PLASTICINE: A Flexible Buffer Management Scheme for Data Center Switches

Maria Apostolaki ETH Zürich
Vamsi Addanki ETH Zürich
Manya Ghobadi MIT
Laurent Vanbaver ETH Zürich

Abstract

Today, network devices share buffer across output queues to avoid drops during transient congestion with less buffer space, thus with a lower per-chip cost. While effective most of the time, this sharing can cause undesired interactions among seemingly independent traffic, especially in high load. As a result, low-priority traffic can cause increased packet loss to high-priority traffic, and long flows can prevent the buffer from absorbing incoming bursts. The cause of this perhaps unintuitive outcome is today’s buffer sharing techniques that are unable to guarantee isolation even to a small portion of the traffic, without statically allocating buffer space. To address this issue, we designed PLASTICINE, a novel buffer management scheme that offers strict isolation guarantees without keeping the buffer underutilized. PLASTICINE significantly reduces Query Completion Times (QCT) and Flow Completion Time (FCT) of short flows while achieving on-par throughput compared to conventional buffer management algorithms even when they are paired with DCTCP. We show that PLASTICINE is practical and runs at line-rate on existing hardware (Barefoot Tofino).

1 Introduction

To reduce costs and maximize utilization, network devices often rely on shared buffer whose allocation across ports and queues is dynamically adjusted by a buffer allocation algorithm such as Dynamic Thresholds. While sharing buffer avoids wasting buffer space during low utilization, it can also create unpredictable interferences across seemingly independent traffic (e.g., traffic going to different ports) during high utilization. As we show, these interferences are disruptive. For instance, high-priority traffic going to a certain port can experience drops because of the excessive buffering caused by low-priority traffic on other port(s).

While they can help mitigating interferences, Congestion Control (CC) algorithms and scheduling techniques cannot solve the problem as they operate at the queue- and port-level, respectively. Concretely, while CC might help decreasing buffer usage on a per queue basis, it cannot mitigate interferences across queues as it is oblivious to the respective buffer utilization and relative importance of the traffic competing in different ports/queues. Likewise, scheduling cannot prevent incoming packets from being dropped if the corresponding queue has reached its maximum allocation.

To address the problem of interferences, we design PLASTICINE: a novel buffer management algorithm that adapts buffer allocation not solely based on the current buffer occupancy (as Dynamic Thresholds) but also based on the buffer content and its temporal characteristics. Concretely, PLASTICINE dynamically bounds the buffer allocation at different levels (e.g., per traffic class, priority) to prevent certain types of traffic across queues from monopolizing the buffer. In particular, PLASTICINE favors queues that free-up their used buffer faster, effectively making it accessible to more traffic. As a result, PLASTICINE can offer strict burst-tolerance and minimum allocation guarantees without resorting to static allocations.

We found that PLASTICINE outperforms, under high load, all alternative buffer management techniques even when they are combined with DCTCP. Among others, we show that PLASTICINE reduces Query Completion Times by 23% and the 99-th percentile Flow Completion Time by 83% compared to Dynamic Thresholds with DCTCP, and that, while achieving on-par throughput. We also show that PLASTICINE is implementable in programmable data planes (Barefoot Tofino).

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 inefficiencies both experimentally and analytically.
- A novel approach for buffer management that can
provenly provide burst-tolerance guarantees without statically allocating buffer ($\S4$).

- A novel hardware design and implementation of PLASTICINE on a Barefoot Tofino switch $\S5$ that demonstrates its practicality in today’s hardware ($\S5$).

- A comprehensive evaluation demonstrating that PLASTICINE outperforms state-of-the-art buffer management algorithms even when combined with DCTCP ($\S6$).

2 Background & Motivation

In this section, we first describe our model, namely the network device architecture we consider ($\S2.1$). Next, we explain how Dynamic Thresholds (DT), the most commonly-used buffer allocation algorithm, works ($\S2.2$). Finally, we reveal DT’s core inefficiencies ($\S2.3$).

![Figure 1:](image)

**Figure 1:** Incoming packets are stored in the shared buffer for future transmission if their corresponding queues’ length is below the threshold that a buffer management algorithm calculates and dropped otherwise.

2.1 Switch Model

Fig. 1 shows a simplified output-queued shared-memory packet switch $\S2$. The switch implements a fixed or programmable logic according to which incoming packets are mapped to one output queue for future transmission. Multiple queues are associated with each egress port. Each queue is dedicated to a traffic class. Upon the arrival of a packet, the Traffic Manager (TM) decides whether it can be enqueued and thus stored in the shared buffer; otherwise the packet is dropped. To do so, the TM compares the queue’s length with a threshold that is calculated according to the buffer management algorithm, e.g., Dynamic Thresholds (DT).

We assume that the traffic is grouped into classes, and classes are of high or low priority. Yet, our model also applies to a single class and/or a single priority.

$\text{Traffic Class}$ The use of multiple traffic classes allows the operator to differentiate traffic corresponding to different applications for isolation purposes $\S4$. Each traffic class uses exclusively one queue in each port. As an intuition, in Fig. 1 storage, VoIP and MapReduce traffic correspond to different traffic classes. Thus, each of them uses a separate queue in each port.

High & Low Priority The grouping of classes to high and low priority allows the operator to prioritize certain classes. As an intuition, in a cloud environment, customer-generated traffic would be high-priority as it is subject to Service Level Agreements (SLAs), while internal traffic (e.g., maintenance traffic) would be low-priority, effectively using the available network in a best-effort manner.

As an illustration, in Fig. 1 the MapReduce-traffic class is high-priority, while VoIP and storage are low.

2.2 DT’s workings

We now describe Dynamic Thresholds (DT), the most common buffer management algorithm in today’s devices $\S2.3$. DT works by dynamically adapting the threshold $T_c(t)$ beyond which a queue of class $c$ in port $i$ cannot grow according to the remaining buffer and a parameter $\alpha$ $\S2.2$ (see Eq. (1)). Intuitively, DT allocates buffer to each queue: (i) negatively proportional to its utilization: the less remaining buffer there is, the less a queue can grow; and (ii) directly proportional to a parameter $\alpha$ configured per class $\S3$ the higher $\alpha$, the more the queue can grow.

$$T_c^i(t) = \alpha^i_c \cdot (B - Q^i(t))$$ (1)

$T_c^i(t)$: Queue threshold of class $c$ in port $i$

\(\alpha^i_c\): $\alpha$ parameter of class $c$ in port $i$

\(B\): Total buffer space

\(Q^i(t)\): Total buffer occupancy at time $t$

The $\alpha$ parameter of a queue impacts its maximum length and its relative length with respect to the other queues. Thus, an operator is likely to set higher $\alpha$ values for high-priority traffic classes compared to low-priority ones. Despite its importance, there is no systematic way to configure $\alpha$, meaning different vendors and operators reportedly use different $\alpha$ values. For instance, Yahoo uses $\alpha = 8$ $\S3$ while Cisco has default $\alpha = 14$ $\S3$ and Arista $\alpha = 1$.

DT in action Assume the switch shown in Figs 2a 2b with a shared buffer of 60 packets and 2 ports. Incoming packets are of two traffic classes colored in red and yellow corresponding to high and low priority respectively. The

\(1\) We describe the mapping of our model to RMT architecture in $\S5$

\(2\) TM’s architecture is the same for both fixed-function and reconfigurable switches $\S28$, including Barefoot’s Tofino and Broadcom’s Trident series.

\(3\) While the operator can configure $\alpha$ per queue, in practice it is configured per class.

\(4\) For simplicity we assume a single buffer chip per device.
As we observe in Fig. 3 before time $t_0$, only a queue of the low-priority yellow class, say $Q1$, uses the buffer. The buffer at this state is also illustrated in Fig. 2a. The buffer is in steady state, meaning all active queues’ length equals to DT’s threshold. Indeed, $Q1$’s length is 30 packets, the remaining buffer space is equal to 30 packets and thus $Q1$’s threshold is $1 \times 30 = 30$. At time $t_0$, a burst of the high-priority red-class destined to port 2 hits the switch. The burst forms a queue, say $Q2$, whose length increases very fast in the time frame $t_0 - t_1$ due to its high incoming rate, as we observe in Fig. 3 (red solid line). As $Q2$ grows in the buffer, the remaining buffer decreases and so does $Q1$’s threshold (yellow dashed line). Observe that the $Q1$’s threshold decreases faster than its queue length (yellow solid line). During the time frame $t_1 - t_2$, the buffer is in transient state, meaning the length of a queue (in particular $Q1$) is above its threshold. During this time frame, all incoming packets to $Q1$ are dropped. As a result, $Q1$’s length decreases at a rate equal to its dequeue rate until the queue’s length reaches its threshold, at time $t_2$. At $t_2$ the buffer reaches again steady state. The buffer at this state is also illustrated in Fig. 2b. This time, the remaining buffer is 15 packets. The yellow queue ($Q1$) occupies $15 = 1 \times 15$ packets while the red queue ($Q2$) occupies $30 = 2 \times 15$ packets. Notably, while the buffer is in transient state, 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 the instantaneous threshold (red dashed line) that DT calculates. As a result, during this time frame $t_1 - t_2$, some of the packets that should have been enqueued to $Q2$ are dropped. Observe that the $Q2$’s drops in the time frame $t_1 - t_2$ could have been avoided if the buffer had reached steady state faster.

