Practical Packet Deflection in Datacenters

SEPEHR ABDOUS, Johns Hopkins University, USA
ERFAN SHARAFZADEH, Johns Hopkins University, USA
SOUDEH GHORBANI, Johns Hopkins University & Meta, USA

Bursts, sudden surges in network utilization, are a significant root cause of packet loss and high latency in datacenters. Packet *deflection,* re-routing packets that arrive at a local hotspot to neighboring switches, is shown to be a potent countermeasure against bursts. Unfortunately, existing deflection techniques cannot be implemented in today's datacenter switches. This is because, to minimize packet drops and remain effective under extreme load, existing deflection techniques rely on certain hardware primitives (*e.g.,* extracting packets from arbitrary locations in the queue) that datacenter switches do not support. In this paper, we address the implementability hurdles of packet deflection. This paper proposes heuristics for approximating state-of-the-art deflection techniques in programmable switches. We introduce *Simple Deflection* which deflects excess traffic to randomly selected, non-congested ports and *Preemptive Deflection (PD)* in which switches identify the packets likely to be selected for deflection and preemptively deflect them before they are enqueued. We implement and evaluate our techniques on a testbed with Intel Tofino switches. Our testbed evaluations show that Simple and Preemptive Deflection improve the 99^{th} percentile response times by 8× and 425×, respectively, compared to a baseline drop-tail queue under 90% load. Using large-scale network simulations, we show that the performance of our algorithms is close to the deflection techniques that they intend to approximate, *e.g.,* PD achieves 4% lower 99^{th} percentile query completion times (QCT) than Vertigo, a recent deflection technique that cannot be implemented in off-the-shelf switches, and 2.5× lower QCT than ECMP under 95% load.

CCS Concepts: • **Networks** → **Data center networks**.

Additional Key Words and Phrases: packet deflection, programmable hardware, congestion management

ACM Reference Format:
Sepehr Abdous, Erfan Sharafzadeh, and Soudeh Ghorbani. 2023. Practical Packet Deflection in Datacenters. *Proc. ACM Netw.* 1, CoNEXT3, Article 25 (December 2023), 25 pages. https://doi.org/10.1145/3629147

1 INTRODUCTION

Datacenters have long been plagued by traffic burstiness [10, 19, 36, 84]. Bursts make performance unpredictable and result in excessive queueing delay and packet drops [6, 19, 55, 82, 84]. These performance anomalies are especially problematic for today's increasingly distributed and tail-sensitive workloads with tight latency requirements such as distributed machine learning workloads [30, 50, 59]. Bursts are unpredictable and can last as short as a few microseconds, making burst management challenging [36, 64, 84]. Managing bursts is further compounded by the trend of deploying increasingly shallow buffered switches in datacenters [38]. Preventing the formation of bursts is also difficult due to the wide variety of their root causes, such as scheduling, in-network collisions, ACK compression, and segmentation offloading [10, 12, 44, 45, 48, 57, 66, 84].

Authors' addresses: Sepehr Abdous, sabdous1@jh.edu, Johns Hopkins University, USA; Erfan Sharafzadeh, erfan@cs.jhu.edu, Johns Hopkins University, USA; Soudeh Ghorbani, soudeh@cs.jhu.edu, Johns Hopkins University & Meta, USA.

2834-5509/2023/12-ART25 $15.00
https://doi.org/10.1145/3629147

Packet deflection has recently been proposed as a powerful technique to defuse bursts [6, 70, 82]. As a descendant of hot potato routing [75] and commonly deployed in networks with scarce and expensive buffer space, such as optical networks [20–22, 25, 39, 41, 49], networks-on-chip [28, 56], and ATM switching [16, 31, 47, 62, 83], deflection is having a revival in datacenter networks. With packet deflection, when a packet arrives and the output buffer is full, the switch re-routes the packet to a neighboring switch instead of dropping it. A naive implementation of deflection causes various challenges such as excessive packet reordering, congestion collapse, and head-of-line blocking for short flows. Recent proposals resolve these challenges and show the power of deflection in managing datacenter bursts [6, 70, 82]. One major hurdle remains: datacenter switches cannot implement the existing deflection techniques. This blocks their deployment despite their effective handling of bursts. This paper addresses this challenge.

