 45, 50, 71] observe per-queue statistics but cannot estimate and thus act upon the overall occupancy of the shared buffer. In effect, such algorithms cannot preferentially treat a certain high-priority queue over other low-priority ones that are attached in different ports. Second, pure RTT-gradient approaches such as TIMELY [50] have been proved to be unstable [72] and cannot maintain low overall buffer occupancies. Third, CC algorithms cannot fundamentally handle sub-RTT bursts as it requires at least one RTT to receive network feedback (or credits from the receiver) and converge to an equilibrium. Finally, some CC algorithms [37, 51] work only under very specific assumptions about the network, such as perfect load balancing and no oversubscription.

B.6 FB wastes buffer space.
Any buffer management scheme that cannot over-write packets would need to keep some buffer empty to prevent newly-formed queues from starvation and for allowing bursts to pass. Observe that while FB keeps the buffer occupancy lower than DT it achieves better performance, as we show in § 6.2 for two key reasons. First, by reducing the buffer occupancy in steady-state (which is critical for throughput) FB is able to provide more headroom (which is critical for throughput for bursts). Second, FB does not statically allocate any buffer. The steady-state is the maximum possible to satisfy the set burst-tolerance guarantees.

B.7 What if low priority traffic is bursty?
While FB focus on providing burst tolerance guarantees to high priority traffic without sacrificing throughput it also allows burst absorption the low priority. For instance, consider the case in which the operator wants to allow a class of low priority to allocate more buffer (i) when high priority is not heavily using the buffer; or (ii) when high priority has only small burst-tolerance requirements. The operator needs to only configure a higher $\alpha$ for this class compared (i) to the other low priority classes or (ii) the high-priority class. In essence, FB provides the formulas to adapt the buffer to the particular requirements of each network. Observe that DT works on a best effort manner as we explained in § ??.

B.8 if I needed to use all 8 classes, is the scheme till useful

B.9 Traffic classification being a practical assumption.