### 2.3 DT’s inefficiencies

DT allocates buffer in a highly unpredictable, best-effort manner; thus it is unable to offer any guarantees for steady or transient-state allocation. Next, we demonstrate the key reasons for DT’s unpredictable behavior both analytically and using intuitive examples. We use the same examples to show how PLASTICINE would allocate buffer in §3.

#### DT offers no minimum buffer guarantee.

DT can differentiate performance across traffic classes only via a static parameter ($\alpha$). Yet, $\alpha$ offers no guarantee, since the actual threshold (Eq. 1) depends on the overall remaining buffer, which can reach arbitrarily and uncontrollably low values, even in the steady state.

As an illustration, recall in the case of Figs. 2a, 2b the operator favors the high-priority red class by configuring a higher $\alpha$ ($alpha = 2$) compared to $alpha = 1$ of the low

---

5This analysis shown in Fig. 3 is based on the fluid model, but we have also verified the results in simulation. For more details on the model please refer to Appendix B
priority classes). Indeed, the red-class queue could potentially occupy a large portion of the buffer (40 packets) and 2x more buffer than queues of any other class. In practice though, the buffer that the red-class queue will occupy depends on the buffer’s occupancy and thus at all the other queues. For example, consider the state illustrated in Fig. 2c where there are three queues of the yellow class (low priority) in ports 2,3 and 4 and one red-class queue (high priority) in port 1. The red queue will only be able to occupy 20 packets as the remaining will be 10. Observe also that the three other queues collectively occupy 30 packets, namely more than the critical red queue that belongs to high priority.

To sum up, when DT is used, the buffer occupancy and thus the remaining buffer depends on the number of queues that are congested, as we observe in Eq. 2, and that, independently of the priority or class they belong to. This insight is experimentally supported by [20], in which authors observed the behavior of programmable switches of different vendors. The effect of this insight is illustrated in Fig. 4, where we plot the buffer that a high-priority queue will take as a function of the number of low-priority queues in the buffer. Notably, even a single queue of seemingly high-priority traffic can end up with little to no buffer due to the presence of low-priority traffic. Similar to [7], we refer to this phenomenon as buffer pressure.

\[ Q(t) = B \cdot \frac{\sum_{i=1}^{N} \alpha_i^c}{1 + \sum_{i=1}^{N} \alpha_i^c} \]

\(Q(t)\): Buffer occupied  
\(N\): set of queues (i,c) that are congested

**DT offers no burst-tolerance guarantees** Apart from the unpredictability of the steady-state allocation, DT’s transient-state allocation, which is critical for burst-absorption, is uncontrollable. Indeed, DT perceives buffer space as a scalar entity ignoring its content and expected occupancy over time.

For instance, in Fig. 5a multiple packets of different classes are queued-up waiting to be transmitted from port 2. Thus, DT allocates 10 packets for each queue, as they are configured with the default (low-priority) \(\alpha = 1\) and the remaining buffer space in the steady-state is 10. In Fig. 5b a high-priority 5 : 1 incast hits port 1, namely 5 incoming ports simultaneously send to port 1. Due to DT’s prior allocations, though, the device has neither enough remaining buffer nor can it be emptied fast enough to keep up with the incoming traffic. As a result, the high-priority burst starts to experience drops while it only has 8 packets in the buffer, that is 9 packets less than its steady-state allocation, which would be \(\sim 17\) packets. In other words, the high-priority traffic class is experiencing drops in the transient state, which could have been avoided if the buffer could reach steady state faster. This is the case because the 5 low-priority queues share the dequeue rate of a single port.

Notably, DT has no way to distinguish between 5 queues coexisting in a single port and 5 queues each attached to a separate port. Yet, if each queue was in a different port, in Fig. 5b then the buffer would reach steady state 5x faster and the burst could have been absorbed.

To sum up, DT has unpredictable transient state. In effect, the time frame during which a queue experiences drops before it allocates its fair (steady-state) share of the buffer can be arbitrary long. Thus, DT cannot guarantee the absorption of a burst.

### 3 Overview

To address DT’s shortcomings, we design PLASTICINE, a novel buffer management scheme that offers isolation, effectively mitigating cross-priority interferences and provides strict burst-tolerance guarantees. Next, we describe the key insights behind PLASTICINE’s design using the same examples we used to illustrate DT’s inefficiencies in §2.3.

**PLASTICINE dynamically bounds the buffer allocation in steady-state.** PLASTICINE prevents any single-priority traffic from monopolizing the buffer by dynamically bounding buffer usage per priority. In effect, both priorities are guaranteed a minimum allocation. Notably, PLASTICINE’s allocation does not pre-allocate buffer, which would wastefully keep the buffer unoccupied. As a result, PLASTICINE’s allocation is not equivalent to statically allocating space to a single queue (complete partitioning) or to a group of queues (application pools). As an illustration, unlike DT, which decreases the buffer space occupied by high-priority traffic proportionately to the number of low-priority queues (Fig. 2c).
LASTICINE bounds low priority queues to 15 packets on aggregate, equally shared across the three queues. As a result, the high-priority queue can use 30 packets of buffer (Fig. 5a). As we show in §3, LASTICINE makes the buffer that high-priority occupies independent of the number of queues the low-priority employs. Note that no allocation is static. Namely, if the high-priority queue does not use/need its maximum buffer occupancy, the yellow low-priority will get more buffer.

LASTICINE makes bursts first-class citizens in the buffer by minimizing transient-state drops. LASTICINE 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) adequate aggregate dequeue rate (i.e., the buffer can be emptied fast enough), to accommodate a burst. Indeed, both above 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. On the other hand, an incoming burst can also be absorbed independently of the aggregated dequeue rate of the buffer at its arrival, if the unoccupied buffer is sufficient to directly accommodate it. In the common case, LASTICINE achieves a balance between the two extremes.

As an illustration, since the aggregate dequeue rate in Fig. 5b is inevitably low (queues share a single port), LASTICINE limits each low-priority queue to 6 packets, as shown in Fig. 6b effectively leaving an incoming burst enough free buffer to be stored. As a result, when the same (as in Fig. 5b) 5:1 high-priority burst arrives, the buffer can reach steady state fast enough to prevent the burst from being dropped (Fig. 6c).

4 Design

Having explained LASTICINE’s high-level properties (§3), we now describe LASTICINE in detail. We elaborate on LASTICINE’s threshold calculation (§4.1), before we explain its consequences in LASTICINE’s performance and guarantees (§4.2).

4.1 LASTICINE’s workings

LASTICINE limits the buffer space each queue can use depending on both queue-level and buffer-level information. Particularly, as shown in Eq. 3, LASTICINE’s per-queue threshold equals the product of: (i) an \( \alpha \) value assigned to the class that the queue belongs to: \( \alpha_c \); (ii) the inverse number of congested queues of the priority (low or high) that the class belongs to: \( \frac{1}{N_p(t)} \); (iii) the normalized dequeue rate of this queue: \( \gamma_c(t) \); and (iv) the remaining buffer space: \( B - \bar{Q}(t) \).

\[
T_c(t) = \alpha_c \cdot \frac{1}{N_p(t)} \cdot \gamma_c(t) \cdot (B - \bar{Q}(t)) 
\]  
(3)
$N_p(t)$ : Number of congested queues of priority $p$ at time $t$
$B - B_{oc}(t)$ : remaining buffer
$\gamma_c(t)$ : normalized dequeue rate of $q_c$

**PLASTICINE on a single queue per port** Eq.3 can be naturally adapted to work in a deployment where only a single queue is available per port. Particularly, $\gamma$ 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, PLASTICINE applies different thresholds for packets that are mapped to the same queue.

PLASTICINE’s threshold calculation differs from that of DT (Eq.1) by two factors: (i) the number of congested queue per priority, $N_p(t)$; and (ii) the normalized dequeue rate of $\gamma_c(t)$ We explain how each of those differentiates PLASTICINE’s buffer allocation below.

$N_p(t)$ bounds steady state allocation, effectively creating minimum allocation guarantees  PLASTICINE divides per-queue thresholds with $N_p$: the number of congested queues of the priority that the class belongs to, as seen in Eq.3. The consequence of this factor to PLASTICINE’s allocation is twofold: (i) it bounds per-class and per-priority occupancy; and (ii) it allows weighted fairness across classes of 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 $\frac{\alpha_q}{1+\alpha_q}$ of the total buffer and one to the per-priority occupancy to $\frac{\alpha_p}{1+\alpha_p}$ of the total buffer, where $\alpha_p$ is the highest alpha of the priority. As a result, the overall buffer occupancy is also upper-bounded, as shown in Eq.4 where $\alpha_L$ and $\alpha_H$ are the maximum $\alpha$ values of the classes of high and low priorities respectively. Observe that the maximum aggregate buffer allocation of PLASTICINE is independent of the number of active 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.