We dissect the recently proposed datacenter deflection techniques, identify the elements that prevent them from being implemented in programmable switches (§2), provide heuristics to approximate these techniques (§3), implement them in Intel Tofino programmable switches (§4), and demonstrate the practicality and the power of deflection in managing datacenter bursts via extensive testbed and simulation experiments (§5). We classify deflection techniques into two broad categories: (1) *Simple Deflection,* a basic defensive mechanism against bursts that re-routes burst packets to neighboring switches, effectively defusing bursts when the degree of the load is low, and (2) *Selective Deflection* that differentiates between short bursts and long-lasting congestion, and treats them differently, *e.g.,* by deflecting microburst packets and dropping the packets of a lasting congestion event. Selective Deflection is motivated by the fact that Simple Deflection fails under load. For instance, while DIBS [82], a state-of-the-art Simple Deflection technique, reduces the average query completion time (QCT) by 3.5× compared to ECMP under 35% load, its mean QCT is 2× *higher* than ECMP under 95% load. This is because deflected packets traverse longer paths and increase the overall network utilization in an already congested network [6]. Selective Deflection's differential approach helps it remain performant under extreme load: while DIBS increases the tail (99^{th} percentile) QCT by 2× compared to ECMP under 95% load, Vertigo [6], a state-of-the-art Selective Deflection technique, improves the tail QCT by 5×. Alas, implementing Selective Deflection requires extracting already enqueued packets from switch buffers (*e.g.,* to make room for higher priority packets) which is not supported by existing datacenter switches. Similarly, to avoid simply redirecting the bursts to a different hotspot and reducing packet drops, Simple Deflection techniques such as DIBS filter out the ports toward the neighboring switches with a full buffer before deflecting a packet. This operation is key to Simple Deflection's efficiency, *e.g.,* under 55% load, filtering out congested ports reduces the number of packet drops and average latency under Simple Deflection by 46× and 3×, respectively, compared to a similar baseline that does not filter out the congested ports. Alas, this is challenging to implement since it requires visibility into buffer occupancy in the ingress pipeline,[1] maintaining a list of non-congested ports at all times, and randomly selecting from it for every incoming packet.

This paper proposes methods for implementing both categories of deflection in programmable hardware. We tackle the implementability challenges of Selective Deflection by proposing the notion of *Preemptive Deflection (PD).* The fundamental idea behind Preemptive Deflection is to identify packets that are likely to be selected for deflection in the future and preemptively deflect them as they arrive. To realize Preemptive Deflection, we design an admission control scheme that uses the relative priority of a newly arrived packet, *i.e.,* the priority of the new packet compared to

[1]While being available in recent programmable hardware, such as Tofino 2 [5], the queue occupancy information is not accessible at the ingress pipeline in the earlier releases of programmable switches, such as Tofino 1 [3, 80]. To address this, we can use special packets that recirculate in the switch and transfer the queue occupancy information from the egress to the ingress pipeline [80].

the packets inside the destination queue, and the destination queue occupancy to decide whether to enqueue or deflect it. Similarly, the admission control technique uses the packet's priority relative to the packets inside the deflection port's queue and the deflection port's queue occupancy to determine if the deflected packet should be enqueued or dropped. To make Simple Deflection implementable, we design a high-performance mechanism to share queue occupancy information between switch pipelines. To synchronize the state in ingress and egress pipelines, we use data structures that are constantly updated by the data packets, *i.e.,* packets flowing between the end-hosts in datacenters, and *control* packets, *i.e.,* packets recirculating in a switch carrying the state data. We also propose a random number generation technique that operates on the queue occupancy data structure and selects a non-congested port uniformly at random for packet deflection.

We implement and evaluate our techniques in an Intel Tofino testbed. Our results show that Simple and Preemptive Deflection improve the 99^{th} percentile response time of high-priority TCP flows by 8× and 425×, respectively, compared to a droptail queue baseline under 90% load. Using large-scale simulations, we also evaluate our designs while operating alongside a diverse set of congestion control techniques including the fastest ones (*e.g.,* Bolt [13] with a sub-RTT control loop) and the ones designed to control large-scale bursts, *e.g.,* Swift [54]. Our results show the power of our deflection techniques in improving the performance of all these congestion control algorithms, *e.g.,* with Swift under 95% load, PD outperforms ECMP, AIFO[2], and DIBS by finishing the incast queries 2×, 9×, and 3× faster, respectively. We show that packet deflection also benefits Bolt by absorbing bursts locally. Specifically, in conjunction with Bolt, preemptively deflecting packets results in 2.5× and 2× lower tail QCT than ECMP and DIBS, respectively, under 95% load. In summary, this paper enables practical implementations of packet deflection in datacenters.

2 PACKET DEFLECTION: CHALLENGES AND OPPORTUNITIES

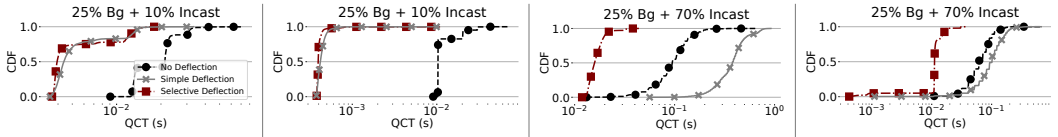
In this section, we explain (a) the fundamental idea behind deflection and the challenges that its naive realization creates, (b) how the state-of-the-art deflection techniques resolve these challenges, and (c) why these techniques cannot be implemented in programmable datacenter switches.

