RESEARCH-ARTICLE

Occamy: A Preemptive Buffer Management for On-chip Shared-memory Switches

DANFENG SHAN

YUNGUANG LI

JINCHAO MA

ZHENXING ZHANG, Huawei Technologies Co., Ltd., Shenzhen, Guangdong, China

ZEYU LIANG

XINYU WEN

View all

Open Access Support provided by:

Huawei Technologies Co., Ltd.

Published: 30 March 2025

Citation in BibTeX format

EuroSys '25: Twentieth European Conference on Computer Systems
March 30 - April 3, 2025
Rotterdam, Netherlands

Conference Sponsors:
SIGOPS

Occamy: A Preemptive Buffer Management for On-chip Shared-memory Switches

Danfeng Shan[†], Yunguang Li[†], Jinchao Ma[†], Zhenxing Zhang[‡], Zeyu Liang[†], Xinyu Wen[†], Hao Li[†], Wanchun Jiang[+], Nan Li[‡], Fengyuan Ren[◇]

[†]*Xi'an Jiaotong University* [‡]*Huawei* [+]*Central South University* [◇]*Tsinghua University*

Abstract

Today's high-speed switches employ an on-chip shared packet buffer. The buffer is becoming increasingly insufficient as it cannot scale with the growing switching capacity. Nonetheless, the buffer needs to face highly intense bursts and meet stringent performance requirements for datacenter applications. This imposes rigorous demand on the Buffer Management (BM) scheme, which dynamically allocates the buffer across queues. However, the de facto BM scheme, designed over two decades ago, is ill-suited to meet the requirements of today's network.

In this paper, we argue that shallow-buffer switches, intense bursts, along with dynamic traffic call for a *highly agile* BM that can quickly adjust the buffer allocation as traffic changes. However, the agility of the current BM is fundamentally limited by its *non-preemptive nature*. Nonetheless, we find that preemptive BM, considered unrealizable in history, is now feasible on modern switch chips. We propose Occamy[1], a preemptive BM that can quickly adjust buffer allocation. Occamy utilizes the redundant memory bandwidth to actively reclaim and reallocate the over-allocated buffer. Testbed experiments and large-scale simulations show that Occamy can improve the end-to-end performance by up to ~55%.

CCS Concepts: • **Networks** → **Data path algorithms**.

Keywords: Datacenter Networks, Buffer Management, Traffic Bursts

[1]Named after a fantastic beast in the Wizarding World that can quickly grow or shrink to fit the available space.

EuroSys '25, March 30–April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1196-1/25/03
https://doi.org/10.1145/3689031.3717495

1 Introduction

To support the ever-growing datacenter traffic, switching capacity has rapidly increased over the past decade, roughly doubling every two years [23, 39]. By 2024, leading switch ASIC vendors had increased their switching capacity to 51.2Tbps [22, 24, 62, 63]. To enable fast packet access at such high speeds, the packet buffers are embedded onto the switch chip [34, 37, 65]. As a result, the buffer size is limited by the chip area and does not scale with the increasing switching capacity. Over the last decade, the buffer size (relative to capacity) has shrunk by a factor of 4× [18, 45, 72, 93]. Consequently, buffers are becoming increasingly insufficient and this trend inclines to continue.

Despite the deficiency of buffer space, the performance requirements and traffic characteristics of datacenter networks (DCNs) place stringent demands on packet buffers. Modern networked applications require ultra-low latency (*e.g.,* <10μs for resource disaggregation [20, 42]) and high throughput (*e.g.,* >100Gbps for NLP model training [77]) from the network. Additionally, workflows such as partition-aggregation and MapReduce can aggregate thousands of flows at a single point [58], resulting in highly intense traffic bursts. Moreover, most bursts occur at a sub-RTT level [44, 94], and are beyond the control of end-to-end mechanisms. To avoid packet loss, enough buffer space is required to absorb these bursts. Hence, the buffer on the switch is a key factor in performance.

Modern datacenter switches have multiple ports, and each port contains several queues, each of which is dedicated to a traffic class [21, 86]. The on-chip buffer is typically shared across all queues to improve buffer utilization [12, 34, 37, 64, 85]. The Buffer Management (BM) scheme, which dynamically allocates the shared buffer to queues, is expected to be efficient for burst absorption [44] and high throughput [17, 18], as well as fair for performance isolation [1]. However, the de facto BM (*i.e.,* Dynamic Threshold or DT [27, 28]), designed over two decades ago, is far from satisfactory in terms of both burst absorption [10, 44, 71, 94] and performance isolation [1, 9].

In this paper, we argue that the underlying cause of these problems is that the *non-preemptive nature* of the current BM confines its agility to adjust buffer allocations when facing dynamic traffic. By non-preemption, we mean that a previously accepted packet is not allowed to be expelled.

As a result, the BM can only adjust the buffer allocation by passively waiting for the over-allocated queue to naturally release the buffer (*i.e.,* by sending out the packets). This incurs two issues.

(1) Inefficient: Due to the limited agility in adjusting buffer allocations, non-preemptive BMs have to proactively reserve a portion of free buffer space to protect newly active queues from starving for buffer. As a result, non-preemptive BMs fail to efficiently utilize the scarce buffer, which is undesirable given that datacenter switch buffers are already insufficient and need to be fully utilized to absorb bursts when facing bursty short flows [44, 76] and achieve high throughput when facing long flows [17, 18, 93].

(2) Unfair: In DCN, the traffic arrival rate can be much higher than the departure rate as a result of highly bursty traffic. In such cases, non-preemptive BMs fail to adjust the buffer allocation in time, resulting in *anomalous behaviors* where packets are reluctantly dropped before obtaining the deserved buffer. In particular, we find that non-preemptive BM can suffer from the *buffer choking problem* (§3.1), in which high-priority (and usually delay-sensitive) traffic thirsts for buffer, whereas low-priority traffic holds considerable over-allocated buffer while draining slowly.

If BM could support preemption (*i.e.,* actively drop packets residing in the buffer), to some extent, the above issues can be easily addressed. Preemptive BMs had been studied a lot in history, and a kind of BM called Pushout had been proven optimal in terms of throughput and loss probability [25, 29–31, 57, 79, 81, 82, 89]. Pushout accepts the arriving packets whenever there is free buffer space, and when the buffer becomes full, Pushout expels packets from the longest queue to make room for the incoming packets. However, implementing Pushout was historically considered challenging within the traditional architecture of switching fabric [27, 29] (details in §2.2). Nonetheless, over the past two decades, the buffer architecture has evolved a lot. Notably, the packet buffer has been embedded into the switch chip, significantly increasing the memory bandwidth. Thus, it is necessary to ask: *is it now feasible to support preemptive operations in today's on-chip shared-memory switch?*

In this paper, we answer the above question with Occamy, a simple preemptive BM that can quickly adjust buffer allocation by actively expelling packets for the over-allocated queues (§4). Occamy has two key differences from Pushout, which makes it simple to be implemented. (1) Unlike Pushout whose enqueue operations may need to wait for the completion of expelling packets, Occamy decouples packet expulsion from packet enqueue, keeping enqueue operations simple. (2) Unlike Pushout that drops packets from the longest queue, Occamy expels packets from all over-allocated queues in a round-robin manner, avoiding the prohibitive costs of tracking the longest queue.

Combining the above two ideas, we design Occamy with two components: a proactive component and a reactive component. The proactive component can avoid buffer starvation while achieving high buffer efficiency. As the enqueue operations do not wait for packet expulsion, Occamy proactively reserves a *small* fraction of free buffer to protect newly active queues from starving for buffer. The reactive component endows Occamy with agility. When traffic changes, Occamy reactively adjusts the buffer allocation by actively expelling packets for the over-allocated queues. The proactive component can be realized by directly utilizing the off-the-shelf BM in commodity switch chips with adjusted parameters, while the reactive component can be achieved by several simple circuit components.

We implement the core components of Occamy with 286 lines of Verilog code, and evaluate its hardware cost with both the Vivado Design Suite [8] and an open-source 45nm ASIC technology library [41]. The results show that Occamy requires only ∼1300 LUTs, ∼50 Flip-Flops, less than $0.03mm^2$ ASIC area, and consumes just $1mW$ power. Additionally, we implement a proof-of-concept hardware prototype using P4 (§5.2) and a software prototype based on DPDK (§5.3). Experiments on an Intel Tofino switch demonstrate that Occamy improves burst absorption (*i.e.,* maximum burst size without packet drops) by up to ∼57% (§6.1). Experiments using a DPDK-based software switch show that Occamy improves the average Query Completion Time (QCT) by up to ∼55%, while effectively ensuring performance isolation between different traffic classes and avoiding buffer choking (§6.2). Furthermore, large-scale simulations show that Occamy reduces the average QCT by up to ∼44% with web-search workload, ∼33% with all-to-all traffic, and ∼48% with all-reduce traffic (§6.4).

We believe Occamy showcases a step towards preemptive BM on modern switch chips. Since two decades ago, the BM studies have been concentrating on innovating non-preemptive BMs. In contrast, our exploration demonstrates the potential and feasibility of utilizing preemption to approach the ideal performance. While actually implementing preemptive BM in high-speed commodity switch ASICs may still require lots of effort, we hope our exploration can inspire the research and industry community to embrace the preemptive approaches.

In summary, our key contributions are:

- We show that, with insufficient buffer and dynamic traffic, today's DCN requires a highly agile BM, while the agility of the current BM is fundamentally limited by its non-preemptive nature.
- We design Occamy, showing that preemptive BMs are feasible to be realized with several simple components.
- We use extensive experiments to show that Occamy outperforms non-preemptive BMs in terms of burst absorption and flow completion time.

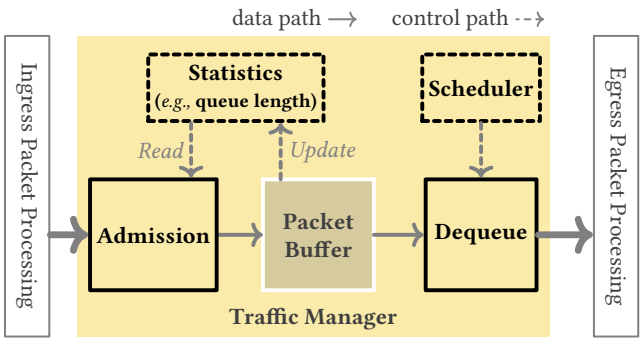


Figure 1. Structure of on-chip shared-memory switch

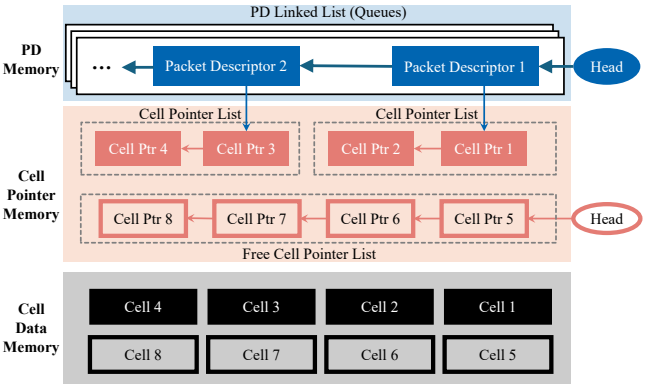


Figure 2. Structure of packet buffer

The code of Occamy is available at https://github.com/ants-xjtu/Occamy.

2 Background

In this section, we provide background on buffer architecture and buffer management schemes in modern switches.

2.1 Architecture of Shared-memory Switches

Switch structure. Figure 1 depicts the high-level structure of modern shared-memory switch chips [4, 12, 32, 64, 90]. A typical switch chip mainly consists of three parts: Ingress Packet Processing, Traffic Manager (TM), and Egress Packet Processing. In this paper, we mainly focus on TM, which accommodates the packets and dynamically allocates the buffer across queues.

For each arrival packet, TM first performs admission control (*e.g.,* BM and active queue management) to decide whether the packet can be accepted into the buffer. Once admitted, the packet is written into a packet buffer, which is a centralized and globally shared on-chip SRAM. At the egress side, a scheduler selects a queue to fetch packets using a scheduling algorithm (*e.g.,* deficit round robin).