Other than bounding allocation, dividing by $N_p$ offers weighted fairness among classes of the same priority. Namely, 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 low priority, she can do so by assigning higher $\alpha$ to this class.

$$B_{oc}(t) \leq \frac{B \cdot (\alpha_L + \alpha_H)}{1 + (\alpha_L + \alpha_H)}$$ (4)

$\gamma_c(t)$ reduces transient state’s duration  PLASTICINE allocates buffer to each queue proportionately to its dequeue 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, PLASTICINE splits it to queues proportionately to their service rate, effectively minimizing the time it takes for the buffer to be emptied. In effect, PLASTICINE reduces the time needed to transition from one steady-state allocation to another.

**4.2 PLASTICINE benefits**

Having explained PLASTICINE’s properties we discuss how those affect PLASTICINE in performance metrics.

**PLASTICINE improves throughput and reduces queuing delays** While PLASTICINE bounds the amount of buffer used, it maximizes its effectiveness to achieve higher aggregated throughput. Previous research has shown that TCP benefits from buffer size proportionate to the capacity of the bottleneck link \[16, 14, 21\]. Reflecting this observation to the buffer management scheme PLASTICINE multiplies the thresholds with the queues’ dequeue rate, effectively benefiting throughput while being agnostic to the scheduling algorithm used. PLASTICINE allocates on average less buffer than DT. Indeed, comparing Eq.3 with Eq.1 we observe that the added factors decrease the allocated buffer. As a result, PLASTICINE keeps queuing delays lower than DT and the buffer ready to absorb bursts.

**PLASTICINE guarantees the absorption of a given burst.** A burst is characterized by its incoming rate $r$ (normalized) and duration $t$. Whether a burst will be absorbed depends on: (i) its incoming rate ($r$); (ii) the state of the buffer at its arrival (steady state); and (iii) the buffer’s ability to dequeue fast (transient state). We can configure PLASTICINE to have two types of guarantees.

First, PLASTICINE can guarantee that a burst of a given incoming rate will be absorbed without any transient losses. Concretely, the incoming traffic will provably occupy the fair steady-state buffer space that corresponds to its ports before experiencing any drops. Indeed, Eq.5 shows the maximum $\alpha$ with which the low-priority classes would need to be configured to allow a burst with incoming rate of $r$ to pass. Please refer to Appendix B5 for the full proof.

$$\alpha_L \leq \frac{1}{r-2}$$ (5)
Second, owing to its strategic steady-state and transient-state allocation, PLASTICINE can guarantee that a burst of a given incoming rate \( r \) and duration \( t \) will be fully absorbed. Concretely, PLASTICINE guarantees that such a burst will provably experience no drops if \( \alpha_L \) and \( \alpha_H \) i.e., the maximum \( \alpha \) values of high and low priorities respectively are set according to Eq. 6 and 7. Observe, that neither is dependent on the number of queues occupying the buffer or their type. Please refer to the Appendix B.5 for the full proof.

\[
\alpha_L \leq \frac{B}{(r-1)\cdot t} - 1 \tag{6}
\]

\[
\alpha_H > \frac{1}{(r-1)\cdot (1+\alpha_L)} - \frac{r-2}{r} \tag{7}
\]

Observe, that traditional buffer management schemes can only provide such guarantees by statically reserving buffer space for a given burst. Even DT, which is assumed to be more dynamic, would have to limit all other priorities extremely, effectively assuming the worst-case scenario all the time to provide burst-tolerance guarantees.

5 PLASTICINE’s Hardware Design

Implementing any buffer management algorithm on a Protocol Independent Switch Architecture (PISA) device is challenging mostly because the Traffic Manager, responsible for managing the buffer, is not programmable [28]. Resorting to the ingress and egress pipelines for implementing PLASTICINE entails three challenges:

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 [28].

C2 Calculating PLASTICINE’s thresholds requires aggregated metrics over multiple queue lengths, e.g., remaining buffer or number of congested queues. Yet, accessing multiple values of a single memory block is not possible in PISA switches [10,11].

C3 Calculating PLASTICINE’s thresholds requires floating-point operations, which is traditionally not supported by PISA switches.

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

PLASTICINE’s high-level design We built PLASTICINE upon five main components: four register arrays and two Match & Action (M&A) table as shown in Fig. 7. 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, namely before the Traffic Manager accesses the packets. The aforementioned state includes the instantaneous remaining buffer (Remaining) the number of controlled queues per port and priority (N.Port, N.Priority) as well as the queues’ length (Q.length). 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 PLASTICINE-related fields. Finally we use another M&A table (Shift) to approximate the required floating-point operations.

Packet’s journey Upon arrival (1), a packet’s destination and priority tag are matched against the Routing table to multiple action parameters which are stored as metadata fields: namely an \( \alpha \) value and three indexes corresponding to: (i) the queue’s length in the Q.length array; a priority’s counter for congested queues in N.Priority and (ii) a port’s counter for active queues in N.Port. These indexes are used to read the relevant information about the state of the buffer (remaining buffer, buffer occupied by specific 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 queue length of its queue to the Q.length array in the egress (7).

Queue lengths available to the ingress pipeline The length of any given queue is only available as a metadata field for packets that have been enqueued in this queue, thus while they traverse the egress pipeline. Yet, PLASTICINE requires visibility of queue lengths in the ingress. To address this, we transfer queue-length information from the egress to the ingress in two steps. First, we create a register array, which resides in the egress pipeline and 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 PLASTICINE’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. 7. 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. Syncing state between ingress and egress using special packets have been used before in [28]. While SYNC packets need to be sent frequently to allow real-time visibility over the queue lengths, the overhead is minimal as the traffic generator of the device itself can create them, and they use a special pipe that is reserved for recirculation.

Calculating aggregated metrics Other than the length of the queue of interest, calculating PLASTICINE’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. 3. 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, even after calculating this meta-information, the latter 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, we use the SYNC that copy the queue lengths from the egress (as described above) to update the remaining buffer as they anyway traverse all indexes. Each such packet updates the single-value Remaining register array. Second, we use SYNC to count the active queues per port, which is equivalent to the normalized dequeue rate per queue given the scheduling algorithm. To that end, each SYNC packet updates a single port’s count with the number of queues above a threshold. Finally, we use SYNC to count the controlled queues per priority. All SYNC 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).

Approximating floating point operation Even after having all required information available, PLASTICINE needs to multiply the remaining buffer with other factors (i.e., the reverse of the number of congested queues $\frac{1}{N_{Q}}$, the reverse of the number of active queues per port $\frac{1}{N_{P}}$ and the $\alpha$) 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, of course, not done in the data-plane. Instead, we pre-calculate it for all possible values and store all the results using match-action rules matching on three values $\alpha$, $N_{P}$, and $N_c$. 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 $\alpha$ (8 for Tofino), and a few decades congested queues at most.

6 Evaluation

<table>
<thead>
<tr>
<th>Single-queue</th>
<th>Burst Abs (%)</th>
<th>QCT (ms)</th>
<th>FCT (ms)</th>
<th>Throughput (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC+TCP</td>
<td>81</td>
<td>24</td>
<td>8</td>
<td>91</td>
</tr>
<tr>
<td>PC+DCTCP</td>
<td>86</td>
<td>20</td>
<td>12</td>
<td>91</td>
</tr>
<tr>
<td>CS+TCP</td>
<td>40</td>
<td>114</td>
<td>121</td>
<td>87</td>
</tr>
<tr>
<td>CS+DCTCP</td>
<td>81</td>
<td>26</td>
<td>29</td>
<td>91</td>
</tr>
<tr>
<td>DT+TCP</td>
<td>31</td>
<td>149</td>
<td>119</td>
<td>92</td>
</tr>
<tr>
<td>DT+DCTCP</td>
<td>42</td>
<td>87</td>
<td>19</td>
<td>91</td>
</tr>
<tr>
<td>IB+TCP</td>
<td>37</td>
<td>106</td>
<td>40</td>
<td>89</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Multi-queue</th>
<th>Burst Abs (%)</th>
<th>QCT (ms)</th>
<th>FCT (ms)</th>
<th>Throughput (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC+TCP</td>
<td>78</td>
<td>29</td>
<td>13</td>
<td>87</td>
</tr>
<tr>
<td>PC+DCTCP</td>
<td>79</td>
<td>28</td>
<td>16</td>
<td>87</td>
</tr>
<tr>
<td>CS+TCP</td>
<td>41</td>
<td>110</td>
<td>151</td>
<td>84</td>
</tr>
<tr>
<td>CS+DCTCP</td>
<td>64</td>
<td>46</td>
<td>71</td>
<td>87</td>
</tr>
<tr>
<td>DT+TCP</td>
<td>50</td>
<td>59</td>
<td>122</td>
<td>88</td>
</tr>
<tr>
<td>DT+DCTCP</td>
<td>69</td>
<td>38</td>
<td>73</td>
<td>87</td>
</tr>
<tr>
<td>IB+TCP</td>
<td>48</td>
<td>62</td>
<td>8</td>
<td>86</td>
</tr>
</tbody>
</table>

Figure 8: Summary of results: PLASTICINE (PC), shown in the first 2 rows, achieves the best performance for high-priority traffic across all alternatives (as shown in columns 2 and 3), while not sacrificing lower priority traffic (as shown in columns 4 and 5). Greener cells illustrate better performance.