(a) Deflection and its challenges. Datacenter traffic studies unveil bursts, periods of high utilization, that cause packet drops and impose extra latency [10, 36, 84]. Given the pervasiveness of shallow buffered switches in datacenters, recent studies propose *packet deflection*, re-routing the packets that arrive at local hotpots instead of dropping them [6, 70, 82]. Intuitively, this is effective in defusing bursts. However, if applied naively, deflection creates various challenges: (i) if blindly redirected to other hotspots, deflected packets can still be dropped, (ii) deflected packets traverse longer paths and experience higher latency, and (iii) deflection increases the overall utilization and leads to head-of-the-line blocking in switch buffers for latency-sensitive flows.[3]

(b) Recent deflection techniques address these challenges. We divide the state-of-the-art deflection techniques into two broad groups: 1) Simple Deflection techniques such as DIBS [82] and Pabo [70] that deflect packets arriving at congested ports to neighboring switches and 2) Selective Deflection techniques such as Vertigo [6] that distinguish between short and long bursts and treat them differently, observing that many of the pitfalls of deflection stem from indiscriminate handling of short, local bursts and lasting, global congestion.

[2]AIFO [80] is an admission control paradigm that approximates SRPT scheduling in programmable switches.

[3]Packet deflection also leads to packet re-ordering and subsequently reduces throughput. Datacenter deflection techniques resolve this issue by deploying host-side techniques, *e.g.,* DIBS [82] disables the fast re-transmission process in congestion control protocols such as DCTCP which reduces the sending rate after receiving consecutive reordered packets, and Vertigo [6] deploys an ordering component below the transport layer at receiver hosts to restore the correct ordering of the packets before passing them to higher-layer protocols. Our focus in this paper is on implementing deflection in switches.



(a) 35% load – 10/40 Gbps (b) 35% load – 100 Gbps (c) 95% load – 10/40 Gbps (d) 95% load – 100 Gbps

Fig. 1. (a/b) Simple and Selective Deflection deliver low completion times for a bursty incast application under low load. (c/d) Selective Deflection retains its effectiveness under load. Both categories require hardware primitives that are not available in today's programmable datacenter switches.

In the first group, to reduce the spreading of congestion and packet loss, DIBS, a recent Simple Deflection technique, deflects packets destined for a congested port to a randomly selected port with sufficient buffer capacity. This requires filtering out the congested ports before selecting a port to deflect a packet. To show the impact of port filtering, we simulate a 2-tier leaf-spine datacenter network with 4 core switches, 8 leaf switches, and 40 servers connected to each leaf switch. We set the switch buffer capacity to 300 KB per port and use 10 Gbps and 40 Gbps links to connect leaf switches to servers and spine switches, respectively. We generate 25% background load, using the flow sizes and inter-arrival times of Facebook's cache workload [64], and 10% incast load by initiating 4000 incast queries per second. In every incast query, a client sends 100 requests to randomly selected servers asking each for 40 KB of data. We observe that DIBS reduces the average and 99^{th} percentile query completion time (QCT), *i.e.,* the time from initiating an incast query to receiving all the responses, by 3× and 2×, respectively, compared to a naive deflection scheme that does not filter out congested ports.

However, deflected packets still traverse longer paths and cause bandwidth overhead and head-of-the-line blocking which becomes problematic under load [6]. To evaluate this, we generate 25% background plus 10% and 70% incast loads by changing the arrival rate of incast events under 10/40 Gbps links (Figures 1a and 1c) and 100 Gbps links (Figures 1b and 1d). While effective under 35% load, Simple Deflection breaks as the load increases to 95%. In particular, with 10/40 Gbps links under 95% load, DIBS increases the 99^{th} percentile of flow completion times of the incast flows and the mean QCT by 43% and 51%, respectively, compared to not deflecting packets.

In the second group, to remain effective under load, Selective Deflection tries to control the bandwidth overhead of deflection by differentiating between short and long-lasting congestion and treating them differently. For instance, Vertigo [6], a recent Selective Deflection technique, deflects and, in cases of extreme congestion, drops the packets of the flows that contribute to long-lasting congestion with higher probability. This limits the extra bandwidth overhead imposed by deflected packets and helps keep the packets of short flows on shorter (and faster) paths. Figure 1 shows that Selective Deflection manages to keep the latency low under load, *e.g.,* with 100 Gbps links, Selective Deflection achieves 6× lower mean and tail QCT than the case with no packet deflection.

(c) Existing deflection techniques cannot be implemented in modern datacenter networks. Both Selective and Simple Deflection require primitives that are not easily implementable in today's programmable switch hardware. To effectively defuse burts under load, when a packet arrives and the destination queue is full, Selective Deflection extracts the packet with the largest number of remaining bytes in its corresponding flow from the queue and deflects it [6]. Unfortunately, *arbitrary packet extraction is not implementable in programmable switches* as they are restricted to strict priority FIFO queues [8, 60, 68, 80]. Simple Deflection, on the other hand, deflects packets to ports with enough buffer capacity [82], which requires generating a list of candidate ports for every incoming packet. *Generating such a list is hard given the rapid changes in ports' queue occupancy.* In the next two sections, we address these challenges by designing practical and high-performance deflection techniques and implementing them in Intel Tofino switches.