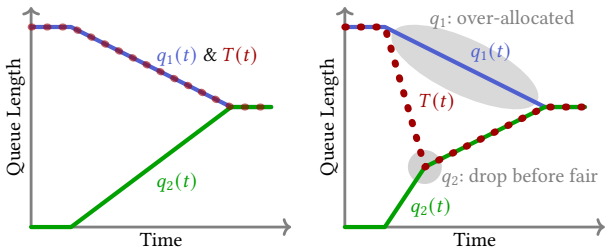
Queue/buffer structure. Figure 2 dives into the details of how packets are stored and queues are organized in the packet buffer. Typically, there are three pieces of memory: cell data memory, cell pointer memory, and Packet Descriptor (PD) memory. At the bottom, the cell data memory accommodates the actual packet data, which is divided into equal-sized cells. In the middle, the cell pointer memory holds the cell pointers to these cells. For a packet partitioned into multiple cells, its cell pointers are linked together through a *cell pointer list.* Moreover, the cell pointer memory also manages the free cells through a linked list, referred to as *free cell pointer list.* The free cell pointer list contains the addresses of all free cells. Upon the arrival of a packet, TM allocates memory for the packet by removing several cell pointers from the free cell pointer list. Upon the departure of a packet, TM frees the occupied memory by returning all cell pointers of the packet to the free cell pointer list. Finally, at the top, the PD memory contains packet descriptors (PDs). Each packet is associated with a PD, containing the packet metadata (*e.g.,* packet length) and the head(s) of cell pointer list(s). A queue is organized as a linked list of PDs.

Modern high-speed switches employ several techniques to speed up packet readings and writings. For example, the three types of memory are usually in different physical SRAMs, allowing parallel accesses to PDs, cell pointers, and cells. Furthermore, to speed up packet readings, high-speed switches may divide the cell pointer list into multiple sublists (*e.g.,* dividing Cell Ptr 1 → Cell Ptr 2 into Cell Ptr 1 and Cell Ptr 2). In this way, multiple cell pointers can be read simultaneously at the cost of maintaining multiple link headers (*e.g.,* Cell Ptr 1 and Cell Ptr 2).

2.2 Buffer Management Schemes

TM uses a Buffer Management (BM) scheme to dynamically allocate the shared-buffer across queues. Depending on whether an accepted packet can be dropped, BM schemes can be divided into two categories: non-preemptive BMs and preemptive BMs.

Non-preemptive BMs. Non-preemptive BMs are dominant in today's switch chip due to their simplicity of implementation. With non-preemptive BMs, a packet won't be expelled or overwritten once accepted into the buffer. As a result, they have to *proactively* reserve a fraction of free buffer to accommodate transient bursts arriving at newly active queues. Furthermore, when the buffer allocation (*i.e.,* the amount of buffer allocated to each queue) needs to be adjusted, non-preemptive BMs can only *passively* wait for the over-allocated queue to naturally release the buffer through packet transmissions. However, the queue drain rate is limited by the port speed and can be even lower when bandwidth is shared with other queues. Hence, non-preemptive BMs are not agile enough to adjust the amount of buffer allocated to each queue under the condition of high traffic arrival rate or low departure rate, leading to anomalous behaviors. Nevertheless, non-preemptive BMs are straightforward to implement by limiting the queue length with a threshold on the admission module.



(a) Healthy behavior **(b)** Anomalous behavior

Figure 3. Dynamic behavior of DT

Next, Dynamic Threshold (DT) is used to exemplify the healthy and anomalous behaviors of non-preemptive BM schemes. DT is very prevalent in commodity switch chips [11, 12, 19, 21, 33, 35, 37, 46, 48, 59, 87]. It limits queue lengths with a threshold (denoted by $T(t)$), which is dynamically adjusted based on the buffer occupancy. Specifically, the threshold is proportional to the current free buffer size, namely,

$$T(t) = \alpha \left(B - \sum_{i=1}^{N} q_i(t) \right) \tag{1}$$

where α is a control parameter, B is the shared buffer size, and $q_i(t)$ is the length of queue i at time t. In practice, α is usually a power of two, so that the threshold can be simply calculated by shifting the free buffer size. The intuition of Eq. (1) is that, when the switch is less congested, DT allocates more buffer to each queue for high efficiency; when the switch is more congested, DT limits queue lengths more strictly for fairness.

To illustrate the behavior of DT, consider a scenario with two queues. Only queue 1 is congested and has reached the steady state at the beginning (*i.e.,* its queue length has reached $T(t)$). Then some traffic bursts arrive at queue 2, needing to occupy some buffer. Figure 3(a) shows the evolutions of queue length and threshold in the healthy case. As the length of queue 2 (*i.e.,* $q_2(t)$) grows, DT gradually reduces the threshold. This, in turn, causes the length of queue 1 (*i.e.,* $q_1(t)$) to decrease by draining the excess buffer occupancy. Finally, both queue 1 and queue 2 will occupy the same amount of buffer, achieving fair buffer sharing.

Figure 3(b) illustrates the anomalous behavior, where queue 1 cannot decrease its length as fast as $T(t)$. This case occurs either when $q_2(t)$ increases too quickly due to high traffic arrival rate, or when $q_1(t)$ reduces too slowly due to the shared capacity of the egress port with other queues. Either way, DT is unable to adjust the buffer allocation in time. As a result, queue 1 occupies excess buffer, forcing queue 2 to reluctantly drop packets before receiving its deserved buffer space.

Preemptive BMs. Preemptive BMs can expel or overwrite the accepted packets. Thus, preemptive BMs usually do not need to proactively reserve buffer for inactive queues as they can quickly make room by packet expulsion. Instead, they *reactively* adjust the buffer allocation when facing dynamic traffic. Consequently, preemptive BMs can achieve higher buffer efficiency.

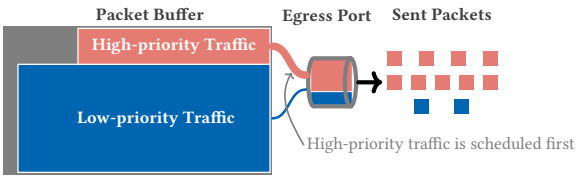


Figure 4. The buffer choking problem

In particular, a kind of preemptive BM, known as Pushout, had been regarded as optimal [25, 29–31, 57, 79, 81, 82, 89]. Pushout allows a packet to enter into the buffer whenever there is some free space. When the buffer is out of space, Pushout expels a packet from the longest queue to make room for the new arrival packet. However, implementing Pushout in switch chips was historically considered difficult due to the following three reasons [13, 27, 29] (more detailed analysis in [73]):

Difficulty 1: *Requirement of higher memory bandwidth.* When the buffer is full, accepting a packet requires reading another packet out of the memory (note that queues were not in a separate memory [51]), which imposes a higher requirement on memory latency and bandwidth. However, memory access was considered as the critical path at that moment, as the memory speed could not keep pace with the switching capacity [51–53]. The inadequate memory bandwidth is primarily used to achieve high switch capacity, leaving no redundancy for BM.

Difficulty 2: *Complex enqueue operations.* When a packet arrives, the enqueue operation may need to wait for the completion of expelling packets from another queue, which requires extra buffer at the ingress side and coordination between ingress and egress.

Difficulty 3: *Monitoring the longest queue in real time.* The longest queue can be any queue and may change whenever a packet arrives and leaves. Monitoring the longest queue across numerous queues in real time is time-consuming [27] (more details in [73]), and thus is unacceptable in high-speed switches.

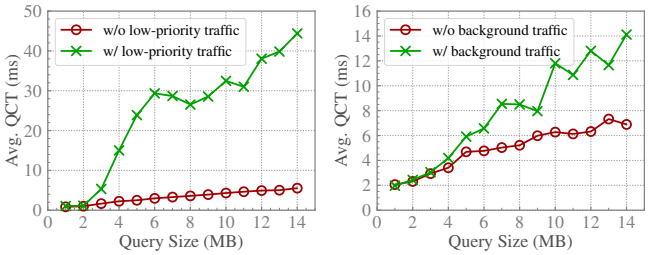
3 Motivation

In this section, we demonstrate the issues of non-preemptive BMs and analyze the underlying causes. Then, we discuss the opportunities opened up by modern switch chips, which motivate us to innovate preemptive BMs.

3.1 Issues of Non-preemptive BMs

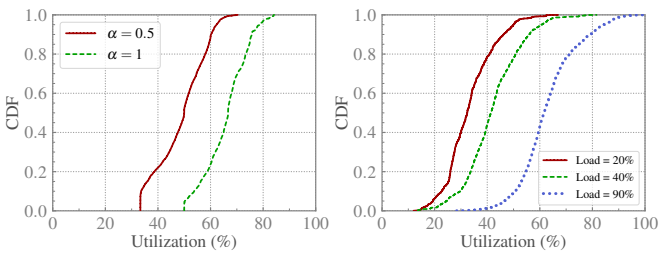
Despite their dominance in commodity switch chips, non-preemptive BMs have some intrinsic issues.

Non-preemptive BMs can work *anomalously*. This is due to two reasons. (1) The queue drain rate can be very low, causing non-preemptive BMs to suffer from the *buffer choking problem*, as shown in Figure 4. Specifically, DCN operators



(a) Low-priority influence **(b)** Inter-port influence

Figure 5. Performance degradation due to anomalous behavior



(a) Buffer **(b)** Memory bandwidth

Figure 6. CDF of buffer/memory bandwidth utilization

can leverage different queues in a port to isolate the performance of different services [16]. Latency-sensitive traffic is usually placed in the high-priority queue with strict priority queueing, or assigned with higher weight with weighted fair queueing. Consider a case when the low-priority queues are congested earlier, and later some bursty traffic arrives at the high-priority queue. In this case, the low-priority queues hold considerable buffer. However, since most of the bandwidth is allocated to the high-priority queue, the low-priority queues drain slowly. As a result, the high-priority queue runs out of buffer space soon, while buffer management (BM) fails to free up enough space for it. Consequently, high-priority packets are unexpectedly dropped due to insufficient buffer allocation.

To quantitatively demonstrate this issue, we build a testbed comprising 4 hosts connected to a Huawei CE6865 switch. Each host is equipped with an Intel XL710 Dual Port 40GbE NIC. Using network namespaces, we isolate two ports on each NIC to emulate two separate NICs and end nodes per host (8 nodes in total). The switch is configured to dynamically allocate 2MB buffer across eight 40Gbps ports via DT. The switch supports 8 class-of-service queues, with one designated as high priority, and the rest as low priority. We employ DCTCP as the congestion control algorithm, with the ECN threshold set to 300KB as recommended in [17]. We generate two types of traffic, which are from different senders to the same receiver: ① High-priority incast traffic: The receiver simultaneously sends queries to 5 senders, each responding with data upon receiving the query. The incast degree is 40 (*i.e.,* 8 flows per sender). ② Low-priority background traffic: We generate 14 long-lived flows from 2 other senders to the same receiver, each of which is classified into one of 7 low-priority queues. To examine the performance of incast traffic, we measure the query completion time (QCT), which is the completion time of all incast flows. We compare QCT between two scenarios: one with low-priority traffic and the other without. For a fair comparison, we configure DT such that the incast traffic should be allocated the same amount of buffer in both scenarios. Specifically, we set $\alpha = 8$ for the high-priority queue with low-priority traffic and $\alpha = 1$ without low-priority traffic. For low-priority queues, we set $\alpha = 1$. In this way, the incast traffic deserves 1MB buffer either with or without low-priority traffic [1, 27], and ideally, the QCT performance should be unaffected by low-priority traffic.

Figure 5(a) shows the average QCT with different query sizes (*i.e.,* the total volume of incast traffic), with and without low-priority traffic. Although the QCT is expected to remain unaffected, the presence of low-priority traffic significantly degrades the average QCT by up to ~8× compared to the scenario without low-priority traffic. This is because DT cannot quickly drain the low-priority traffic from the buffer due to buffer choking. As a result, the high-priority incast traffic receives only a small portion of the buffer and experiences packet drops before getting its deserved buffer.