In our evaluation, we answer two main questions:

Q1 How does PLASTICINE compare against other buffer management schemes?

Q2 Is PLASTICINE useful if DCTCP already controls queues, effectively improving buffer utilization?

We show that Q1: PLASTICINE achieves higher burst absorption, lower FCT and on-par throughput compared to all other buffer management schemes; and Q2: PLASTICINE demonstrates 24% – 72% lower QCT, 57% – 82% lower 99th percentile FCT compared to DT with DCTCP.

We first describe the methodology §6.1 before we elaborate on our results §6.2.

6.1 Methodology

We evaluate PLASTICINE by comparing it with three alternative buffer management algorithms, namely (i) Complete Sharing (CS), which allows queues to grow arbitrar-
We evaluate PLASTICINE’s performance in a Leaf Spine topology \footnote{Our implementation of different buffer-management in ns-3 will be made available online.} of two leaves and two spines with four links connecting each pair of leaf and spine. We use ECMP to load-balance traffic across uplinks. We set the buffer size to 1MB and the bandwidth per port to 1Gbps. Each leaf is connected to 40 servers (oversubscription of 5). We set $R_T T=200\mu s$.

**Traffic mix** We generate traffic of two types: \(i\) web search (realistic workload based on traffic measurements in deployed datacenters); and \(ii\) query traffic (models the user queries of a web application backend). For the former, we used the flow size distribution from \footnote{High load is required to test the buffer management capabilities.} and tuned the mean of Poisson inter-arrivals such that a 90\% load is achieved. For the latter, we assume that a query arrives at each server according to a Poisson process with mean 1 query/second following \footnote{6}. Each query consists of a server attached to a leaf requesting a `Query-Size` file from all the servers connected to the other leaf. Each request is then responded by 40 servers, each transmitting $\frac{1}{40}$ of the file. Each pair of servers has a persistent TCP connection as in \footnote{6}. A query is completed when the requester receives the `Query-Size` file.

**Deployment Scenarios** We consider two deployment scenarios: `single-queue`; and `multi-queue`, which differ on the number of output queues used per port and the entity that marks packets. In the `single-queue` deployment scenario, we assume single-queue per port. Traffic is not marked with priority tags by end-hosts, but a flow classifier similar to the elephant trap that IB uses is available in the device. Unlike IB and PLASTICINE, CS and DT cannot use such information, because the former only employs a
single threshold per queue and the latter does not enforce any threshold anyway.

In the multi-queue scenario, we assume five queues per port and Round Robin scheduling. Incoming packets are mapped to a queue according to a tag they carry. The query traffic is marked with a high-priority tag and the web-search traffic with 4 equally low-priority tags. Tags are added by the servers uniformly at random. 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.

Configuration We configure PLASTICINE with $\alpha_L = 0.5$ and $\alpha_H = 20$ for short and long flows respectively in single-queue scenario; and for low-priority and high-priority traffic respectively in multi-queue scenario; We configure DT with $\alpha_L = 0.5$ in single queue scenario; and with $\alpha_L = 0.5$ and $\alpha_H = 20$ for low-priority and high-priority traffic respectively in multi-queue scenario. We configure IB with $\alpha_L = 0.5$ and uses headroom for short flows in both the single and multi-queue scenarios. IB also uses a separated priority queue for short flows in multi-queue scenario. Finally, IB uses Approximate Fair Dropping. When DCTCP is used, all queues are RED [15] with min and max threshold set to $20$ ($K = 20$) following the recommendations in [17]. All other queues are DropTail When TCP is used queues are DropTail, except for IB. TCP minRTO is set to 100ms.

Metrics We use different performance metrics for our two traffic mixes to match their nature. For query traffic, we evaluate (i) burst absorption; and (ii) 99th percentile Flow Completion Time (FCT); For web-search traffic, we measure (i) 99th percentile Completion Time (QCT); For web-search traffic, we measure (i) 99th percentile Flow Completion Time (FCT); and (ii) throughput. Alongside those metrics we report the average and the 99-th percentile buffer occupancy. We perform 10 experiments and report average values.

6.2 Results

Fig. 8 summarizes our results in two tables corresponding to the single- and the multi-queue deployment scenario. Each line in the table corresponds to a combination of

---

8 Setting $\alpha = 20$ would approximate CS’s behavior, that is no drops until buffer is full.

9 Concretely, we measure throughput at the up-links without differentiating across traffic type. Yet, since only web-search contains long flows we consider it a metric for this type.
buffer management algorithm with a TCP version. Each column corresponds to a performance metric. We color the cells according to how good the corresponding algorithm is in the particular metric compared to the alternatives. Green represents the best performance, while red is the worst. Values in each cell correspond to the median of the values we plot in Fig. 9[10]. Next, we elaborate on these results.

**Plasticine** is the most effective algorithm for absorbing bursts or isolating high-priority traffic. We first evaluate burst absorption, namely the percentage of queries in our traffic-mix that experiences zero loss. In the single-queue deployment scenario, we find that **Plasticine** achieves 81% higher burst absorption than DT when TCP is used and 51% when DCTC is used. Indeed, Fig. 9a shows the burst absorption as a function of the query size for the single-queue scenario for different buffer management algorithms in conjunction with TCP or DCTCP. Query size is expressed relatively to the buffer. Fig. 9a shows the burst absorption for the multi-queue scenario.

**Plasticine**’s high burst absorption results in 83% (77%) lower Query Completion Time (QCT) compared to DT with TCP (DCTCP), as we observe in Fig. 9b. In Fig. 9c, we summarize the QCT in the single-queue scenario for the different schemes we consider. The QCT for the multi-queue scenario can be seen in Fig. 9e.

**Plasticine**’s performance benefits in the single-queue scenario come from the prioritization of bursts over long flows of the same queue achieved using different thresholds for the same queue. In the multi-queue scenario (where bursts are already marked as high-priority traffic), **Plasticine** improves performance by keeping just enough buffer unoccupied to guarantee zero transient losses for high-priority traffic.

**Plasticine** does not sacrifice low-priority traffic. While **Plasticine** prioritizes bursts and high-priority traffic, it also achieves on-par throughput and lower 99th percentile FCT for short flows belonging to the low-priority traffic compared to almost all other schemes.

Fig. 9c, 9f show the 99th percentile FCT of flows with < 100KB in the single-queue and multi-queue scenario respectively. **Plasticine** with TCP achieves 93% (89%) lower 99th percentile FCT than DT with TCP in the single-queue (multi-queue) scenario. **Plasticine** achieves this improvement through keeping average buffer utilization low, effectively achieving low queuing delay. Notably, IB achieves the best FCT for the background short flows in the multi-queue scenario (Fig. 9f). That is the case since IB puts short flows in a separated high-priority queue even when they carry a low-priority tag.

Fig. 10c, 10f show the average uplink throughput for the single-queue and multi-queue scenario respectively. Throughput is mostly due to the long flows of low-priority traffic. **Plasticine** achieves on-par throughput, demonstrating that **Plasticine** (unlike IB) does not prioritize high-priority traffic by sacrificing low-priority traffic.

When it comes to buffer, less is more. Complete Sharing (CS) with TCP keeps the buffer 60-80% utilized on average as we observe in Fig. 10a and 100% in at least 1% of the time, as we observe in Fig. 10b where we show the 99th percentile buffer occupancy of all algorithms in the single-queue scenario. While CS with TCP uses more buffer than the any other combination, its performance is the worst in all metrics, including throughput. Indeed, CS is equivalent to no buffer management, and with no marking scheme to limit the queues (TCP), there is no control over buffer allocation. Uncontrolled buffer allocation leads to unfair and even harmful buffer overutilization resulting in high queuing delays, unfair/uncontrolled allocation and even throughput loss. In fact, looking at Fig. 10a and Fig. 10b we observe a correlation between low average buffer utilization and good performance. Also, looking at Fig. 10b and Fig. 10e we observe that good performance is correlated with high tail buffer occupancy which increases as the burst size increases, as we observe in the cases of **Plasticine** and DCTCP.

**Plasticine allocates buffer only when it is useful.** Notably, **Plasticine** achieves high-performance benefits by a cautious but strategic allocation of the buffer. In particular, while **Plasticine** uses the least buffer on average across the alternatives (Fig. 10c), it also rarely fully utilizes it, as we observe in Fig. 10b where its utilization approaches CS. Notably, **Plasticine**’s 99th percentile allocation increases as the burst size increases in both single- and multi-queue scenarios (Fig. 10e, Fig. 10f) clearly showing a strategic prioritization of burst/high-priority traffic.

While the use of multiple priority queues offer isolation, it cannot achieve the benefits of **Plasticine**. DT and CS demonstrate significantly improved burst absorption capabilities in the multi-queue scenario, as we observe by comparing Figs. 9a, 9e with Figs. 9a, 9e. This is expected as isolating the (pre-marked) high-priority traffic to a single queue reduces interactions between high and low-priority traffic. At the same time, in the case of DT, the higher α for high-priority traffic gives bursts a much better performance. Still, this improved performance comes at a cost for the low-priority traffic whose short flows experience higher FCT in the multi-queue scenario than in the single-queue one. This is the case, as short flows share the same queues with long flows of low-priority traffic in both scenarios.