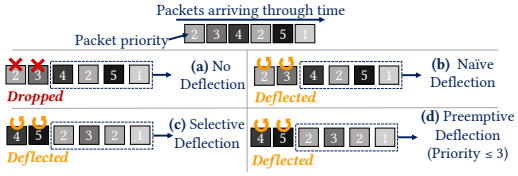


Fig. 2. Comparing four different deflection schemes on packets with priorities.

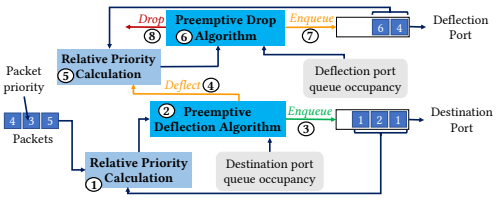


Fig. 3. Preemptive Deflection's workflow.

3 PREEMPTIVE DEFLECTION: A PRACTICAL SELECTIVE DEFLECTION TECHNIQUE

We propose *Preemptive Deflection* as an approximation of Selective Deflection. The intuition behind Preemptive Deflection is to identify the packets likely to be deflected in the future and preemptively deflect them before they enter the destination queue. We design an algorithm that computes the *relative priority* of an incoming packet, *i.e.,* its priority compared to the packets already enqueued in the destination queue, to decide if the newly arrived packet should be enqueued or deflected.

Illustrative example. Figure 2 shows the idea behind Preemptive Deflection compared to other deflection-based techniques and a no-deflection baseline. In this example, the queue capacity is equal to four packets and the numbers on packets represent their priority (lower numbers indicate higher priority). When we have no deflection **(a)**, the packets are enqueued in the order that they arrive and the last two packets, *i.e.,* packets 3 and 2, are dropped. With Naive Deflection **(b)**, *i.e.,* deflecting the incoming packet when the destination queue is full, the first four packets are enqueued and the last two are deflected. This imposes an additional latency to high-priority packets due to traversing longer paths in the network. Under Selective Deflection **(c)**, the first four packets are enqueued as they arrive. When the last two packets arrive, two low-priority packets, *i.e.,* packets 5 and 4, are extracted from the queue to make room for packets with higher priorities. While this is highly effective in keeping high-priority packets on shorter paths, packet extraction from arbitrary locations in the queue is not implementable in programmable hardware. To resolve this challenge, the switch should predict which packets may be deflected in the future and preemptively deflect them. In the above example, preemptively deflecting every packet with a priority larger than 3 **(d)** will lead to a similar outcome as Selective Deflection.

High-level design. Our proposal makes deflection decisions before admitting the packet into the switch buffer. To this end, when a packet arrives, the Relative Priority Calculation module (§3.1) first calculates its priority relative to the packets inside the destination queue and feeds the relative priority alongside the destination queue occupancy to the Preemptive Deflection Algorithm (§3.2) to determine if the packet should be enqueued or deflected. Selectively deflecting packets based on the remaining bytes of their flows results in low latency under various degrees of load [6]. Accordingly, for our design, we assume that packets are marked with the remaining bytes of their flows. In this scheme, packets that belong to flows with smaller remaining bytes have a higher priority and therefore, are more likely to stay on shorter forwarding paths.[4] Augmenting packets with flow size information can be achieved by deploying a marking layer at the sender hosts [6].

3.1 Relative Priority Calculation

The relative priority of an incoming packet represents its precedence over the packets inside the destination queue. We develop two methods for estimating the relative priority to balance the accuracy of the estimation vs. processing resource consumption trade-off.

[4]Our design also works with other marking paradigms such as the Least Attained Service (LAS) scheme [14] in which packets are tagged with the number of bytes that have been sent from their flows. Appendix A shows that, with LAS, Preemptive Deflection achieves 73% lower 99^{th} percentile QCT than Simple Deflection under 95% load. Furthermore, our testbed experiments in §5 demonstrate that our technique also works with simpler schemes such as per-flow prioritization.

1) Quantile estimation. Measuring a packet's *quantile*, *i.e.,* the number of packets in the destination queue with higher priorities than the newly arrived packet divided by the total number of packets inside the queue, can accurately compute its relative priority [80]. However, quantile calculation is a resource-intensive operation [80] as it requires comparing the priority of the new packet to all the packets inside the destination queue. For instance, in PISA architecture, 8 processing stages are required to calculate the quantile relative to 20 packets. Inspired by [80], to limit the resources required for quantile-based relative priority calculation and avoid compromising implementability, we approximate a packet's quantile by comparing its priority to a portion of recently enqueued packets instead of all the packets inside the destination queue. Additionally, we develop a statistical method that reduces the memory footprint and the processing resources required for relative priority estimations, as we next explain.