(2) The traffic arrival rate can be very high. The traffic patterns such as incast are very prevalent in datacenters [61]. As a result, thousands of flows can aggregate at the switch [58], introducing intense traffic bursts. Without the ability to quickly make room for the highly bursty traffic, packets can be dropped before enough buffer is allocated to them.

To quantitatively illustrate this issue, we use the same experimental settings as before, except that two types of traffic are congested at different ports, thereby eliminating the impact of buffer choking. Figure 5(b) shows that the average QCT with background traffic still degrades by up to ~2× compared to the scenario without background traffic, despite both receiving the same buffer allocation in theory. This is because DT is not quick enough to adjust the buffer occupancy when facing fast incast traffic, leading to packet drops before the buffer is properly reallocated.

Non-preemptive BMs are inefficient. As non-preemptive BMs cannot immediately make room for newly active queues, they have to proactively reserve some buffer, which is essential to absorb transient bursts. As a result, they cannot fully utilize the buffer. To show this issue, we conduct a simulation with a leaf-spine topology of 8 spine switches, 8 leaf switches, and 128 servers connected via 100Gbps links. Flows are generated according to the web-search workload [5]. The network load is 40% (detailed settings in §6.4). Figure 6(a) shows the buffer utilization on packet drop with DT. With typical settings (*i.e.,* $\alpha = 0.5$) in production datacenters [19], the 99th percentile buffer utilization is only ~66%. Given the situation that on-chip buffer is very scarce and adequate buffer occupancy is needed for both low latency [44, 74, 94]

and high throughput [17, 18, 93], it is better not to waste the scarce buffer.

Analysis of root causes. The radical reason for the above issues is that non-preemptive BMs are *not agile enough* to adjust buffer allocations. The underlying factor that limits their agility is the *non-preemptive nature*. Specifically, without the ability to actively expel packets, non-preemptive BMs have to passively wait for the over-allocated queues to naturally release buffer through normal packet transmissions. Moreover, due to the limited speed of buffer adjustment, non-preemptive BMs have to proactively reserve some free buffer to prevent newly active queues from buffer starvation, resulting in buffer inefficiency. Thus, an effective approach to address the above issues is preemptive BM that can actively expel packets for the over-allocated queues.

3.2 Opportunities

As shown in §2.2, preemptive BM schemes were historically considered hard to implement due to three difficulties. In this part, we argue that the limitation of bandwidth (*i.e.,* Difficulty 1) is less critical with the current buffer architecture, which opens up opportunities to innovate preemptive BMs by overcoming Difficulty 2 and Difficulty 3.

(1) *It is easier to extend the memory bandwidth in modern switches.* Different from the traditional switches using off-chip buffer, modern switches have embedded the packet buffer in the switch chip. Different off-chip buffer, on-chip buffer has a significantly wider memory data path, since no external pins (*i.e.,* pins connected to external memory) are required [36, 37, 70], thereby reducing the requirements of IC packaging (*i.e.,* without the need for inter-chip connection). As a result, the memory bandwidth can be significantly increased.

(2) *Preemptive BM does not consume the bandwidth of cell data memory.* As explained in §2.1, with the current buffer structure, dropping a packet only needs to dequeue the PD and move the packet's cell pointers to the free cell pointer list. These operations can be finished within PD memory and cell pointer memory. There is no need to access the cell data memory, thereby reducing the memory bandwidth requirement.

(3) *The throughput of reading cell pointers can be improved by batching.* A potential bottleneck of dequeuing a packet lies in the cell pointer memory. This is because cell pointers are very small, and the throughput of reading cell pointers is limited by the clock rate (rather than the memory bandwidth), which is unfortunately hard to be increased due to the limits of thermal dissipation. Nevertheless, as described in §2.1, the read throughput can be increased by reading multiple cell pointers at a time. For example, a PD can maintain 4 parallel cell pointer lists, allowing 4 cell pointers to be read simultaneously.

(4) *In practice, the switch does not always use up all memory bandwidth.* For a commodity switch chip, the TM is designed to be able to achieve full bisection bandwidth. However, in practice, it is unlikely that all ports are sending and receiving packets at line rate, as the traffic load rarely reaches 100% for all ports. To concretely show this, we conduct the same large-scale simulation as in §3.1. Figure 6(b) shows the CDF of memory bandwidth utilization on packet drop with different network loads, where utilization = $\frac{\text{consumed memory bandwidth}}{\text{overall memory bandwidth}}$. We can see that, even under 90% network load, the median free memory bandwidth is ~38%.

4 Occamy Design

In this section, we first introduce Occamy's high-level ideas and overall structure. Then, we present the design details.

4.1 Overview

To maintain simplicity while leveraging redundant memory bandwidth for packet expulsion, Occamy embodies three main ideas:

(1) *Occamy decouples admission and expulsion.* To overcome Difficulty 2, Occamy makes the admission and expulsion mutually independent. In this way, the admission module can enqueue the admitted packets immediately, keeping the enqueue operations simple. However, this can result in buffer starvation. As the enqueue operations do not wait, any arriving packet is dropped once the buffer is full. Thus, we use the following idea to avoid this issue.

(2) *Occamy proactively reserves a small fraction of free buffer.* Since the enqueue operations do not wait, Occamy proactively reserves a portion of free buffer to protect newly active flows from buffer starvation. Note that only a small fraction is needed because Occamy can agilely vacate buffer for newly active flows.

(3) *Occamy reactively expels packets for all over-allocated queues in a round-robin manner.* To overcome Difficulty 3, Occamy monitors all over-allocated queues, which is simple as it can be realized by comparing the queue lengths to a threshold. Besides, rather than only pushing out the longest queue, Occamy expels packets for all over-allocated queues in a round-robin manner.

Overall, Figure 7 depicts the high-level structure of Occamy. Different from a non-preemptive BM, Occamy adjusts the buffer allocation with both **packet admission** and **packet expulsion**.

Packet admission (§4.2). Packet admission lies at the ingress side of TM. For each arriving packet, Occamy determines whether to admit it into the packet buffer based on queue length statistics.

Packet expulsion (§4.3). Packet expulsion lies at the egress side of TM. It contains four modules: head-drop selector, arbiter, demultiplexer, and head-drop executor. First, a head-drop selector picks an over-allocated queue for expulsion

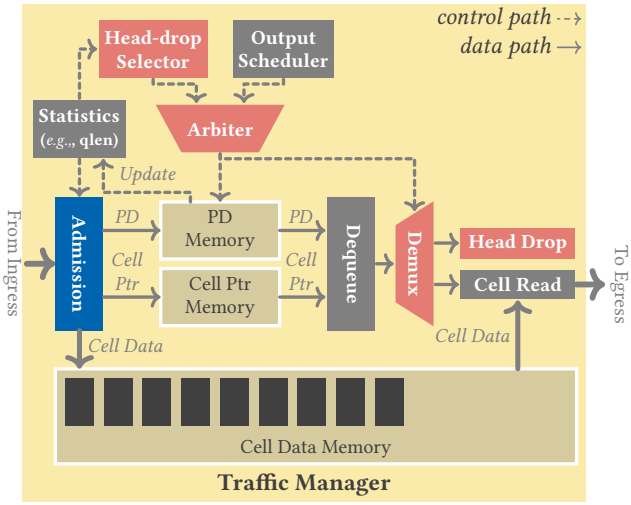


Figure 7. Occamy overview

based on queue length statistics. Next, an arbiter resolves the read conflicts between the head-drop selector and the output scheduler. For each fetched PD, Occamy determines whether to drop or dequeue the packet for transmission through a demultiplexer, with the head-drop executor handling the packet drops.

4.2 Packet Admission

Similar to non-preemptive BM schemes, Occamy proactively reserves a fraction of free buffer to avoid buffer starvation for newly active queues. Unlike non-preemptive BM schemes, Occamy only needs to reserve a *small* fraction of free buffer, as it can utilize head drop to quickly expel the over-allocated buffer.

To realize this, Occamy leverages DT with adjusted parameter for both high efficiency and ease of implementation. As detailed in §2.2, DT is very simple and has already been widely deployed in the commodity switch chip. Utilizing the existing mechanisms can facilitate the implementation of Occamy. Moreover, the efficiency of DT can be tuned through its parameter, α [27]. Concretely, a larger α enables DT to make more efficient use of scarce memory resources. For example, with $\alpha = 8$, DT allows a queue to occupy 88.9% of the buffer. Nevertheless, α should not be arbitrarily large, as Occamy requires some buffer reservation to accommodate the newly arriving bursts (more details in §4.4).

Combining the above two ideas, Occamy avoids introducing new mechanisms for the packet admission module. Instead, we only need to adjust the parameter of DT in the existing switch chip.

4.3 Packet Expulsion

The core idea of this module is to leverage redundant memory bandwidth to actively expel the over-allocated buffer. Next, we will present its design details.

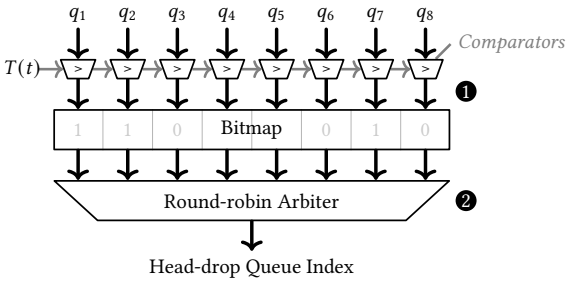


Figure 8. Head-drop selector

Selecting a head-drop queue. As analyzed in §3.1, a BM can work anomalously when the queue length is higher than the threshold (*i.e.,* $T(t)$). Thus, Occamy considers a queue over-allocated if and only if its queue length is higher than $T(t)$. Moreover, rather than maintaining the longest queue that can incur non-trivial costs (more details in [73]), Occamy maintains all over-allocated queues and selects a queue in a round-robin manner. The above functions are achieved in the head-drop selector module, whose circuit diagram is shown in Figure 8. The head-drop selector module contains two parts: maintaining the over-allocated queues (❶) and iterating over them (❷).

❶ To maintain the over-allocated queues, Occamy uses a bitmap, whose length is equal to the number of queues, to indicate whether a queue is over-allocated. Each bit of the bitmap represents whether the queue length exceeds $T(t)$. For example, the bitmap `00000101` denotes that queue 1 and queue 3 are over-allocated. Each bit is set by comparing the queue length to $T(t)$, using some simple combinatorial logic that consists of some cheap comparators.

❷ With multiple over-allocated queues, Occamy iterates over them with a round-robin arbiter, which is a common hardware component in high-speed crossbar switches [38, 47, 75]. The round-robin arbiter takes the bitmap as input and gives the index of an over-allocated queue.

Resolving the conflicts between the output scheduler and head-drop selector. The head-drop operation also needs to consume the read bandwidth of PD memory and cell pointer memory. Thus, the read conflict arises when both output scheduler and head-drop selector fetch packets simultaneously. In such cases, the head-drop selector should give way to the output scheduler. Otherwise, Occamy may not guarantee line-rate forwarding, which is unacceptable. Thus, Occamy uses a fixed-priority arbiter to resolve the conflict. The read requests from the head-drop selector are blocked whenever the output scheduler needs to fetch a packet.

Performing the head-drop action. If a head-drop request is granted, Occamy extracts a packet at the head of the queue and drops it. As explained in §3.2, the head-drop operation does not need to access the cell data memory. Instead, Occamy simply dequeues the corresponding PDs from the PD linked list, and returns the cell pointers of the packet to the free cell pointer list. Moreover, as the head-drop action shares

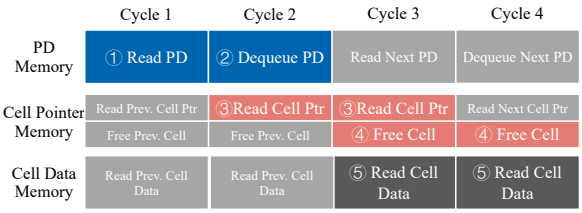
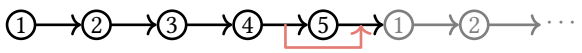


Figure 9. Packet dequeue pipeline

lots of operations with the normal dequeue action, it can be synthesized into the existing packet dequeue pipelines.