Unlike **Plasticine**, DCTCP cannot deal with buffer pressure. DCTCP, when combined with Complete Sharing, demonstrates burst absorption similar to **Plasticine**, at least in the single-queue scenario. That is because DCTCP significantly reduces the amount of buffer that
long flows take on average (Fig 10a), by early marking packets on the single queue per port. As a result, when a burst hits the device, there is enough buffer unoccupied, which can be taken by the burst as there is no limit by the buffer management algorithm (CS) (Fig 10b). Still, the lack of a buffer management algorithm becomes obvious in the multi-queue scenario. As DCTCP has no visibility on the overall buffer usage, it cannot prioritize a queue over another, resulting in significantly worse burst absorption capabilities compared to PLASTICINE. In other words, DCTCP cannot prevent the buffer pressure that the low-priority traffic creates to disturb the high-priority traffic. In particular, CS with DCTCP increases QCT by, on average, $\sim 58$ms compared to PLASTICINE with TCP.

DCTCP and/or priority queuing cannot compensate for DT’s inefficiencies. While DCTCP and IB visibly improve the performance of DT, the latter have notable limitations. First, DT’s burst capabilities in the single-queue scenario are devastating, as we observe in Fig. 9a. In particular, DT increases QCT by 87% (even when combined with DCTCP) compared to PLASTICINE. As DT employs a single threshold per queue, it cannot distinguish bursts from low-priority traffic concerning buffer usage. Surprisingly, DT performs worse than CS when both are combined with DCTCP in the single-queue scenario. This is the case as DT always keeps some space unoccupied. Even in the multi-queue scenario though (Fig. 9d), when packets belonging to a burst are marked and use a dedicated queue, DT, even combined with DCTCP, achieves 23% (26%) worst QCT compared to PLASTICINE with DCTCP (TCP). This demonstrates the unbounded nature of DT. Indeed, DT allocates a large percentage of the buffer on low-priority queues on aggregate (Fig. 10a, 10b). As a result, when a burst arrives, the buffer cannot dequeue fast enough to avoid transient losses regardless of how high the $\alpha_p$ that corresponds to it is. Finally, PLASTICINE achieves $\approx 2$ times lower FCTs (Fig. 9c). DT allocates buffer per queue, ignoring their expected dequeue rate, effectively increasing the queuing delay.

**PLASTICINE is useful beyond traffic classes** PLASTICINE’s strategic allocation facilitates efficient sharing of the buffer across other types of traffic. For example, we have evaluated PLASTICINE in two additional scenarios: (i) when TCP and DCTCP traffic coexist; and (ii) when DC and WAN traffic coexist. We moved the results to the Appendix A due to space limitations.

7 Case study

We implement our hardware design §5 on a Barefoot Tofino Wedge 100BF-32X to verify that it works on real hardware and can improve performance.

Our testbed (Fig. 11a) includes one Tofino switch and two servers (S1 and S2). S1 connects to the Tofino via port $p_1$, while S2 via ports $p_2$ and $p_3$. Each port is mapped to 8 queues with aggregate bandwidth 80Kbps 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. All queues share the same buffer pool limited to 9000 cells. We did not use all the buffer space provided in a Tofino due to the low-traffic rate. We generate UDP flows from S1 to S2 on port $p_2$. We also generate 100 TCP flows 8KB in size from S2 $p_3$ to S1. We run the experiment once with DT and once with PLASTICINE. Fig. 11b presents the CDF of FCT results. PLASTICINE limits the buffer used by low-priority UDP flows, allowing short flows of high-priority to achieve good performance. As a result, PLASTICINE achieves at least 50ms smaller FCTs.

8 Related Work

Buffer management has been studied in the context of ATM networks [25, 24, 17, 19], yet, these approaches are only applicable to Call Admission Control (CAC). Pushout based algorithms such as [31], [29] offer prioritized packet buffering. However, pushout mechanisms add high complexity to hardware and are considered impractical [9]. Recent works such as [27, 8] only introduce the notion of prioritizing bursts without changing the buffer allocation scheme of Dynamic Thresholds (DT) [12]. As a result, they face similar problems as that of DT.

9 Conclusion

In this paper, we demonstrate the inefficiencies of today’s most common buffer management algorithm: Dynamic Thresholds, experimentally and analytically. We present PLASTICINE, a novel algorithm that offers provable burst-tolerance guarantees, while not statically allocating buffer. We show that PLASTICINE outperforms all other buffer management techniques even when combined with DCTCP. Finally, we show that PLASTICINE’s design is practical by implementing it on a Barefoot Tofino.
References


### A.1 TCP and DCTCP interactions

#### A.1.1 Setup

We evaluate PLASTICINE performance in a mixed congestion control traffic pattern. Particularly, we generate the same traffic pattern as in Sec. 6 (Web-search and Query traffic) with a change that, a flow is either DCTCP or TCP chosen uniformly at random. At all the network devices, there exists a mechanism which splits DCTCP and TCP flows into two separate queues, ECN enabled and tail drop queues respectively. The aim of this evaluation is to find whether mixed CC effects the performance of PLASTICINE and how other buffer management schemes work in this scenario.

#### A.1.2 Results

**PLASTICINE effectively isolates TCP and DCTCP, achieves high burst absorption** We find that, although DCTCP consumes less buffer, TCP flows tend to overuse the buffer in case of DT and even worse in case of Complete Sharing as observed in Fig. [13]. PLASTICINE dynamically allocates buffer to TCP flows to achieve high throughput. At the same time, bounds the overall allocations leaving enough buffer for bursts and short flows. Finally, Fig. [13] shows that PLASTICINE is unaffected even when TCP flows coexist with DCTCP.

### A.2 WAN and DC traffic interactions

Finally, we evaluate our algorithm PLASTICINE in a setup where wide-area-network (WAN) and data center (DC) network traffic coexist. In particular, such a traffic mix majorly has a single point of congestion at the top of the rack switch.

#### A.2.1 Setup

We set the intra-datacenter RTT to 200µs and WAN RTT to 10ms. All the links in the network are 1Gbps except the links connecting to WAN which are 10Gbps. We use DCTCP in all the evaluation from here. WAN and DC are split into separate queues at the switches and a marking threshold of 200 (and buffer management $\alpha = 0.75$) and 20 packets (and buffer management $\alpha = 1$) is set for each queue respectively. A high alpha value $\alpha = 1024$ is used for all short and bursty flows in the case of PLASTICINE. We evaluate in two scenarios where WAN to DC traffic ratio is 5:1 and 1:5 with an overall load of 80% on the uplinks. We use weighted round-robin scheduling for the two queues whose weights are set to the same ratio of traffic mix. Finally, the foreground query traffic is part of data center traffic i.e., rack-to-rack. In this setup, we are interested in the abilities of buffer-management to sustain good performance for data center traffic while not losing throughput of WAN. In other words, isolation of the two traffic type. We use the same performance metrics as in the previous sections to compare different buffer management schemes.

#### A.2.2 Variation across Burst sizes

**PLASTICINE better isolates WAN and DC traffic, achieves the best burst absorption** First, we observe from Fig. [14] that PLASTICINE achieves significantly high burst absorption $\approx 40\%$ greater than Complete Sharing (CS) and $\approx 80\%$ greater than Dynamic Threshold (DT). In particular, CS and DT perform much worse than in the pure DC scenario (Fig. [9a]). On the other hand, PLASTICINE performs well effectively isolating different traffic types. PLASTICINE dynamically allocates buffer to WAN traffic and bounds them to a maximum of $\alpha \times \frac{B}{1 + \alpha} \approx 40\%$ of buffer, where $\alpha = 0.75$. At the same time PLASTICINE bounds the long flow buffer occupancy of data center traffic to a maximum of $\alpha \times \frac{B}{1 + \alpha} \approx 50\%$ of buffer, where

---

10 We only evaluate using DCTCP as congestion control, 1Gbps bottleneck and 10ms WAN RTT i.e., 1.25MB of BDP. We use a total buffer size of 1MB. In this case, traditional TCP performs poorly as Buffer < BDP even for single port. However DCTCP enables lower buffer requirement with a marking threshold of $\approx 200KB$ in this case. For higher capacity links and today’s available buffer sizes often experience Buffer < BDP and how to reduce the buffer requirement is a hot-topic in research [25] which falls under congestion-control research and is out of scope of this paper.
Figure 13: Evaluation using TCP and DCTCP mixed traffic pattern showing how burst absorption and flow completion times are severely affected in the case of Dynamic Threshold and Complete Sharing while PLASTICINE successfully limits TCP buffer consumption enabling high burst absorption and low flow completion times for short flows.

Figure 14: In WAN + DC traffic-mix, % of queries that experience zero loss i.e., burst absorption, FCTs of short flows showing how DT and Comp. Sharing suffer from the presence of WAN traffic while PLASTICINE still achieves high burst-absorption with a 40 + % gain and lowers flow completion times without sacrificing throughput.
Figure 15: Buffer occupancy and throughput comparisons in WAN+DC traffic-mix. PLASTICINE successfully reduces the average buffer utilizations, strategically utilizes the 99-th percentile buffer space as the burst sizes increase and achieves on-par throughput in both 5:1 and 1:5 WAN/DC traffic mix ratios.