2) Statistical distribution mapping. We develop a lightweight method that estimates the relative priority of the packet using a statistical model of the workload. Our objective here is to mathematically model the distribution of packet priorities and use that model to estimate the relative priority of an incoming packet. Using historical data of packet priorities in each queue and distribution fitting, the operator identifies the class of distribution (*e.g.,* normal, exponential, etc.) that best fits the priorities of enqueued packets for each port.[5] This distribution class is then loaded in each switch for each port. At the deflection time, the switch generates the distribution of currently enqueued packets in the output port using the distribution class and certain statistical properties of the enqueued packets (*e.g.,* the average priority of the packets). The relative priority of the incoming packet is then estimated using this distribution. We find experimentally that under Selective Deflection and with current datacenter workloads, the exponential distribution better fits the priority distribution compared to some alternatives such as uniform, normal, and Pareto distributions (Appendix B). As such, in the rest of the paper, we use the exponential distribution: when a packet arrives, the switch calculates the average remaining bytes (M) of the packets in the destination queue and uses the CDF of the exponential distribution, *i.e.,* $1 - e^{-\frac{x}{M}}$ (x being the remaining bytes of the flow corresponding to the incoming packet), to calculate the relative priority.

All that remains is choosing among the two methods which becomes a question of resource availability. As we show in §5, both methods outperform the existing techniques, and the quantile method is more resilient than the statistical counterpart under extreme load. The statistical approach, on the other hand, requires 4× fewer processing stages to be implemented in Tofino switches (§4), making it a better fit for networks with limited resources and a lower degree of transient congestion. In Appendix C, we further investigate the trade-off between the accuracy and resource consumption of these methods.

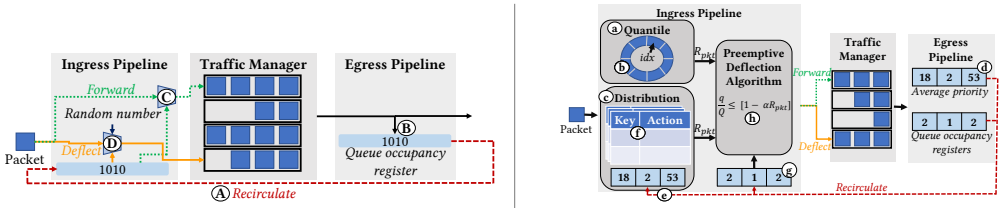
3.2 Preemptive Deflection Algorithm

Using the relative priority, calculated in the previous section, the Preemptive Deflection algorithm sets a threshold (τ) on the destination queue occupancy and deflects an incoming packet if the queue occupancy is more than the threshold. Equation 1 illustrates the relation used for calculating the τ based on the relative priority (R_{pkt}) and the destination queue capacity (Q):

$$\tau = Q \times [1 - \alpha \times R_{pkt}] \tag{1}$$

The user-defined parameter, α, indicates the aggressiveness of the Preemptive Deflection algorithm in deflecting packets. Larger α values result in a tighter threshold on queue occupancy and thus, a higher chance of preemptively deflecting packets. In §5, we evaluate the effect of various α values on the performance of Preemptive Deflection. After deciding to deflect a packet, the same algorithm is applied to the deflection port's queue occupancy and the relative priority of the

[5]Recording packet priorities and mapping various distributions to them can be done offline.



(a) Simple Deflection (b) Preemptive Deflection

Fig. 4. Implementing Simple and Preemptive Deflection in the PISA switch architecture.

deflected packet inside the deflection port's queue to decide if the packet should be dropped or forwarded to the deflection port.

Workflow. Figure 3 presents the workflow for preemptive packet deflection. When a packet arrives, we calculate its relative priority compared to the packets inside the destination queue (1) and feed the relative priority and the destination queue occupancy information into the Preemptive Deflection algorithm (2). Using this information, the algorithm decides whether the packet should be enqueued (3) or deflected (4). Deflecting low-priority packets leaves headroom for absorbing bursts of high-priority packets. If the destination queue is full, we deflect the packet irrespective of its priority. To deflect a packet, we determine its relative priority compared to the packets inside the deflection port's queue (5) and apply a Preemptive Drop algorithm (6) that uses the deflection port's queue occupancy and the packet's relative priority to decide if the packet should be enqueued (7) or dropped (8). Preemptively dropping the low-priority packets suppresses large flows, by signaling the congestion to the endpoints, and averts network over-utilization.

We discussed how we use Preemptive Deflection to address the practicality challenges of Selective Deflection. The challenges associated with implementing Simple and Preemptive Deflection in programmable hardware and our proposed solutions are described next.

4 HARDWARE IMPLEMENTATION

We implement Simple Deflection, quantile-based Preemptive Deflection, and distribution-based Preemptive Deflection in an Intel Tofino switch using 707, 1011, and 875 lines of P4 code, respectively. This section describes the challenges of implementing these techniques and how we address them.