Specifically, as shown in Figure 9, a dequeue action mainly consists of 5 operations: ① Read a PD from PD memory; ② Dequeue the PD (*i.e.,* advance the head of the PD linked lists); ③ Read cell pointer; ④ Free cell (*i.e.,* move the cell pointer to the free cell pointer list); ⑤ Read cell data. With a packet partitioned into multiple cells, operation ③, ④, and ⑤ can be executed multiple times. Moreover, as PD memory, cell pointer memory, and cell data memory are physically separate, accesses to them can be parallelized. In this pipeline, the only difference between the head-drop action and the normal dequeue action is that the head-drop action does not need to read the cell data (*i.e.,* operation ⑤). Thus, with Occamy, these operations can be recomposed to



where ⟶ denotes the (extra) execution path introduced by Occamy.

4.4 Parameter Settings

Occamy includes a parameter α, which influences the buffer efficiency and fairness of Occamy. In this part, we analyze and discuss the settings of α.

Since Occamy directly uses DT for admission control, we can follow the method and conclusion in [27] to analyze Occamy's efficiency and fairness. According to [27], the amount of reserved free buffer in the steady state can be given by

$$F = \frac{B}{1+\alpha N} \tag{2}$$

where B is the buffer size, N is the number of congested queues whose length is equal to the threshold. Eq. (2) denotes that Occamy reserves less free buffer with a larger α, resulting in higher efficiency. Nevertheless, the efficiency improvement is undermined as the α becomes larger. For example, when $N = 1$, the free buffer reservation is $B/9$ with $\alpha = 8$ and $B/17$ with $\alpha = 16$, resulting in only a 5.2% increase in buffer utilization (note that α is typically a factor of two to facilitate the threshold calculation). Thus, it is not necessary to set a very high value for α.

On the other hand, according to [27], when bursty traffic arrives at inactive queues, Occamy can fairly allocate buffer only if

$$R \leqslant V\left(1+\frac{1+\alpha N}{\alpha M}\right) \tag{3}$$

where R is the arrival rate of traffic bursts, V is the expulsion rate, M is the number of queues receiving traffic bursts, and N is the number of over-allocated queues. Inequality (3) can be rewritten as

$$\frac{1}{\alpha} \geqslant \left(\frac{R}{V}-1\right)M - N \tag{4}$$

Inequality (4) implies that α can be set to a larger value with a higher packet expulsion rate. For example, with one over-allocated queue (*i.e.,* $N = 1$) and one burst arrival (*i.e.,* $M = 1$), Inequality (4) can be rewritten as $\frac{1}{\alpha} \geqslant \frac{R}{V} - 2$. When $V \geqslant R/2$, α can be arbitrarily high in theory.

In sum, setting α involves a tradeoff between efficiency and fairness, and this tradeoff is less tense with a higher expulsion rate. Our experiments (in §6.3) show that $\alpha = 8$ is an appropriate choice. Increasing α further does not result in notable improvement in efficiency, but can result in unfairness.

4.5 Discussions

Impact of preemption. One might wonder whether the preemption (*i.e.,* expelling packets by head drop) introduced by Occamy could impact normal packet processing (*e.g.,* scheduling) at switches. Our design ensures that the preemption does not cause any impacts for the following two reasons. ① With the fixed-priority arbiter, the preemption only occurs when the output scheduler is not fetching packets from the queue, avoiding interference with the ongoing packet processing. ② If the output scheduler needs to fetch packets from the queue while Occamy is performing preemption, the preemption process can be interrupted at any time. Specifically, as shown in Figure 9, interruption occurs at the beginning of either Cycle 1/2 or Cycle 3/4. In the former case (at the beginning of Cycle 1 or 2), Occamy has not yet modified to the queue (*i.e.,* PD linked list), allowing the output scheduler to dequeue packets as if Occamy were not present. In the latter case (at the beginning of Cycle 3 or 4), the PD to be expelled has already been removed from the queue by the end of Cycle 2, so the output scheduler will believe that the corresponding packet has been dequeued, thereby starting to dequeue a new packet.

What if there is no redundant bandwidth? Without redundant memory bandwidth (*i.e.,* the memory bandwidth is saturated), Occamy operates similarly to Dynamic Threshold (DT), which is sufficient because there is no need to agilely adjust buffer allocations. Specifically, memory bandwidth saturation occurs only when all ports are transmitting and receiving packets at full rate. In this case, the overall ingress traffic rate is nearly equal to the egress traffic rate. From the buffer's perspective, the traffic arrival rate matches the departure rate, leading to relatively stable buffer occupancy in each queue, thereby reducing the need for agile buffer management.

Table 1. Hardware Cost

Module	FPGA Cost		ASIC Cost		
	LUTs	Flip Flops	Timing (ns)	Area (mm^2)	Power (mW)
Selector	1262	47	1.49	0.023	0.895
Arbiter	3	0	0.17	2.3e-5	0.003
Executor	47	7	0.38	7.3e-4	0.044

5 Implementation

In this section, we first analyze the hardware costs of Occamy's components. Then we describe the implementations of a proof-of-concept hardware prototype based on P4 and a full-featured software prototype based on DPDK.

5.1 Hardware Cost

To evaluate the hardware cost of Occamy, we implement the head-drop selector (with a 64-bit bitmap), arbiter, and head-drop executor with 215, 11, and 60 lines of Verilog code, respectively. To analyze the overhead of each hardware component, we use Vivado Design Suite [8] to evaluate the FPGA resource consumption. Furthermore, we use Design Compiler [78] to synthesize each component on an open-source 45nm ASIC technology library [41] to evaluate the timing, chip area, and power consumption.

Table 1 shows the FPGA resource consumption of each component reported by Vivado and the ASIC cost reported by Design Compiler. Note that the numbers are per switch chip, as all queues share the same head-drop selector, arbiter, and executor. The majority of hardware cost comes from the head-drop selector. Specifically, the head-drop selector requires ~1200 LUTs and dozens of Flip Flops, incurring little resource consumption on FPGA (note that an FPGA chip typically contains hundreds of thousands of LUTs and Flip Flops [7]). Besides, the timing reports show that the head-drop selector has a delay of less than 1.5ns, indicating that Occamy can expel a packet every 2 cycles with a 1GHz clock, which is fast enough, as expelling a packet usually requires several cycles to dequeue multiple cell pointers. Furthermore, Occamy introduces less than $0.03mm^2$ ASIC area and $1mW$ power, which are negligible for a commodity switch ASIC.

5.2 P4-based Hardware Prototype

We implement Occamy with 616 lines of code in P4, which can run on our Intel Tofino switch. As the TM cannot be programmed, we cannot implement the head-drop selector and fixed-priority arbiter. Consequently, we only implement the packet admission module and head-drop operations to examine the benefits brought by actively expelling packets for over-allocated queues.

Admission. The admission control is implemented in the ingress pipeline. The admission module requires queue lengths and threshold. In our switch, queue lengths can be obtained only at the egress pipeline. We follow the method in [9, 92] to make them available at the ingress pipeline. Specifically, we generate special packets called SYNC packets. These packets read the queue lengths from the egress pipeline and return to the ingress pipeline via recirculation, where the queue lengths are copied to the ingress register array. The threshold is also calculated as SYNC packets traverse through the egress pipeline and is synchronized to the ingress pipeline in the same manner. Whenever a packet arrives at the ingress pipeline, we compare the length of the corresponding queue to the threshold given by DT, deciding whether to accept the packet.

Head drop. The head drop is implemented in the egress pipeline. Whenever a packet arrives, we compare the threshold to the length of the queue it comes from. The head drop operation is conducted by setting the metadata `drop_ctl`. Specifically, we set `drop_ctl=1` if the queue length is higher than $T(t)$, which tells the switch to drop the packet.

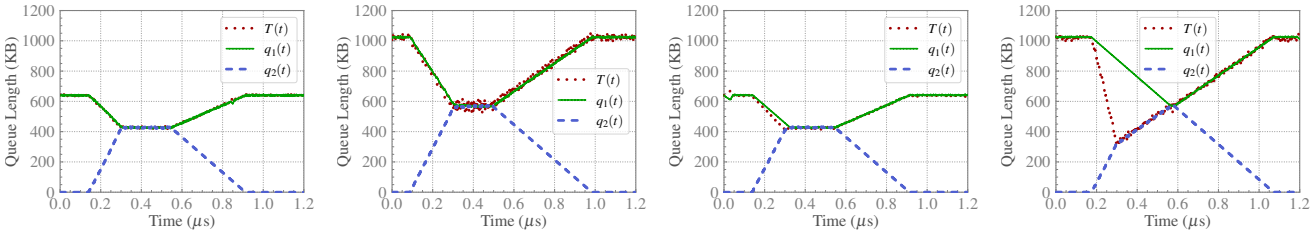
5.3 DPDK-based Software Prototype

We also implement a software prototype of Occamy with DPDK. We use a server with eight 10Gbps NICs to emulate the switch and implement Occamy on top of it. The prototype consists of four modules: an RX module, a forwarding module, a TX module, and an expulsion module. Each module runs on separate CPU cores. Occamy is implemented on the forwarding module and expulsion module.

RX, forwarding, and TX module. The RX module polls each NIC port for packets and delivers them to the forwarding module through an RX queue. The forwarding module fetches packets from the RX queue, and determines the destination queue of each packet. Then the forwarding module executes Occamy's admission control component, *i.e.,* it decides whether to admit the packet based on DT. Passing the BM, the packet is put into the corresponding output queue. The TX module fetches packets from the output queues and delivers them to the NIC. Furthermore, to accelerate the TX module, we assign a dedicated CPU core to send packets for each NIC port.

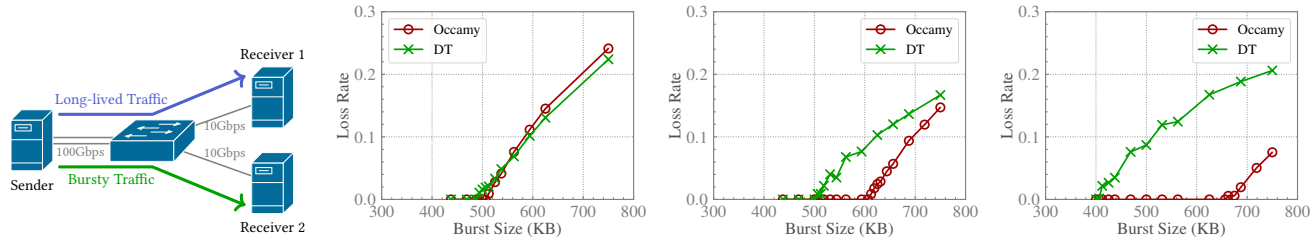
Expulsion module. The expulsion module has two functions. ① It ensures that packet expulsion is performed only with redundant memory bandwidth. To achieve this goal, we maintain a token bucket, and generate tokens based on the switching capacity. Specifically, our software switch has eight 10Gbps ports and thus can forward packets at a rate of 80Gbps. We assume that each packet is partitioned into 200B cells[2], which means that the switch can dequeue a cell every 20ns. Consequently, we generate a token every 20ns and put it into a bucket[3]. When a packet is dequeued or dropped, the corresponding number of tokens is removed from the bucket.

[2]Note that, in practice, packets are not divided into cells within the switch. The division into cells is merely a theoretical assumption used to analyze the token generation process.



(a) Occamy, $\alpha = 1$ **(b)** Occamy, $\alpha = 4$ **(c)** DT, $\alpha = 1$ **(d)** DT, $\alpha = 4$

Figure 10. [P4] Queue length evolution



(a) Scenario **(b)** $\alpha = 1$ **(c)** $\alpha = 2$ **(d)** $\alpha = 4$

Figure 11. [P4] Ability of absorbing traffic bursts with different αs

To guarantee line-rate forwarding, the TX module is always allowed to dequeue packets and remove tokens from the bucket even without enough available tokens (in this case the number of tokens will become negative). In comparison, only with enough available tokens can the expulsion module fetch a packet. In this way, the expulsion module can only utilize the redundant memory bandwidth for packet expulsion. ② The expulsion module selects an over-allocated queue and drops the head packet. To achieve this, we maintain a bitmap to track which queues are over-allocated. Then the expulsion module iterates over the over-allocated queues and drops a packet at the head of the queue (if there are enough tokens).