$\alpha = 1$. Finally, PLASTICINE allocates as much buffer as need to short and bursty flows enabling a high burst absorption without interference from WAN traffic.

Analogous to burst absorption, similar performance order is seen in Query completion times in Fig. [14]. High burst absorption enables better query completion times. As a result, PLASTICINE achieves 40ms and 100ms lower query completion times compared to CS and DT respectively with 5:1 WAN/DC traffic mix. In the scenario with lower volume of WAN traffic, PLASTICINE still outperforms convention buffer management algorithms with 30ms and 100ms lower query completion times compared to CS and DT respectively.

**PLASTICINE enables lower FCTS for short flows even under WAN/DC interactions** We observe from Fig. [14] that PLASTICINE outperforms in flow completion times for short flows in data center traffic. In particular CS performs very poorly in flow completion times although it performs well in burst absorption and query completion times. This is due to the fact that CS allows each queue to take up as much buffer as available and has no performance guarantees. On the other hand, PLASTICINE’s isolation guarantees and low average buffer occupancies enable low queuing delays and minimal packet loss for short flows.

**PLASTICINE strategically minimizes buffer utilization, does not sacrifice throughput** Finally, we show the buffer occupancy and achieved aggregate uplink throughput in Fig. [15]. We observe that PLASTICINE enables a lower average buffer utilization on one hand and dynamically takes more 99-th percentile buffer to accommodate more buffer on the other hand. In particular, we observe that although CS performs good enough in burst absorption and query completion times, it experiences throughput loss when WAN traffic dominates in the traffic-mix (5:1 WAN/DC).

To conclude, PLASTICINE effectively isolates the interactions between WAN and DC traffic, achieves better performance for data center flows at the same time not losing throughput in WAN.
B Analysis

B.1 Assumptions

The analysis is based on a fluid model where packet (bits) arrivals and departures are assumed to be fluid and deterministic. A switch with arbitrary number of ports with arbitrary number of queues per port is assumed. In particular, each port has only one queue per class (defined in Sec. 2).

- $B$: Total shared buffer space of switch.
- $Q(t)$: Instantaneous occupied buffer space at time $t$
- $\alpha_c$: A parameter for buffer-management algorithm, for each class.

B.2 PLASTICINE

PLASTICINE works based on two-levels of hierarchy i.e class and priority. The general notion of class remains same i.e., each class is associated with a separate queue at each port. In addition, PLASTICINE requires the classes to be mapped to priorities. 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.

PLASTICINE buffer-management algorithm requires an $\alpha$, parameter per class. The buffer-allocation is based on threshold calculations per queue. In particular, the threshold of a queue at port $i$, of class $c$ and belonging to a priority $p$ is calculated by PLASTICINE algorithm as,

$$Threshold = (\alpha) \times (FairShare) \times (Norm.DequeueRate) \times (remaining)$$

i.e,

$$T^i_c(t) = \alpha_c \cdot \beta_p(c) \cdot \gamma^i_c(t) \cdot (B - Q(t))$$

where, $\beta_p(c)(t)$ is the inverse of the total number of congested queues of priority $p(c)$ (to which the class $c$ belongs to) at time $t$ i.e., $\frac{1}{\sum_{i \in P} N_i}$ and $\gamma^i_c(t)$ is the normalized dequeue rate (or normalized service capacity) of the queue at time $t$.

Observe that, $\beta_p(c)(t)$ remains same for all the priorities belonging to a group and can be expressed as $\beta_P(t)$ where $P$ denotes the priority $p(c)$.

Here after for simplicity $\omega^i_c(t)$ is defined as,

$$\omega^i_c(t) = \alpha_c \cdot \beta_P(t) \cdot \gamma^i_c(t)$$

Key properties of Omega:

$$\sum_{i} \sum_{c \in P} \omega^i_c(t) = \beta_P(t) \cdot \sum_{i} \sum_{c \in P} \alpha_c \gamma^i_c(t)$$

If, all the queues of all classes of priority $P$ share same alpha parameters (say $\alpha_P$) and have same normalized dequeue rate at time $t$ (say $\gamma_P$), then Eq. [11] reduces to,

$$\sum_{i} \sum_{c \in P} \omega^i_c(t) = \alpha_P \cdot \gamma_P$$

Further, if $\gamma_P = 1$,

$$\sum_{i} \sum_{c \in P} \omega^i_c(t) = \alpha_P$$

In general there exists a limit given by,

$$\sum_{i} \sum_{c \in P} \omega^i_c(t) \leq \max_{c \in P}(\alpha_c) = \alpha_{max}$$

We will see later, how these properties enable PLASTICINE to achieve certain isolation and burst-tolerance guarantees.
B.3 Steady-State Analysis

In this subsection, it is assumed that load-conditions remain stable and a steady-state of buffer is achieved. Following this assumption, all the equations in this subsection are expressed without the time variable. Under this state, the queue lengths remain stable at less than or equal to the corresponding threshold. For simplicity, it is assumed that all the queues-lengths are at their respective thresholds. Then the total buffer occupancy can be expressed as,

\[ Q = \sum_i \sum_c Q^i_c \quad (15) \]

From the assumption that the queue-lengths are equal to their thresholds, using Eq. 9 and Eq. 10,

\[ Q = \sum_i \sum_c \omega^i_c \cdot (B - Q) \quad (16) \]

Solving for \( Q(t) \) gives,

\[ Q = \frac{B \sum_i \sum_c \omega^i_c}{1 + \sum_i \sum_c \omega^i_c} \quad (17) \]

where \( \omega^i_c \) is given by Eq. 10

Using, Eq. 17, the remaining buffer space \( B - Q(t) \) can be expressed as,

\[ \text{Remaining} = \frac{B}{1 + \sum \sum \omega^i_c} \quad (18) \]

Under steady-state, from Eq. 18 and Eq. 9 the threshold of a queue at port \( i \) and of class \( c \) is given by,

\[ T^i_c = \frac{B \cdot \omega^i_c}{1 + \sum \sum \omega^i_c} \quad (19) \]

Key properties of Remaining Buffer space:

Using the notion of groups, Eq. 18 can be expanded as,

\[ \text{Remaining} = \frac{B}{1 + \sum_p \sum_{c \in p} \omega^i_c} \quad (20) \]

Using the maximum limit of \( \omega \) property from Eq. 14,

\[ \text{Remaining} \geq \frac{B}{1 + \sum_p \omega^i_{c_{\text{max}}}} \quad (21) \]

This key property shows how the remaining buffer space and buffer-occupancy are bounded.

For example, lets consider, there exists two classes \( c_0 \) and \( c_1 \) with alpha parameters \( \alpha_0 \) and \( \alpha_1 \), each of which belongs to a separate priority. The remaining buffer space, irrespective of the number of queues and their dequeue rates, Eq. 21 can be expressed as,

\[ \text{Remaining} \geq \frac{B}{1 + \alpha_0 + \alpha_1} \quad (22) \]

B.4 Transient-State Analysis

Given a steady-state of buffer assuming that all the queue lengths are controlled by a threshold, when traffic to empty queues appear, load conditions change. The new queues increase in length creating changes in the remaining buffer. As a result, the thresholds and queue lengths under go a transient state. Due to the appearance of new queues, \( \omega^i_c \) of some of the existing queues get affected due to the changes in \( \beta_{c(p)} \) (number of queues belonging to a class \( c \) of priority \( p \)) and \( \chi^t_c \) (normalized dequeue rate). Let \( G_e \) denote the set of queues whose \( \omega^i_c \) gets affected and \( G_{ne} \) denote the set of queues whose \( \omega^i_c \) does not get affected. Note that the \( \omega^i_c \) values of \( G_e \) only reduce. (It is not possible that \( \omega^i_c \) increases due the appearance of a new queue). For simplicity lets denote the queue at port \( i \) and of class \( c \) with ordered pairs as \((i,c)\). The set of ordered pairs of existing queues is denoted as \( S_{old} \). The ordered pairs of new queues that trigger transient state are denoted as \( S_{new} \). Observe that \( S_{old} = C_{ne} \cup C_{e} \).
The arrival rate of traffic at each new queue is denoted by \( r \) and the arrival process is fluid and deterministic. At \( t = 0 \),

\[
T_c^i(0) = \frac{\omega_c^i \cdot B}{1 + \sum_{\forall(i,c) \in S_{old}} \omega_c^i} 
\]

(23)

\[
Q_c^i(0) = \begin{cases} 
\frac{\omega_c^i \cdot B}{1 + \sum_{\forall(i,c) \in S_{old}} \omega_c^i}, & \text{for } \forall(i,c) \in S_{old} \\
0, & \text{for } \forall(i,c) \in S_{new} 
\end{cases}
\]

(24)