4.1 Implementing Simple Deflection

To minimize packet loss, Simple Deflection reroutes the packets toward randomly selected switches to which there are ports with enough buffer capacity [82]. Filtering out congested ports before deflection improves the average latency of Simple Deflection by 3× under 55% load. Accordingly, implementing Simple Deflection requires distinguishing between congested and non-congested ports toward the neighboring switches and selecting a non-congested port uniformly at random.[6] Figure 4a provides an illustrative presentation of how we implement these operations.

Identifying ports with enough buffer capacity. To differentiate between congested and non-congested ports, we require queue occupancy information at the ingress pipeline which decides whether or not to deflect a packet. Alas, not all switch architectures provide queue occupancy information at the ingress pipeline [80]. To address this, we define two register arrays (one in the ingress pipeline and one in the egress pipeline) representing two bitmaps with sizes equal to the number of ports. The ones and zeros in the bitmaps represent congested and non-congested ports, respectively. We use *control* packets that recirculate inside the switch and transfer the queue

[6]Existing work on Simple Deflection chooses a random, non-congested port for deflecting packets [82]. While deflecting packets to the least congested port toward neighboring switches might provide better load distribution, implementing it in Tofino imposes further resource consumption constraints.

occupancy bitmap from the egress to the ingress pipeline (A). The bitmap at the egress pipeline is updated any time a data packet is forwarded to a port. When a data packet enters the egress pipeline, it updates the bit that corresponds to the destination port (B). Note that the *control* packets are not required for the architectures that provide queue occupancy information at the ingress pipeline, such as Tofino 2 [5, 80].

Choosing a random non-congested port, with sufficient buffer capacity, for packet deflection. At the ingress pipeline, we forward a packet if the destination port's corresponding bit is 0 (C). Otherwise, to deflect it (D), we generate a random number, r, between 0 and the number of neighboring switches and deflect the packet to the first non-congested port toward a switch whose corresponding bit is located after the r'th bit of the bitmap. In Appendix E, we prove that this technique *approximately results in uniform random selection* between non-congested ports toward the neighboring switches.

4.2 Implementing Preemptive Deflection

Realizing Preemptive Deflection (PD) requires implementing quantile-based and distribution-based relative priority calculation and the deflection algorithm. Figure 4b depicts the implementation of these methods in the PISA architecture.

Quantile estimation. The quantile-based PD (a) uses the quantile of the newly arrived packet as its relative priority. Considering the buffer capacity of commodity switches, it is not feasible to calculate a packet's quantile relative to all the packets in a queue. However, it is possible to estimate the quantile of a packet by comparing it to a portion of the packets inside the queue using a circular window and packet sampling [80]. To this end, as packets arrive, we store samples of their priorities, *i.e.,* the remaining bytes of their corresponding flows, and calculate a packet's priority relative to the ones stored in the circular window (b). To update the circular window, we keep an index (idx) pointing to the oldest element in the window and overwrite the element to which the index points, with the priority of the newly enqueued packet, moving the index forward by 1.

Exponential distribution mapping. To estimate the relative priority using an exponential distribution (c), when a packet arrives, we decide if the packet should be enqueued or deflected using the average priority of the packets inside the destination port's buffer. We then update the average priority corresponding to the forwarding or deflection port if the packet was enqueued or deflected, respectively, based on the priority of the incoming packet. Reading the average priority of the previously enqueued packets to make Preemptive Deflection decisions and storing it after it is updated requires accessing a memory block in two different stages, which is not supported by Tofino switches [17, 34, 81].

To overcome this challenge, we use two register arrays, one at the ingress and one at the egress pipeline (d), storing a moving average of the priority of the packets enqueued in every port's buffer. To update the register arrays, data packets read an element from the ingress pipeline's register array based on the port to which they are being forwarded, update the moving average using the priority assigned to them by end hosts, and carry the newly calculated average to the egress pipeline. We use pre-filled match-action tables to implement the moving average function on packet priorities. In particular, we look up the table using the current moving average value and the priority of the newly arrived packet and retrieve the updated average from the table. This is because existing PISA switches like Tofino do not offer primitives for multiplication, division, and floating point operations [67]. In the egress pipeline, we update the register array using the new average priority carried by the data packets. To keep the register arrays synchronized, we use *control* packets to recirculate the most recent average priority information from the egress to the ingress pipeline (e).

As the data packets enter the ingress pipeline, the average priority (M) is retrieved from the register array to calculate the relative priority, *i.e.,* $R_{pkt} = 1 - e^{-\frac{x}{M}}$ where x is the priority of the

incoming packet. Although the PISA architecture does not support all the operations required for calculating the CDF of an exponential distribution, these operations can be approximated using lookup match-action tables [67] (f). Accordingly, similar to our technique for updating the average priority, we use pre-filled match-action tables to calculate the relative priority of incoming packets. Using exponential distribution to estimate the relative priority requires two processing stages at the ingress pipeline which is 4× lower compared to quantile estimation.