Differences from an ASIC switch. As we don't control the operating system, we cannot exactly control the timing of the software prototype as a real ASIC-based hardware switch. As a result, the software prototype is not identical to an ASIC-based hardware switch. The differences lie in: ① fixed-priority arbiter that can ensure the priority access to buffer for output scheduler, and ② packet dequeue that fetches PDs, cell pointers, and cells from different memory in a pipelined manner. Nevertheless, the software prototype implements Occamy's preemption behavior, albeit with some differences. Specifically, the software prototype can quickly detect whether the queue length exceeds the threshold and actively vacate the over-allocated buffer. Additionally, it employs a token-bucket-based mechanism to ensure that preemption only utilizes redundant memory bandwidth, which can reflect the intended preemption logic.

[3]In practice, it is difficult to achieve such a fine granularity. Instead, we generate $d/20$ tokens every time, where d is the interval between token generations.

6 Evaluation

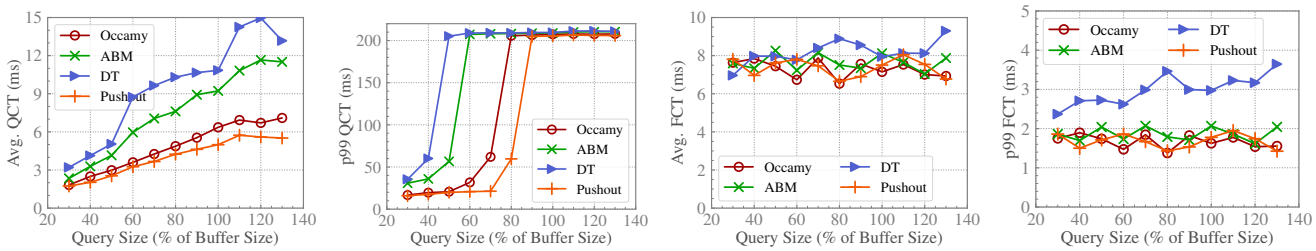
In this section, we evaluate Occamy with both testbed experiments (§6.1, §6.2, and §6.3) and ns-3 simulations (§6.4).

6.1 P4-based Hardware Prototype

Testbed Setup. We build a small testbed with one sender and two receivers. The topology is shown in Figure 11(a). Each server has a 16-core 2.10GHz x86 CPU and 16GB memory. The sender is equipped with a dualport 100GbE NIC, connected to the switch with two 100Gbps links. Each receiver is equipped with an 10GbE NIC, connected to the switch with a 10Gbps link. The switch has a 6.5Tbps Intel Tofino switch chip. The traffic are generated by `Pktgen-DPDK` [66].

We consider a scenario shown in Figure 11(a). At the beginning, we generate long-lived traffic from the sender to receiver 1. After a while, we generate bursty traffic from the sender to receiver 2. The long-lived traffic is ongoing throughout the experiment, while the bursty traffic only lasts for ~0.8μs. The long-lived traffic and bursty traffic are sent from different NIC ports and heading for different receivers so as not to affect each other beyond the switch.

Occamy is agile and can quickly adjust buffer allocation. Figure 10 shows the queue length evolution of Occamy and DT. Figure 10(a) and Figure 10(b) show that Occamy can quickly adjust buffer allocation as the bursty traffic arrives, and the bursty traffic does not drop packets until it is allocated with the fair-share amount of buffer. In comparison, Figure 10(c) and Figure 10(d) show that only with enough free buffer reservation (*i.e.*, $\alpha = 1$) can DT adjust



(a) Query: Average QCT **(b)** Query: 99th QCT **(c)** Overall Bg: Average FCT **(d)** Small Bg: 99th FCT

Figure 12. Ability of absorbing traffic bursts

buffer allocation in time. Without enough buffer reservation (*i.e.,* Figure 10(d)), DT is not able to quickly release the over-allocated buffer for the long-lived traffic. As a result, the packets of the newly arrived bursty traffic are dropped before obtaining the deserved buffer allocation.

Occamy can absorb more traffic bursts. Figure 11 shows the loss rate of bursty traffic with different burst sizes. We make two observations:

(1) Occamy can absorb more bursty traffic by avoiding the anomalous behavior. With the same amount of free buffer reservation, Occamy can absorb more bursty traffic than DT. For example, Figure 11(d) shows that Occamy can absorb 57% more bursty traffic than DT with an α of 4. This is attributed to the agility of Occamy, *i.e.,* it can quickly make buffer for newly arrived bursty traffic. In comparison, DT is not agile enough, thereby reluctantly dropping packets of bursty traffic before getting fair buffer share.

(2) Occamy can absorb more bursty traffic by higher buffer efficiency. Figure 11(b) and Figure 11(d) show that, with $\alpha = 4$, Occamy can absorb 29% more bursty traffic than that with $\alpha = 1$. In comparison, DT absorbs 12% less bursty traffic with $\alpha = 4$ than that with $\alpha = 1$. This is because Occamy is agile even with a little free buffer reservation, and thus can utilize more buffer for burst absorption. In comparison, DT is not agile enough without enough free buffer reservation, and thus cannot improve burst absorption by higher buffer efficiency.

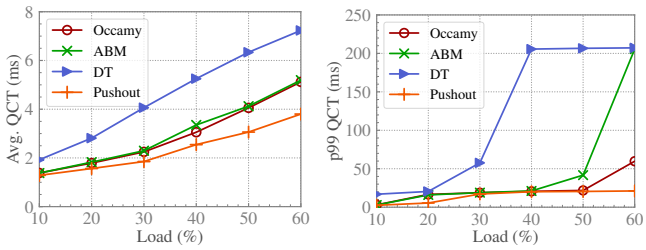
6.2 DPDK-based Software Prototype

Testbed Setup. We build a small testbed with eight hosts. Each host is a Dell PowerEdge R730 server with a 16-core Intel Xeon E5-2620 2.1GHz CPU and 16GB DDR4 memory. Each server is equipped with an Intel 82599 10GbE NIC, connected to a DPDK-based software switch with a 10Gbps link. The software switch is emulated by a server with two Intel XL710 Quad Port 10GbE NICs. Following the features of Broadcom Tomahawk switch chip [84], the switch contains 5.12KB buffer-per-port-per-Gbps (*i.e.,* 410KB in total). Different from the testbed in P4-based evaluations, the traffic is sent through Linux kernel's network stack, which allow us to use DCTCP as the congestion control algorithm. The ECN threshold is set to 65 packets as suggested by [5]. We compare Occamy with DT, ABM [1], and Pushout, where ABM is a recently proposed non-preemptive BM that can ensure performance isolation. Unless otherwise specified, we set $\alpha = 1$ for DT (as suggested by [27]), $\alpha = 2$ for ABM to achieve better performance[4], and $\alpha = 8$ for Occamy.

Occamy can absorb more traffic bursts. We use the open-source traffic generator [16] to generate two kinds of traffic. ① Bursty query traffic: A client on each host periodically sends queries to 16 servers on other hosts (each host runs 2 servers). Receiving the query, each server will generate a response to the client. The total amount of response data (termed as query size) is varied to generate different burst sizes. The queries are generated according to a Poisson process, with a network load of 1%. ② Background traffic: This kind of traffic follows a 1-to-1 pattern. We generate background flows according to a Poisson process. The sender and receiver are randomly chosen, and the flow size follows the web-search distribution [5]. The load of background traffic is 50%.

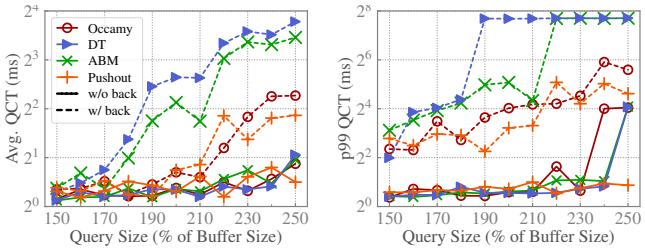
Figure 12 shows the query completion time (QCT) of the query traffic and the flow completion time (FCT) of the background traffic. Here, FCT refers to the time required to transmit a flow, while QCT refers to the time required to transmit all flows in a query task. We make two observations. (1) Occamy can significantly improve the performance of query traffic. Figure 12(a) shows that, compared to DT and ABM, Occamy can reduce the average QCT by up to ~55% and ~42%, respectively. Figure 12(b) shows that Occamy can avoid retransmission timeout (RTO) with the burst size below 80% of the buffer size, which is 33% and 60% higher than DT and ABM, respectively. (2) Occamy's performance gain of burst absorption does not come at the cost of hurting background flows. Figure 12(c) shows that the average FCT of Occamy is comparable to that of DT and ABM. Figure 12(d) shows that the 99th percentile FCT for small flows (*i.e.,* flow size < 100KB) is comparable to ABM and up to ~57% shorter than DT. This is because Occamy only drops the over-allocated buffer of background flows, rather than seizing their deserved buffer.

[4]We do not set a larger α for unscheduled packets because we assume that the end hosts remain untouched.



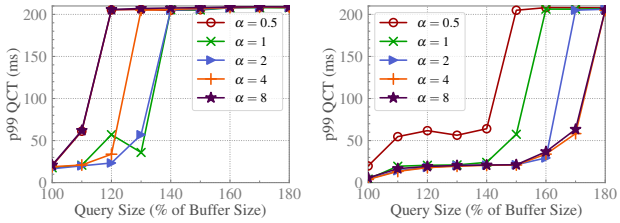
(a) Average QCT **(b)** 99th QCT

Figure 13. Performance isolation



(a) Average QCT **(b)** 99th QCT

Figure 14. Mitigation of buffer choking



(a) DT: 99th QCT **(b)** Occamy: 99th QCT

Figure 15. Impact of α

Occamy can ensure performance isolation. For this experiment, we set up our switch with two service queues per port, which are fairly scheduled with Deficit Round Robin (DRR). The background traffic and query traffic are assigned to different queues for performance isolation. The background flows use CUBIC as their congestion control algorithm. Other settings remain unchanged. We vary the load of background traffic to examine its impact on the performance of query traffic.

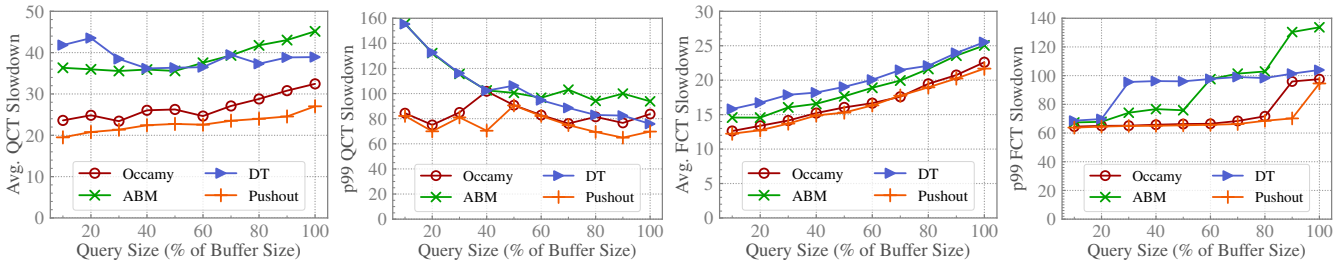
Figure 13(a) and Figure 13(b) show the average QCT and 99th percentile QCT of the query traffic, respectively. Figure 13(b) shows that, as the load of background traffic increases, DT and ABM can result in RTOs for query traffic, which significantly extends the QCT. Since the background traffic and the query traffic are put into different queues, the RTOs mainly result from the inability to quickly adjust the buffer allocation. In comparison, Occamy can quickly adjust the buffer allocation by actively expelling the packets for over-allocated queues, achieving significantly lower 99th percentile QCT.