At \( t = 0^+ \), \( \omega_c^i \) of \( G_c \) change and remain same for the entire duration of transient state. At the same time, the \( \omega_c^i \) of \( G_{ne} \) remain unchanged. Hence such changes are assumed to happen and the time variable is dropped for \( \omega_c^i \) in the equations. From Eq. 9, the rate of change of thresholds and queue lengths can be expressed as follows,

\[
\frac{dQ_c^i(t)}{dt} = \omega_c^i \cdot \sum_{\forall(i,c) \in S_{old}} \frac{dQ_c^i(t)}{dt} + \omega_c^i \cdot \sum_{\forall(i,c) \in S_{new}} (r - \gamma_c) 
\]

(26)

It can be proved by contradiction that \( \frac{dT_c^i(t)}{dt} \leq 0 < r - \gamma_c \). Solving Eq. 25 and Eq. 26 for \( t = 0^+ \),

\[
\left( \frac{dT_c^i(t)}{dt} \right)_{(t=0^+)} = -\omega_c^i \cdot \left( \sum_{\forall(i,c) \in G_c} \max[-\gamma_c, \min[\frac{dT_c^i(t)}{dt}, r - \gamma_c]] \right) - \omega_c^i \cdot \sum_{\forall(i,c) \in G_{ne}} (r - \gamma_c) 
\]

(27)

Recall that \( S_{old} = G_c \cup G_{ne} \). All the queues belonging to \( G_c \), will experience a change in their \( \omega_c^i \) values at \( t = 0^+ \) resulting in their queue-lengths greater than threshold. As a result, the rate of change of their queue lengths is their corresponding dequeue rates. Eq. 27 can then be expanded as,

\[
\left( \frac{dT_c^i(t)}{dt} \right)_{(t=0^+)} = -\omega_c^i \cdot \left( \sum_{\forall(i,c) \in G_c} \max[-\gamma_c, \frac{dT_c^i(t)}{dt}] \right) - \omega_c^i \cdot \sum_{\forall(i,c) \in G_{ne}} (r - \gamma_c) 
\]

(28)

From Eq. 28, arrival rate of traffic in new queues i.e \( r \) can be expressed as,

\[
r = \frac{\sum_{\forall(i,c) \in S_{new}} \gamma_c + \frac{dT_c^i(t)}{dt}}{\sum_{\forall(i,c) \in S_{new}} \omega_c^i} - \frac{\sum_{\forall(i,c) \in S_{new}} \gamma_c \cdot \frac{dT_c^i(t)}{dt}}{\omega_c^i \cdot \sum_{\forall(i,c) \in G_{ne}} (r - \gamma_c)} 
\]

(29)

By applying summation across \( \forall(i,c) \in G_c \) over Eq. 28 (will be seen later how this will be useful), \( r \) can be expressed as,

\[
r = \frac{\sum_{\forall(i,c) \in S_{new}} \gamma_c \cdot \frac{dT_c^i(t)}{dt} + \left( \sum_{\forall(i,c) \in G_{ne}} \max[-\gamma_c, \frac{dT_c^i(t)}{dt}] \right) \cdot \sum_{\forall(i,c) \in G_{ne}} \omega_c^i}{\sum_{\forall(i,c) \in S_{new}} \omega_c^i - \left( \sum_{\forall(i,c) \in G_{ne}} \omega_c^i \right) \cdot (\sum_{\forall(i,c) \in S_{new}} 1)} 
\]

(30)

Now it can be observed that the value of \( r \) influences for all \( \forall(i,c) \in G_c \), \( \left( \frac{dT_c^i(t)}{dt} \right)_{(t=0^+)} \). In other words, the value of \( r \) influences the total i.e \( \sum_{\forall(i,c) \in G_c} \left( \frac{dT_c^i(t)}{dt} \right)_{(t=0^+)} \) which is the aggregate rate at which thresholds drop for the non affected set of queues i.e \( G_{ne} \).
B.4.1 Case-1

In this case, the arrival rate \( r \) is such that, the queues belonging to \( G_{ne} \) are able to reduce in length exactly tracking the changes in their thresholds. As a result their queue-lengths remain equal to the threshold throughout the transient state i.e.,

\[
\left( \frac{dT_c^i(t)}{dt} \right)_{(t=0^+)} \geq -\gamma_c
\]

leading to,

\[
\sum_{\forall (i,c) \in G_{ne}} \left( \frac{dT_c^i(t)}{dt} \right)_{(t=0^+)} \geq \sum_{\forall (i,c) \in G_{ne}} -\gamma_c
\]

Using Eq. 31 and Eq. 32 in Eq. 30, the condition on \( r \) can be expressed as,

\[
r \leq \frac{\sum_{\forall (i,c) \in S_{new \cup G_c}} \gamma_c}{\sum_{\forall (i,c) \in S_{new}} 1} + \left( \sum_{\forall (i,c) \in G_c} \gamma_c \right) \cdot \frac{1 + \sum_{\forall (i,c) \in G_{new}} \omega_c^i}{(\sum_{\forall (i,c) \in G_{new}} \omega_c^i) \cdot (\sum_{\forall (i,c) \in S_{new}} 1)}
\]

Note that in Eq. 30 we deliberately apply summation over \( \forall (i,c) \in G_{ne} \) which can be a null set. If \( G_{ne} = \phi \), by applying summation over \( \forall (i,c) \in G_c \) in Eq. 28, \( r \) condition can be expressed as,

\[
r \leq \frac{\sum_{\forall (i,c) \in S_{new}} \gamma_c}{\sum_{\forall (i,c) \in S_{new}} 1} + \left( \sum_{\forall (i,c) \in G_c} \gamma_c \right) \cdot \frac{1 + \sum_{\forall (i,c) \in G_{new}} \omega_c^i}{(\sum_{\forall (i,c) \in G_{new}} \omega_c^i) \cdot (\sum_{\forall (i,c) \in S_{new}} 1)}
\]

For generalization, observe the “*” over the summation terms in Eq. 33. Hereafter, the convention follows that, where ever “*” appears, it means that, the summation is deliberate and can be interchanged between \( \forall (i,c) \in G_{ne} \) and \( \forall (i,c) \in G_c \) if \( G_{ne} = \phi \). All the other summations have usual meaning.

Substituting Eq 31 and Eq 32 in Eq 28 and using the result in Eq 26 gives,

\[
\left( \frac{dT_c(t)}{dt} \right)_{(t=0^+)} = -\omega_c^i \cdot \left( \sum_{\forall (i,c) \in G_c} -\gamma_c + \sum_{\forall (i,c) \in S_{new}} (r - \gamma_c) \right) \frac{1}{1 + \sum_{\forall (i,c) \in G_{new}} \omega_c^i}
\]

\[
\left( \frac{dQ_c^i(t)}{dt} \right)_{(t=0^+)} = \begin{cases} 
-\omega_c^i \cdot \left( \sum_{\forall (i,c) \in G_c} -\gamma_c + \sum_{\forall (i,c) \in S_{new}} (r - \gamma_c) \right) \\
\frac{1}{1 + \sum_{\forall (i,c) \in G_{new}} \omega_c^i} 
\end{cases} \text{, for } \forall (i,c) \in G_{new} \\
-\gamma_c \\
\text{, for } \forall (i,c) \in G_c \\
\text{, for } \forall (i,c) \in S_{new}
\]

These differential equations will be valid as long as \( Q_c^i(t) = T_c^i(t) \) for \( \forall (i,c) \in G_{ne} \) \& \( Q_c^i(t) \geq T_c^i(t) \) for \( \forall (i,c) \in G_c \) \& \( Q_c^i(t) < T_c^i(t) \) for newly created queues i.e \( \forall (i,c) \in S_{new} \). Solving these equations, using the initial conditions, Eq. 23 and Eq. 24 leads to,

\[
T_c^i(t) = \frac{\omega_c^i \cdot B}{1 + \sum_{\forall (i,c) \in S_{old}} \omega_c^i} - \frac{\omega_c^i \cdot \left( \sum_{\forall (i,c) \in G_c} -\gamma_c + \sum_{\forall (i,c) \in S_{new}} (r - \gamma_c) \right) \cdot t}{1 + \sum_{\forall (i,c) \in G_{new}} \omega_c^i}
\]
As a result their queue-lengths remain greater than the threshold throughout the transient state i.e., in this case, the arrival rate $r$ is such that, the queues belonging to $G_{ne}$ will experience zero drops i.e., no transient drops if its duration $t$ satisfies the following condition:

$$t_{1e}^i = -\frac{\omega^i_e \cdot B \cdot (1 + \sum_{(i,c) \in G_{ne}} \omega^i_c) - \omega^i_e \cdot t \cdot (\sum_{v(i,c) \in S_{old}} \omega^i_c \cdot \gamma^i_c \cdot (r - \gamma^i_c))}{(1 + \sum_{(i,c) \in S_{old}} \omega^i_c \cdot (r - \gamma^i_c) \cdot (1 + \sum_{v(i,c) \in G_{ne}} \omega^i_c) + \omega^i_e \cdot (\sum_{(i,c) \in G_{ne}} \omega^i_c - \gamma^i_c + \sum_{v(i,c) \in S_{new}} (r - \gamma^i_c)))},$$ \hspace{1cm} \text {for} \forall (i,c) \in G_{ne} \tag{38}$$