Preemptive Deflection algorithm. In addition to the relative priority of the incoming packet, the Preemptive Deflection algorithm requires the queue occupancy information (g). Similar to Simple Deflection, we use *control* packets to synchronize the queue occupancy information between switch pipelines.[7] Note that both queue occupancy information and the average priority corresponding to a port can be transmitted using a single control packet. We deflect a packet if the destination queue occupancy is over a threshold (Equation 1) (h). The threshold values are pre-calculated and inserted into the match-action tables to avoid resource exhaustion.

Overall, out of the 12 processing stages available in a Tofino 1 switch, our implementations of Simple Deflection, quantile-based Preemptive Deflection, and distribution-based Preemptive Deflection require 8, 11, and 2 processing stages, respectively.

5 PERFORMANCE EVALUATION

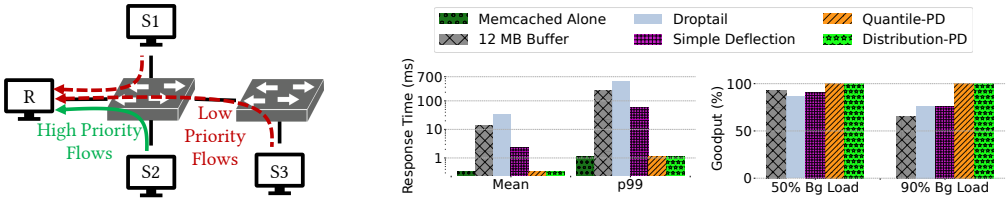
We evaluate our approximations of Simple and Selective Deflection on a testbed with two Intel Tofino switches and 4 servers and using Omnet++ network simulations.[8] In this section, we use the terms Preemptive Deflection and Simple Deflection, to refer to the quantile-based Preemptive Deflection and our approximation of Simple Deflection, respectively, and explicitly mention whenever we are referring to distribution-based Preemptive Deflection. Our results show that the implementable deflection schemes proposed in this paper closely approximate Simple and Selective Deflection and outperform the existing deployed systems. Our key findings are summarized below:

- In the physical testbed, Simple Deflection, quantile-based Preemptive Deflection, and distribution-based Preemptive Deflection result in 8×, 425×, and 437× lower 99^{th} percentile response times, respectively, compared to a drop-tail queue baseline.
- In large-scale simulations with different categories of congestion control, (a) a drop-base congestion control algorithm, TCP Reno, (b) an ECN-based congestion control, DCTCP [10], (c) a delay-based congestion control, Swift [54], and (d) a technique based on sub-RTT explicit notifications, Bolt [13], our algorithms closely approximate Selective and Simple Deflection techniques while being implementable in programmable switches.
- In all the experiments above, our algorithms outperform the techniques that can be implemented in datacenters today such as ECMP and AIFO [80], *e.g.,* in conjunction with Swift, our implementation of Simple Deflection improves the 99^{th} percentile query completion time (QCT) by 36% and 16× compared to ECMP and AIFO, respectively, under 55% load, and Preemptive Deflection improves the 99^{th} percentile QCT by 2.5× and 15× compared to ECMP and AIFO under 95% load.

5.1 Tetbed Evaluations

To evaluate our prototype implementations, we run a mixed workload on four server machines connected to two Intel Tofino switches that run Simple and Preemptive Deflection data planes. The server machines feature 40 Gbps Intel XL710 NICs, Intel Xeon E5-2620 v4 CPUs, and 64 GB of

[7] In §6, we demonstrate that the overhead of recirculating *control* packets is low even with a high number of active ports.
[8] The artifacts for the large-scale simulations and hardware implementation are publicly available at https://github.com/hopnets/practical_deflection.git.



(a) Testbed Topology (b) TCP Traffic (c) UDP Traffic

Fig. 5. Preemptive Deflection improves the performance of high-priority flows in our physical testbed.

memory. Figure 5a illustrates the topology for our testbed. In our experiments, two servers generate low-priority background flows and one generates high-priority traffic using a combination of *Iperf* and key-value workloads. We evaluate four different cases: 1) No deflection, 2) Simple Deflection, 3) quantile-based Preemptive Deflection (Quantile-PD), and 4) distribution-based Preemptive Deflection (Distribution-PD). In order to emulate packet drops at a small-scale granularity, we set the per-port switch buffer capacity to 80 KB. All results are averaged over 5 runs.

Preemptive Deflection minimizes the impact of low-priority flows on the response time of high-priority Memcached flows. As our first set of experiments on the physical testbed, we generate a background workload consisting of two Iperf instances creating an incast on the link connecting to one of the servers, totaling up to 35 Gbps. Next, we run parallel high-priority Memcached connections that compete with the background traffic by offering 500K Requests-per-second aggregate cache traffic load. We measure the response times for high-priority cache workload. We also evaluate the case of having Memcached traffic without background flows and the case of having a switch with 12 MB of buffer capacity. Figure 5b demonstrates that PD outperforms Simple Deflection and achieves a latency close to the case of having cache traffic without background flows. In particular, PD reduces the 99^{th} percentile response time by 425×, 51×, and 199× compared to the drop-tail queue, Simple Deflection, and the switch with 12 MB buffer capacity, respectively, while adding only 0.59% extra latency compared to only having Memcached traffic.