Occamy can effectively mitigate the buffer choking problem. For this experiment, we set up our host and switch with two priority queues. The query flows are assigned with the high priority (HP), whereas the background flows are assigned with the low priority (LP). For the HP queue, we set the α of DT, ABM, and Occamy to 8 so that more buffer is allocated to HP traffic. For the LP queue, we set the α to 1. Other settings remain unchanged. We let a host receive both query flows and background flows from the other hosts, so that two priority queues are congested at the same port simultaneously. Hopefully, the LP background traffic should not affect the HP query traffic.

Figure 14(a) and Figure 14(b) show the average and 99th percentile QCT, respectively. The solid line depicts the QCT without background traffic, whereas the dashed line depicts the QCT with background traffic. We can observe that Occamy achieves similar performance to Pushout, *i.e.,* the background traffic does not affect the performance of query traffic much even though they share the same buffer. In contrast, the background traffic significantly extends the QCT of DT and ABM. Figure 14(a) shows that, the background traffic can extend the average QCT of DT by up to ~6.6×. Figure 14(b) shows that the background traffic can extend the 99th percentile QCT of DT by up to ~60×. ABM achieves better QCT than DT, which is expected since ABM restricts the queue length of LP queues. However, it cannot radically address the buffer choking problem. Figure 14(a) shows that the background traffic can extend the average QCT of ABM by up to ~5.7×. This is because ABM is also non-preemptive and relies on natural queue drain to adjust buffer allocation.

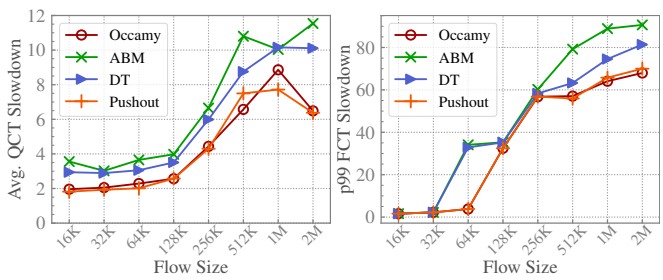
6.3 Parameter Settings

We use our DPDK-based testbed to explore the impact of parameter α. The same as previous experiments, we put background traffic and query traffic into two queues, which are fairly scheduled by DRR. Other settings are kept unchanged. Figure 15 shows the 99th percentile QCT of DT and Occamy with different αs. Figure 15(a) shows that DT achieves better performance with $\alpha = 1$ or $\alpha = 2$. The performance is degraded with either smaller α or larger α. This is because DT is more inefficient with small α, and is prone to anomalous behavior with large α. In comparison, Figure 15(b) shows that Occamy achieves better performance with larger α. This is because Occamy can avoid anomalous behavior under the support of fast buffer expulsion, and thus can use a large α to improve efficiency. Nonetheless, we also observe that the performance improvement is undermined with α larger than 4. Specifically, Figure 15(b) shows that the performance with $\alpha = 8$ is very close to that with $\alpha = 4$. This is because the improvement of buffer utilization is small with large α (as analyzed in §4.4). Thus, we recommend setting α to 8.



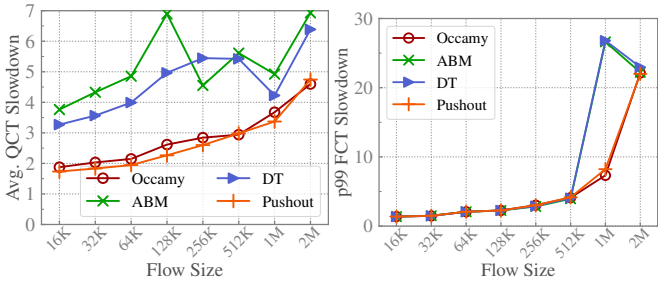
(a) Query: Average QCT **(b)** Query: 99th QCT **(c)** Overall Bg: Average FCT **(d)** Small Bg: 99th FCT

Figure 16. Query Completion Time (QCT) and Flow Completion Time (FCT)



(a) Query: Average QCT **(b)** Overall Bg: 99th FCT

Figure 17. Performance with all-to-all traffic



(a) Query: Average QCT **(b)** Overall Bg: 99th FCT

Figure 18. Performance with all-reduce traffic

6.4 Large-scale Simulations

Topology. We simulate a 128-host leaf-spine topology with 8 spine switches and 8 leaf switches. Each leaf switch is connected to 16 hosts with 100Gbps down links and 8 spine switches with 100Gbps up links. The base RTT across spine is 80μs. We employ ECMP for multi-path load balancing. To emulate the Broadcom Tomahawk switch chip, we make every 8 ports share 4MB buffer [17, 84]. As a result, the leaf switches contain 12MB buffer in total, whereas the spine switches contain 8MB buffer.

Workload. We generate query traffic and background traffic similarly to the previous DPDK-based experiments. Each server generates 200 queries per second. The load of background traffic is 90%. We use DCTCP [5] as the congestion control algorithm, with ECN threshold set to 720KB (*i.e.,* 0.72BDP) as suggested by [17]. The minimum RTO is set to 5ms.

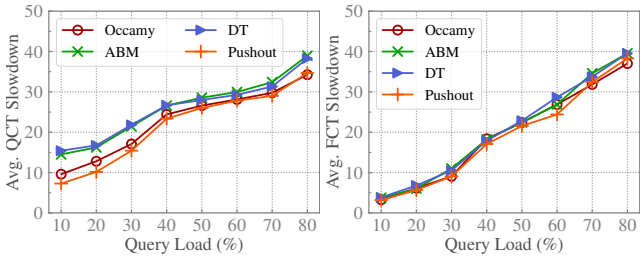
Performance. Figure 16 shows the performance of query traffic (in terms of QCT slowdown) and background traffic (in terms of FCT slowdown). Here, slowdown is the ratio between the actual value and the ideal value without other traffic. We make two observations. (1) Occamy can improve the performance of bursty query traffic. Figure 16(a) shows that Occamy reduces the average QCT slowdown of DT and ABM by up to ~44% and ~36%, respectively. Figure 16(b) shows that Occamy reduces the 99th percentile QCT slowdown of DT and ABM by up to ~46%. (2) Occamy is also beneficial to the background flows. Figure 16(a) shows that Occamy reduces the average FCT slowdown of DT and ABM by up to ~20% and ~13%, respectively. Figure 16(b) shows that, for small flows, Occamy reduces the 99th percentile FCT slowdown of DT and ABM by up to ~32%.

Performance with all-to-all/all-reduce traffic. To examine Occamy's performance in prevalent AI scenarios, we generate all-to-all and all-reduce background traffic. For all-to-all traffic, every host sends the same amount of data to all other hosts. For all-reduce traffic, we generate flows based on the prevailing double binary tree algorithm [69], with all flows having an identical size.

Figure 17 and Figure 18 show that Occamy can improve both the QCT for query traffic and FCT for background traffic. Specifically, in the all-to-all scenario, Occamy can improve the average QCT and 99th percentile FCT of DT by up to ~33% and ~88%, respectively. In the all-reduce scenario, Occamy can improve the average QCT and 99th percentile FCT of DT by up to ~48% and ~73%, respectively.

Performance with higher query traffic rate. In previous scenarios, the load of query traffic is not very high. In this part, we examine Occamy's performance with higher query traffic rate. We vary the load of query traffic from 10% (*i.e.,* 186 queries per second per server) to 80% (*i.e.,* 1,490 queries per second per server)[5]. The query size is 80% of the

[5]Load = # queries_per_second × query_size × oversubscription_ratio / link_capacity, where query_size=3.2MB, oversubscription_ratio=2, and link_capacity=100Gbps.



(a) Query: Average QCT **(b)** Overall Bg: Average FCT

Figure 19. Performance with higher query traffic load

buffer size (*i.e.,* 3.2MB). The load of background traffic is 10%. As a result, the overall load varies from 20% to 90%. Results with higher load can be found in [73].

Figure 19(a) shows that Occamy can improve the average QCT of DT and ABM by up to ~38% and ~34%, respectively. Besides, the performance improvements of both Occamy and Pushout are more notable at lower query load. This is because DT's inefficiency is more pronounced with fewer active ports [27], and thus Occamy has more space to improve performance. Figure 19(b) shows that BM has little influence on the performance of background traffic. This is because the background traffic is light-loaded, and thus has minimal requirement on buffer.

7 Related Work

Preemptive BM. Preemptive schemes (which are also called Pushout) can arbitrarily overwrite or evict packets residing in the buffer. They had been proved to be optimal, but were considered to be hard to implement. SDR [82] accepts a packet whenever there is free buffer space and either drops the arriving packet or swaps out an accepted packet when the buffer becomes full. DoD [89] proposed to purge the longest queue when the buffer becomes full. POT [31] pushes out an accepted packet only when the length of corresponding queue for the newly arrived packet is below a certain threshold. LossPass [56] proposes to evict packets of large flows to make room for newly arrived small flows. Some other work [29, 60] studied the implementation of Pushout. Choudhury and Hahne [29] proposed an implementation of Pushout that can reduce the required memory footprint for maintaining queues. QPO [60] proposed to discard packets in the near-longest queue, which is easy to be maintained. However, it still entails complex enqueue operations. Several studies [25, 30, 57, 79] focused on Pushout schemes with multiple (loss) priorities. However, as analyzed in §2.2, these schemes can introduce unacceptable implementation overhead.

Non-preemptive BM. Non-preemptive schemes only drop packets before they enter into the buffer. They are simple to implement. Typically, they use threshold(s) to restrict the length of each queue. SMXQ [50], SMA [68], SMQMA [54], and Harmonic [55] use static threshold(s) to provide a minimum buffer guarantee and restrict the maximum queue length for fairness. To adapt BM to varying traffic loads, DT [26–28, 40], PSPP [91], and DFT [43] dynamically adjust the threshold(s) based on the buffer occupancy and queue length. Among them, DT [26–28] has become the de facto BM in commodity switch chips due to its simplicity. Other studies [14, 15, 80, 83] proposed to leverage adaptive control to dynamically adjust the thresholds based on the traffic load. In recent years, as the buffer becomes increasingly insufficient, the importance of BM has been brought to attention again. Many BM schemes have been proposed to improve burst absorption and adaptivity. EDT [71], FAB [10], TDT [49], Smartbuf [67], and Protean [6] improve the ability of burst absorption by allocating more buffer to bursty traffic. NDT [88] leverages deep reinforcement learning to dynamically adjust the parameters of DT as the traffic pattern changes. ABM [1] adapts the threshold based on both total buffer occupancy and queue drain time to achieve predictable burst tolerance, performance isolation, and bounded buffer drain time simultaneously. Reverie [2] uses the moving average queue length to improve burst absorption. However, these non-preemptive schemes cannot quickly adjust the buffer allocation as the over-allocated buffer can only be released by naturally sending out the traffic. Credence [3] makes packet drop decisions based on machine-learning-based predictions of future packet arrivals. However, it requires ML algorithms in the dataplane, which can incur high overhead.

8 Conclusion

In this paper, we argue that today's BM requires agile adjustment of buffer allocation facing dynamic traffic, while the agility of the de facto BM is limited by its non-preemptive nature. On the other hand, although preemptive BMs were considered as difficult to implement in history, we find that the new advances in switch chips have opened the door to realizing preemptive mechanisms. We propose Occamy, which utilizes the redundant memory bandwidth to quickly expel the packets for the over-allocated queues. Experiment and simulation results show that Occamy significantly outperforms non-preemptive BMs in terms of burst absorption, performance isolation, and buffer choking mitigation.

Acknowledgments

We would like to thank our shepherd, Gaël Thomas, and the anonymous reviewers for their helpful feedback. We thank Yuxuan Li for the help for the evaluations on Design Compiler. This work is supported in part by National Natural Science Foundation of China under Grant No. 62372363, Grant No. 62132007, Grant No. 62172323, and Grant No. 624722451. Wanchun Jiang is the corresponding author.