In order to offer guarantees, it is absolutely required that either $\gamma^i_c$ is constant. The reason being that there is a dependency between $\gamma^i_c$ and the number of queues of the same port using buffer, a dependency which is fundamentally impossible to evade unless $\gamma^i_c$ is constant. As a result of this assumption, $G_{c} = \phi$ and $S_{old} = G_{ne}$ and Eq. \ref{equation39} reduces to,

$$t_{1e}^i = -\frac{\alpha_H \cdot \frac{1}{N_{p(e)}} \cdot \gamma^i_c \cdot B}{(r - \gamma^i_c) \cdot (1 + \sum_{v(i,c) \in S_{old}} \frac{\omega^i_c}{\omega^i_e} \cdot \sum_{v(i,c) \in S_{new}} 1)},$$ \hspace{1cm} \text {for} \forall (i,c) \in S_{new} \tag{40}$$

We can further simplify for a case where load variations occur for High Priority whose maximum $\alpha$ value is $\alpha_H$ and the existing Low Priority in the queues have a maximum $\alpha$ value of $\alpha_L$. We can then guarantee that for an arrival rate $r$ that satisfies Case-I will experience zero drops i.e., no transient drops if its duration $t$ satisfies the following condition:

$$t_{1e}^i = -\frac{\alpha_H \cdot \frac{1}{N_{p(e)}} \cdot \gamma^i_c \cdot B}{(r - \gamma^i_c) \cdot \left(1 + \alpha_L + \alpha_H \cdot \frac{1}{N_{p(e)}} \cdot \gamma^i_c \cdot \sum_{v(i,c) \in S_{new}} 1\right)},$$ \hspace{1cm} \text {for} \forall (i,c) \in S_{new} \tag{41}$$

Observe that Eq. \ref{equation41} is independent of number of queues of Low Priority and hence it can be said that High Priority isolation can be guaranteed.

### B.4.2 Case-2

In this case, the arrival rate $r$ is such that, the queues belonging to $G_{ne}$ are unable to reduce in length in accordance with the changes in their thresholds. As a result their queue-lengths remain greater than the threshold throughout the transient state i.e.,

$$\left(\frac{dT^i_e(t)}{dt}\right)_{t=0^+} < -\gamma^i_c$$ \hspace{1cm} \text {for} \forall (i,c) \in G_{ne} \tag{42}$$

21
leading to,

$$\sum_{(i,c) \in G_{ne}} \left( \frac{dT_i^c(t)}{dt} \right)_{(t=0^+)} \leq \sum_{(i,c) \in G_{ne}} -\gamma_i^c$$

(43)

Using Eq. 42 and Eq. 43 in Eq. 30, the condition on $r$ can be expressed as,

$$r > \frac{\sum_{(i,c) \in S_{new} \cup G_{ne}} \gamma_i^c}{\sum_{(i,c) \in S_{new}} 1} + \left( \sum_{(i,c) \in G_{ne}} \gamma_i^c \right) \frac{1 + \sum_{(i,c) \in G_{ne}} \omega_i^c}{( \sum_{(i,c) \in G_{ne}} \omega_i^c ) \cdot \left( \sum_{(i,c) \in S_{new}} 1 \right)}$$

(44)

if $G_{ne} = \phi$, then $r$ the above condition is expressed as,

$$r > \frac{\sum_{(i,c) \in S_{new}} \gamma_i^c}{\sum_{(i,c) \in S_{new}} 1} + \left( \sum_{(i,c) \in G_{ne}} \gamma_i^c \right) \frac{1 + \sum_{(i,c) \in G_{ne}} \omega_i^c}{( \sum_{(i,c) \in G_{ne}} \omega_i^c ) \cdot \left( \sum_{(i,c) \in S_{new}} 1 \right)}$$

(45)

Following similar procedure as in Case-1, the equations for Case-2 can be easily determined. Finally, the time $t1_i^c$ at which one of the queues that belong to $S_{new}$ touches it’s threshold can be expressed as,

$$t1_i^c = \frac{\omega_i^c \cdot B}{(1 + \sum_{(i,c) \in S_{old}} \omega_i^c) \cdot (r - \gamma_i^c) + \omega_i^c \cdot \left( \sum_{(i,c) \in S_{old}} -\gamma_i^c + \sum_{(i,c) \in S_{new}} (r - \gamma_i^c) \right)}$$

(46)

Further based on $\omega$ properties and observing that $\sum_{(i,c) \in S_{old}} -\gamma_i^c = (-)Number \ of \ congested \ ports \ of \ S_{old}$ say $-NUM$.

$$t1_i^c = \frac{\alpha_i \cdot \frac{1}{N_{p(c)}} \cdot \gamma_i^c \cdot B}{(1 + \alpha_i) \cdot (r - \gamma_i^c) + \alpha_i \cdot \frac{1}{N_{p(c)}} \cdot \gamma_i^c \cdot \left( -NUM + \sum_{(i,c) \in S_{new}} (r - \gamma_i^c) \right)}$$

(47)

Notice that the presence of NUM in Eq. 47 is a dependency on the number of congested ports of Low Priority. However, NUM only creates a positive effect on $t1_i^c$ i.e., greater the NUM greater is $t1_i^c$. On the other hand, Eq. 47 is independent of negative dependencies as was in the traditional algorithm DT.

### B.5 How it all relates to Burst-Tolerance

First, if only arrival rate $r$ is known and an operator wishes to guarantee zero transient losses, from Eq. 33 assuming that load variation occur on an empty port, $\alpha_L$ is upper bounded by,

$$\alpha_L \leq \frac{1}{r-2}$$

(48)

Denote Burst-Tolerance for a queue of class $p$ at port $i$ as $Burst_i^c$ can be defined as

$$Burst_i^c = r \cdot t1_i^c$$

(49)

where $r$ is the arrival rate of traffic.

Then, the maximum burst that can pass without experiencing drops is given by $Burst_i^c$. Say an operator specifies $Burst_i^c$ i.e $r$ and $t1_i^c$ to be guaranteed to pass at all times. How can $\alpha_L$ be optimized?
Given an arrival rate $r$ and of duration $t$ on a single queue, at a given state of buffer, we are interested in providing a guarantee such that the burst is successfully absorbed. Hence we consider worst case scenarios to derive bounds on $\alpha_L$ and $\alpha_H$, where $\alpha_L$ is the maximum value for Low Priority and $\alpha_H$ is the maximum for High Priority.

Additionally for simplicity, it is assumed that a burst happens on an empty port leading to $\gamma' = 1$ and $G_e = \phi$. Considering a worst case arrival rate $r$ that only satisfies Case-2, we are interested in the buffer required and made available by buffer management without drops i.e., $(r - 1) \cdot t \leq (r - 1) \cdot t_c$. Using Eq. (47) and letting $NUM = 1$ to consider worst case,

$$
(r - 1) \cdot t \leq \frac{(r - 1) \cdot \alpha_H \cdot B}{(1 + \alpha_L) \cdot ((r - 1) + \alpha_H \cdot (-1 + (r - 1)))}
$$

(50)

For an arbitrarily large value of $\alpha_H$, there exists a limit such that,

$$
\alpha_L \leq \frac{B}{(r - 1) \cdot t} - 1
$$

(51)

Similarly, using Eq. (47),

$$
\alpha_H > \frac{1}{\frac{B}{(r - 1) \cdot t} + \gamma' + 1} - \frac{r - t}{r - 1}
$$

(52)

Finally, PLASTICINE also scales to multiple priority levels. Our analysis considers a generalized model with arbitrary number of priorities. We observe that, PLASTICINE’s allocation scheme regardless of number of queues of each priority using the buffer can always guarantee performance for the highest priority. For instance, let $\alpha_n = \alpha_1 + \alpha_2 + \alpha_3 \ldots + \alpha_n$ where $\alpha_n$ is the maximum $\alpha$ of $n$th priority, then in order to guarantee a burst of incoming rate $r$ and duration $t$ to pass at all times, $\alpha_H$ (maximum $\alpha$ across the highest priority) can still be derived by simply replacing $\alpha_L$ with $\alpha_n$ in Eq. (52).

![Figure 16](image-url)

**Figure 16:** Comparison of the burst absorption capabilities of PLASTICINE and DT showing how PLASTICINE’s performance guarantees remain unaffected by the state of buffer.
One could simply use the above analysis of transient state and generate analytical plots as shown in Fig. 16b, Fig. 16d (showing Plasticine), Fig. 16a, Fig. 16c (showing DT). The figures show, for a given $\alpha$ parameter setting, the variation of burst absorption (y axis) when the arrival rate changes. Further different lines correspond to different buffer states (with different number of pre occupied low priority queues). A buffer size ($B$) of 1MB, link capacity of 1Gbps, $\alpha_L = 0.5$ and $\alpha_H = 1024$ are used in the equations. Notice that this setting is same as in Sec. 6 enabling a comparison against analysis and simulation results.

We notice that, DT neither has an upper bound nor a lower bound. On the other hand, the strategic allocation of PLASTICINE allows for a lower bound (corresponding to a buffer state with single queue) and as the arrival rate increases the performance of various buffer states tends towards the lower bound. As a result of such bound in burst absorption, an operator could easily guarantee the absorption of a burst (corresponding to the lower bound).