References

[1] Vamsi Addanki, Maria Apostolaki, Manya Ghobadi, Stefan Schmid, and Laurent Vanbever. 2022. ABM: Active Buffer Management in Datacenters. In *ACM SIGCOMM*.

[2] Vamsi Addanki, Wei Bai, Stefan Schmid, and Maria Apostolaki. 2024. Reverie: Low Pass Filter-Based Switch Buffer Sharing for Datacenters with RDMA and TCP Traffic. In *USENIX NSDI*.

[3] Vamsi Addanki, Maciej Pacut, and Stefan Schmid. 2024. Credence: Augmenting Datacenter Switch Buffer Sharing with ML Predictions. In *USENIX NSDI*.

[4] Anurag Agrawal and Changhoon Kim. 2020. Intel Tofino2 – A 12.9Tbps P4-Programmable Ethernet Switch. In *Hot Chips*.

[5] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *ACM SIGCOMM*.

[6] Hamidreza Almasi, Rohan Vardekar, and Balajee Vamanan. 2023. Protean: Adaptive Management of Shared-Memory in Datacenter Switches. In *IEEE INFOCOM*.

[7] [n. d.] AMD Alveo U50 Data Center Accelerator Card. https://www.amd.com/en/products/accelerators/alveo/u50/a-u50-p00g-pq-g.html.

[8] [n. d.] AMD Vivado Design Suite. https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vivado.html.

[9] Maria Apostolaki, Vamsi Addanki, Manya Ghobadi, and Laurent Vanbever. 2021. FB: A Flexible Buffer Management Scheme for Data Center Switches. *CoRR*, abs/2105.10553. https://arxiv.org/abs/2105.10553.

[10] Maria Apostolaki, Laurent Vanbever, and Manya Ghobadi. 2019. FAB: Toward Flow-Aware Buffer Sharing on Programmable Switches. In *Workshop on Buffer Sizing*.

[11] Alex Arcilla and Tony Palmer. 2019. Broadcom Trident 3 Platform Performance Analysis. White Paper. Broadcom, (May 2019). https://docs.broadcom.com/doc/12395356.

[12] 2013. Arista 7050X3 Series Switch Architecture. White Paper. Arista, (Oct. 2013). https://www.arista.com/assets/data/pdf/Whitepapers/7050X3_Architecture_WP.pdf.

[13] Mutlu Arpaci and John A. Copeland. 2000. Buffer Management for Shared-memory ATM Switches. *IEEE Communications Surveys & Tutorials*, 3, 1, 2–10.

[14] Giuseppe Ascia, Vincenzo Catania, and Daniela Panno. 2002. An Efficient Buffer Management Policy based on an Integrated Fuzzy-GA Approach. In *IEEE INFOCOM*, 1042–1048.

[15] Giuseppe Ascia, Vincenzo Catania, and Daniela Panno. 2005. An Evolutionary Management Scheme in High-Performance Packet Switches. *IEEE/ACM Transactions on Networking*, 13, 2, 262–275.

[16] Wei Bai, Li Chen, Kai Chen, and Haitao Wu. 2016. Enabling ECN in Multi-Service Multi-Queue Data Centers. In *USENIX NSDI*.

[17] Wei Bai, Shuihai Hu, Kai Chen, Kun Tan, and Yongqiang Xiong. 2021. One More Config is Enough: Saving (DC)TCP for High-Speed Extremely Shallow-Buffered Datacenters. *IEEE/ACM Transactions on Networking*, 29, 2, 489–502.

[18] Wei Bai, Shuihai Hu, Kai Chen, Kun Tan, and Yongqiang Xiong. 2020. One More Config is Enough: Saving (DC)TCP for High-speed Extremely Shallow-buffered Datacenters. In *IEEE INFOCOM*.

[19] Wei Bai et al. 2023. Empowering Azure Storage with RDMA. In *USENIX NSDI*.

[20] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the Killer Microseconds. *Communications of the ACM*, (Mar. 2017), 48–54.

[21] Broadcom. 2021. *BCM88800 Traffic Management Architecture*. Design Guide. Broadcom. (Feb. 2021). https://docs.broadcom.com/doc/88800-DG1-PUB.

[22] 2023. Broadcom Now Shipping World's First 51.2 Tbps Switch in Production Volume. Broadcom. (Mar. 15, 2023). https://investors.broadcom.com/news-releases/news-release-details/broadcom-now-shipping-worlds-first-512-tbps-switch-production.

[23] 2022. Broadcom Ships Tomahawk 5, Industry's Highest Bandwidth Switch Chip to Accelerate AI/ML Workloads. Broadcom. (Aug. 16, 2022). https://investors.broadcom.com/news-releases/news-release-details/broadcom-ships-tomahawk-5-industrys-highest-bandwidth-switch.

[24] Rakesh Chopra. 2023. Cisco Silicon One Breaks the 51.2 Tbps Barrier. Cisco. (June 20, 2023). https://blogs.cisco.com/sp/cisco-silicon-one-breaks-the-51-2-tbps-barrier.

[25] Abhijit K. Choudhury and Ellen L. Hahne. 2000. A Simulation Study of Space Priorities in a Shared Memory ATM Switch. *Journal of High Speed Networks*, 9, 2, 67–87.

[26] Abhijit K. Choudhury and Ellen L. Hahne. 2002. Dynamic Queue Length Thresholds for Multiple Loss Priorities. *IEEE/ACM Transactions on Networking*, 10, 3, (June 2002), 368–380.

[27] Abhijit K. Choudhury and Ellen L. Hahne. 1998. Dynamic Queue Length Thresholds for Shared-memory Packet Switches. *IEEE/ACM Transactions on Networking*, 6, 2, 130–140.

[28] Abhijit K. Choudhury and Ellen L. Hahne. 1996. Dynamic queue length thresholds in a shared memory atm switch. In *IEEE INFOCOM*.

[29] Abhijit K. Choudhury and Ellen L. Hahne. 1996. New Implementation of Multi-priority Pushout for Shared Memory ATM Switches. *Computer Communications*, 19, 3, 245–256.

[30] Abhijit K. Choudhury and Ellen L. Hahne. 1993. Space Priority Management in a Shared Memory ATM Switch. In *IEEE GLOBECOM*.

[31] Israel Cidon, Leonidas Georgiadis, Roch Guerin, and Asad Khamisy. 1995. Optimal Buffer Sharing. *IEEE Journal on Selected Areas in Communications*, 13, 7, 1229–1240.

[32] 2013. Cisco Nexus 3100 Platform Switch Architecture. White Paper. Cisco, (Oct. 2013). https://www.cisco.com/c/dam/assets/events/i/interop-ny-Cisco-Nexus3100-Switch-Architecture-Whitepaper.pdf.

[33] Cisco. 2020. *Cisco Nexus 9000 Series NX-OS Quality of Service Configuration Guide, Release 6.x*. Cisco. (Apr. 22, 2020). https://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus9000/sw/6-x/qos/configuration/guide/b_Cisco_Nexus_9000_Series_NX-OS_Quality_of_Service_Configuration_Guide.pdf.

[34] [n. d.] Cisco Silicon One G202 Data Sheet. Cisco. https://www.cisco.com/c/en/us/solutions/collateral/silicon-one/silicon-one-g202-ds.html.

[35] 2014. Congestion Management and Buffering in Data Center Networks. White Paper. Extreme Networks. http://learn.extremenetworks.com/rs/extreme/images/Congestion-Management-and-Buffering-wp.pdf.

[36] 2020. Converged Web Scale Switching and Routing Becomes a Reality. Cisco Silicon One and HBM Memory Change the Paradigm. White Paper. Cisco. https://www.cisco.com/c/dam/en/us/solutions/collateral/silicon-one/white-paper-sp-hybrid-buffer-architecture.pdf.

[37] Sujal Das and Rochan Sankar. 2012. Broadcom Smart-Buffer Technology in Data Center Switches for Cost-Effective Performance Scaling of Cloud Applications. White Paper. Broadcom, (Apr. 2012).

[38] Giorgos Dimitrakopoulos, Anastasios Psarras, and Ioannis Seitanidis. 2015. *Microarchitecture of Network-on-chip Routers. A Designer's Perspective*. Vol. 1025. Springer.

[39] 2021. Driving the Data Center into the Future. Broadcom. (June 30, 2021). https://www.broadcom.com/blog/driving-the-data-center-into-the-future.

[40] Ruixue Fan, Alexander Ishii, Brian Mark, G. Ramamurthy, and Qiang Ren. 1999. An Optimal Buffer Management Scheme with Dynamic Thresholds. In *IEEE GLOBECOM*.

[41] [n. d.] FreePDK45. https://eda.ncsu.edu/freepdk/freepdk45/.

[42] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker.

[n. d.] Network Requirements for Resource Disaggregation. In *USENIX OSDI*.

[43] B. Gazi and Z. Ghassemlooy. 2007. Dynamic Buffer Management using Per-queue Thresholds. *International Journal of Communication Systems*, 20, 5, 571–587.

[44] Ehab Ghabashneh, Yimeng Zhao, Cristian Lumezanu, Neil Spring, Srikanth Sundaresan, and Sanjay Rao. 2022. A Microscopic View of Bursts, Buffer Contention, and Loss in Data Centers. In *ACM IMC*.

[45] Prateesh Goyal, Preey Shah, Kevin Zhao, Georgios Nikolaidis, Mohammad Alizadeh, and Thomas E. Anderson. 2022. Backpressure Flow Control. In *USENIX NSDI*.

[46] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over Commodity Ethernet at Scale. In *ACM SIGCOMM*.

[47] Pankaj Gupta and Nick McKeown. 1999. Designing and Implementing a Fast Crossbar Scheduler. *IEEE Micro*, 19, 1, 20–28.

[48] Yihua He, Nitin Batta, and Igor Gashinsky. 2019. Understanding Switch Buffer Utilization in CLOS Data Center Fabric. In *Workshop on Buffer Sizing*.

[49] Sijiang Huang, Mowei Wang, and Yong Cui. 2021. Traffic-aware Buffer Management in Shared Memory Switches. In *IEEE INFOCOM*.

[50] Marek I. Irland. 1978. Buffer Management in a Packet Switch. *IEEE Transactions on Communications*, 26, 3, (Mar. 1978), 328–337.

[51] Sundar Iyer, Ramana Rao Kompella, and Nick McKeown. 2001. Analysis of a Memory Architecture for Fast Packet Buffers. In *IEEE HPSR*.

[52] Sundar Iyer and Nick McKeown. 2001. Techniques for Fast Shared Memory Switches. Stanford HPNG Technical Report TR01-HPNG-081501.

[53] Youngmi Joo and Nick McKeown. 1998. Doubling Memory Bandwidth for Network Buffers. In *IEEE INFOCOM*.

[54] Farouk Kamoun and Leonard Kleinrock. 1980. Analysis of Shared Finite Storage in a Computer Network Node Environment Under General Traffic Conditions. *IEEE Transactions on Communications*, 28, 7, (July 1980), 992–1003.

[55] Alexander Kesselman and Yishay Mansour. 2002. Harmonic Buffer Management Policy for Shared Memory Switches. In *IEEE INFOCOM*.

[56] Gyuyeong Kim and Wonjun Lee. 2022. LossPass: Absorbing Microbursts by Packet Eviction for Data Center Networks. *IEEE Transactions on Cloud Computing*, 10, 4, 2717–2728.

[57] Hans Kroner, Gerard Hebuterne, Pierre Boyer, and Annie Gravey. 1991. Priority Management in ATM Switching Nodes. *IEEE Journal on Selected Areas in Communications*, 9, 3, 418–427.

[58] Gautam Kumar et al. 2020. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *ACM SIGCOMM*.

[59] Yuliang Li et al. 2019. HPCC: High Precision Congestion Control. In *ACM SIGCOMM*.

[60] Yu-Sheng Lin and C.B. Shung. 1997. Quasi-Pushout Cell Discarding. *IEEE Communications Letters*, 1, 5, (Sept. 1997), 146–148.

[61] Kexin Liu et al. 2021. Floodgate: Taming Incast in Datacenter Networks. In *ACM CoNEXT*.

[62] 2023. Marvell Announces Cloud-Optimized 51.2 Tbps Networking Platform for AI/ML and Data Center Networks. Marvell. (Mar. 2, 2023). https://www.marvell.com/company/newsroom/marvell-next-gen-data-center-512t-networking-solution.html.

[63] 2022. NVIDIA Announces Spectrum High-Performance Data Center Networking Infrastructure Platform. NVIDIA. (Mar. 22, 2022). https://nvidianews.nvidia.com/news/nvidia-announces-spectrum-high-performance-data-center-networking-infrastructure-platform.

[64] 2022. NVIDIA Spectrum-4. 51.2 Tb/s Ethernet Switch ASIC. NVIDIA. (Apr. 2022). https://nvdam.widen.net/s/v6skdhzdrh/ethernet-switches-product-brief-gtc22-spring-spectrum-4-2169045-r8.

[65] 2022. NVIDIA Spectrum-4 Datasheet. NVIDIA. (Apr. 22, 2022). https://resources.nvidia.com/en-us-accelerated-networking-resource-library/ethernet-switches-pr?xs=425229#page=1.

[66] [n. d.] Pktgen-DPDK. https://github.com/pktgen/Pktgen-DPDK.

[67] Hamed Rezaei, Hamidreza Almasi, and Balajee Vamanan. 2021. Smartbuf: An Agile Memory Management for Shared-Memory Switches in Datacenters. In *IEEE/ACM IWQoS*.

[68] Marc A. Rich and Mischa Schwartz. 1977. Buffer Sharing in Computer-Communication Network Nodes. *IEEE Transactions on Communications*, 25, 9, (Sept. 1977), 958–970.

[69] Peter Sanders, Jochen Speck, and Jesper Larsson Träff. 2009. Two-tree Algorithms for Full Bandwidth Broadcast, Reduction and Scan. *Parallel Computing*, 35, 12, 581–594.

[70] Rich Seifert and Jim Edwards. 2008. *The All-New Switch Book: The Complete Guide to LAN Switching Technology*. (2nd ed.). Wiley.

[71] Danfeng Shan, Wanchun Jiang, and Fengyuan Ren. 2015. Absorbing Micro-burst Traffic by Enhancing Dynamic Threshold Policy of Data Center Switches. In *IEEE INFOCOM*.

[72] Danfeng Shan, Yuqi Liu, Tong Zhang, Yifan Liu, Yazhe Tang, Hao Li, and Peng Zhang. 2023. Less is More: Dynamic and Shared Headroom Allocation in PFC-enabled Datacenter Networks. In *IEEE ICDCS*.

[73] Danfeng Shan et al. 2025. Occamy: A Preemptive Buffer Management for On-chip Shared-memory Switches. (2025). https://arxiv.org/abs/2501.13570 arXiv: 2501.13570 [cs.NI].

[74] Erfan Sharafzadeh, Sepehr Abdous, and Soudeh Ghorbani. 2023. Understanding the impact of host networking elements on traffic bursts. In *USENIX NSDI*.

[75] Eung S. Shin, Vincent J. J. Mooney III, and George F. Riley. 2002. Round-robin Arbiter Design and Generation. In *ISSS*.

[76] Arjun Singh et al. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *ACM SIGCOMM*.

[77] Jaeyong Song, Jinkyu Yim, Jaewon Jung, Hongsun Jang, Hyung-Jin Kim, Youngsok Kim, and Jinho Lee. 2023. Optimus-CC: Efficient Large NLP Model Training with 3D Parallelism Aware Communication Compression. In *ACM ASPLOS*.

[78] [n. d.] Synopsys Design Compiler. https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html.

[79] Leandros Tassiulas, Yao Chung Hung, and Shivendra S. Panwar. 1994. Optimal Buffer Control During Congestion in an ATM Network Node. *IEEE/ACM Transactions on Networking*, 2, 4, (Aug. 1994), 374–386.

[80] A. K. Thareja and S. K. Tripathi. 1984. Buffer Sharing in Dynamic Load Environment. In *IEEE INFOCOM*.

[81] Ashok K. Thareja and Ashok K. Agrawala. 1981. On the Design of Optimal Policy for Sharing Finite Buffers. Tech. rep. TR-1081. Department of Computer Science, University of Maryland, (July 1981).

[82] Ashok K. Thareja and Ashok K. Agrawala. 1984. On the Design of Optimal Policy for Sharing Finite Buffers. *IEEE Transactions on Communications*, 32, 6, 737–740.

[83] D. Tipper and M.K. Sundareshan. 1988. Adaptive Policies for Optimal Buffer Management in Dynamic Load Environments. In *IEEE INFOCOM*.

[84] [n. d.] Tomahawk. https://people.ucsc.edu/~warner/Bufs/tomahawk.

[85] [n. d.] Tomahawk 5 / BCM78900 Series. 51.2 Tb/s StrataXGS® Tomahawk® 5 Ethernet Switch Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm78900-series.

[86] Juniper. 2024. *Traffic Management User Guide (QFX Series Switches and EX4600 Switches)*. User Guide. Juniper. (June 2024). https://www.juniper.net/documentation/us/en/software/junos/traffic-mgmt-qfx/traffic-mgmt-qfx.pdf.

[87] 2018. Understanding the Alpha Parameter in the Buffer Configuration of Mellanox Spectrum Switches. Mellanox. (Dec. 2018). https://support.mellanox.com/s/article/howto-configure-mellanox-spectrum-switch-for-lossless-roce.

[88] Mowei Wang, Sijiang Huang, Yong Cui, Wendong Wang, and Zhenhua Liu. 2022. Learning Buffer Management Policies for Shared Memory Switches. In *IEEE INFOCOM*.
[89] Sherry X. Wei, Edward J. Coyle Coyle, and Man-Tung T. Hsiao. 1991. An Optimal Buffer Management Policy for High-Performance Packet Switching. In *IEEE GLOBECOM*.
[90] Bob Wheeler. 2019. Tomahawk 4 Switch First to 25.6Tbps. Broadcom Doubles 400Gbps Ports With Unprecedented 512 Serdes. Microprocessor Report. The Linley Group, (Dec. 2019). https://docs.broadcom.com/doc/12398014.
[91] Ruey-Bin Yang, Yuan-Sun Chu, Ming-Cheng Liang, and Cheng-Shong Wu. 2002. Dynamic Thresholds Buffer Management in a Shared Buffer Packet Switch. In *IEEE International Conference on High Speed Networks and Multimedia Communication*.
[92] Zhuolong Yu, Chuheng Hu, Jingfeng Wu, Xiao Sun, Vladimir Braverman, Mosharaf Chowdhury, Zhenhua Liu, and Xin Jin. 2021. Programmable Packet Scheduling with a Single Queue. In *ACM SIGCOMM*.
[93] Gaoxiong Zeng, Jianxin Qiu, Yifei Yuan, Hongqiang Liu, and Kai Chen. 2021. FlashPass: Proactive Congestion Control for Shallow-buffered WAN. In *IEEE ICNP*.
[94] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. 2017. High-resolution Measurement of Data Center Microbursts. In *ACM IMC*.

A Artifact Appendix

A.1 Abstract

The artifacts associated with this paper consist of five main components, running on five testbeds: (1) Experiments conducted on a commodity Huawei switch, (2) Experiments conducted on a P4-based hardware prototype, (3) Experiments conducted on a DPDK-based software prototype, (4) Experiments conducted using the ns-3 simulator, and (5) Experiments based on Verilog code.

A.2 Description & Requirements

A.2.1 How to access. The code as well as instructions to reproduce the results can be found at https://github.com/ants-xjtu/Occamy or on Zenodo with DOI number 10.5281/zenodo.14871138.

A.2.2 Hardware dependencies.

- A switch supporting the following features (Huawei CE6865 switch in our testbed):
  - Eight 40GbE ports
  - Dynamic buffer allocation via DT, with α tunable.
  - The packet buffer size can be configured
  - At least 8 queues, supporting strict priority and weighted round-robin scheduling.
  - ECN marking
- A programmable switch with an Intel Tofino switch chip (Edgecore Wedge 100BF-32X switch in our testbed)
- A server equipped with a dual-port 100GbE NIC (NVIDIA ConnectX-6 Dx in our testbed)
- Eight servers, each equipped with an 10GbE NIC (Intel 82599 in our testbed)
- Four servers, each equipped with a dual-port 40GbE NIC (Intel XL710 in our testbed)
- A server equipped with eight 10Gbps NIC ports (two Intel XL710 Quad Port 10GbE NICs in our testbed)
- A server with at least 256GB memory

A.2.3 Software dependencies.

- Synopsys Design Compiler 2016, running on Ubuntu 22.04.
- AMD Vivado Design Suite 2024.1, running on Windows 11 or Ubuntu 22.04.
- Intel DPDK 24.11 and `pktgen-dpdk` 24.10.3.
- iPerf3
- Traffic Generator: https://github.com/Hijack8/TrafficGenerator

A.2.4 Benchmarks. None.

A.3 Set-up

The instructions to set up the testbed is detailed in https://github.com/ants-xjtu/Occamy. There are five testbeds:

- **(T1)**: Four servers, each connected to a Huawei CE6865 switch via two 40GbE links. More details in https://github.com/ants-xjtu/Occamy/tree/eurosys25-artifacts/exp/motivation.
- **(T2)**: Three servers, connected to an Intel Tofino switch via 10Gbps/100Gbps links. More details in https://github.com/ants-xjtu/Occamy/tree/eurosys25-artifacts/exp/p4.
- **(T3)**: Eight servers, each connected to a DPDK-based software switch via a 10Gbps link. More details in https://github.com/ants-xjtu/Occamy/tree/eurosys25-artifacts/exp/dpdk.
- **(T4)**: An Ubuntu 24.04 server with at least 256GB memory to run ns-3 simulator. More details in https://github.com/ants-xjtu/Occamy-Simulation.
- **(T5)**: A server with AMD Vivado Design Suite 2024.1 and Synopsys Design Compiler 2016 installed. More details in https://github.com/ants-xjtu/Occamy/tree/eurosys25-artifacts/exp/verilog.

A.4 Evaluation workflow

A.4.1 Major Claims.

- (C1): When reproducing Figure 5, the receiver should be run on a high-performance server, as it must handle massive flows at an extremely high rate (*i.e.,* seven 40Gbps senders). A high-performance server is essential to mitigate CPU bottlenecks. In our testbed, the receiver is equipped with an Intel Core i9-14900K CPU and 64GB of memory.
- (C2): The ns-3 simulator is memory-consuming. When reproducing the simulation results, the server should be equipped with at least 256GB memory.

- (C3): The server to run Intel DPDK should contain at least 12 CPU cores and 64GB memory.

A.4.2 Experiments.

Experiment (E1): [Performance degradation due to anomalous behavior]: The experiments are conducted on testbed T1, reproducing Figure 5. The detailed instructions can be found at https://github.com/ants-xjtu/Occamy/tree/eurosys25-artifacts/exp/motivation.

Experiment (E2): [Experiments on P4-based hardware prototype]: The experiments are conducted on testbed T2, reproducing Figure 10 and Figure 11. The detailed instructions can be found at https://github.com/ants-xjtu/Occamy/tree/eurosys25-artifacts/exp/p4.

Experiment (E3): [Experiments on DPDK-based software prototype]: The experiments are conducted on testbed T3, reproducing Figure 12, Figure 13, Figure 14, and Figure 15, The detailed instructions can be found at https://github.com/ants-xjtu/Occamy/tree/eurosys25-artifacts/exp/dpdk.

Experiment (E4): [Experiments on ns-3 simulator]: The experiments are conducted on testbed T4, reproducing Figure 6, Figure 16, Figure 17, Figure 18, and Figure 19. The detailed instructions can be found at https://github.com/ants-xjtu/Occamy-Simulation.

Experiment (E5): [Estimation of hardware costs]: The experiments are conducted on testbed T5, reproducing Table 1. The detailed instructions can be found at https://github.com/ants-xjtu/Occamy/tree/eurosys25-artifacts/exp/verilog.