Preemptive Deflection favors high-priority UDP flows. In the next experiment, two of the servers in our testbed generate background load using large low-priority UDP flows and the third server sends bursts of high-priority UDP flows. We generate 5 bursts, each consisting of 10000 high-priority packets, in a 5-second period. Figure 5c illustrates the percentage of high-priority packets successfully received by the receiver (*goodput*) under 50% and 90% background loads. We observe that Simple Deflection performs similarly to the case of having a large buffer capacity and delivers 5% more high-priority packets than the baseline droptail queue under 50% load. However, as we increase the load to 90%, Simple Deflection fails to provide any improvement over the baseline. PD, on the other hand, outperforms other techniques and delivers 100% of the high-priority packets under both 50% and 90% load. In particular, both Quantile-PD and Distribution-PD deliver 23%, 24%, and 35% more packets compared to the droptail queue, Simple Deflection, and the switch with 12 MB buffer capacity, respectively.

5.2 Large-scale Simulation Setup

Topology. We simulate a 2-tier leaf-spine topology consisting of 4 spine switches, 8 leaf switches, and 40 machines connected to each leaf. The links connecting the spine switches to the leaf switches and the leaf switches to the servers have 40 Gbps and 10 Gbps bandwidth, respectively [6]. The switches have a 300 KB queue capacity per port [6, 15, 82]. We also run the experiments under two other topology settings: 1) an 8-ary fattree and 2) a leaf-spine topology with 100 Gbps bandwidth links [53, 72, 73, 79].

Workload. We evaluate Simple and Preemptive Deflection under a combination of background load and incast traffic patterns. For the background load, we set the flow sizes and inter-arrival times based on two publicly available datacenter traffic traces, Facebook's cache follower and Google's web search [10, 64], and scale the flow inter-arrival times to generate different degrees of background load. To simulate incast traffic pattern, a randomly selected client sends requests to multiple randomly chosen servers. This would cause the servers to simultaneously send replies to the client and create an incast event. We generate different degrees of burstiness by varying three factors: 1) The number of incast queries per second (QPS), 2) The scale of an incast event, *i.e.,* the number of flows sent per event, and 3) The size of each flow. Unless stated otherwise, we generate 50% background load and 100 flows of size 40 KB per incast query for our evaluations. We alter the incast event arrival rate between 2000 and 70000 QPS depending on the network bandwidth and the intended load [6, 55, 82].

Other approaches. In this section, we evaluate Simple Deflection, quantile-based Preemptive Deflection (Quantile-PD), and distribution-based Preemptive Deflection (Distribution-PD). We compare the performance of these techniques with ECMP, a commonly used load-balancing technique in datacenters, AIFO [80], an early drop scheme that drops packets based on their position in their corresponding flow to approximate Shortest Remaining Processing Time (SRPT) scheduling, and two state-of-the-art representatives of Simple and Selective Deflection: DIBS [82], a technique that deflects packets to randomly selected neighboring switches, and Vertigo [6], a Selective Deflection scheme that marks every packet with the remaining bytes of its corresponding flow and uses this information to deflect the packets of large flows with higher probability.

Addressing packet re-ordering. To recover from the redundant packet reordering resulting from Selective Deflection, Vertigo uses packet priorities, *i.e.,* remaining bytes of their corresponding flows, to restore the correct order of the packets at the destination host. Since Preemptive Deflection also marks packets with the remaining bytes of their flows, we use a similar ordering layer to absorb the reordering. For Simple Deflection, similar to DIBS, we turn off the fast retransmission in TCP to avoid throughput reduction caused by reordering.

Evaluation metrics. For our evaluations, we measure the mean and 99^{th} percentile Flow Completion Times (FCT) and Query Completion Times (QCT), *i.e.,* the time taken from initiating the incast event to receiving all the responses. Additionally, we collect and report the application-level throughput (goodput), the average Round-Trip Time (RTT), the number of packet drops, and the percentage of reordering, *i.e.,* the number of out-of-order packets divided by the total number of packets.

Parameter settings. Preemptive Deflection uses three parameters: 1) α: the deflection aggressiveness, 2) window size: the number of previously enqueued packets considered for quantile-based relative priority calculation, and 3) sampling rate: the rate at which we store packet priorities for quantile calculation. Considering the hardware resource availability and our findings in §5.4, we set the default values of α, window size, and sampling rate to 0.1, 20 packets, and one sample per 20 packets, respectively. Unless stated otherwise, we set TCP's initial window size to 10 packets [6, 82], DCTCP's threshold to 65 packets [10], and the minimum and initial RTO to $10ms$ [6, 82]. We tune the parameters of Swift and Bolt according to [54] and [13], respectively. The other parameters of our simulations are set to their default value in the INET framework [1].

5.3 Large-scale simulation results

We evaluate Simple and Preemptive Deflection in conjunction with distinct congestion control protocols and under different combinations of background load and incast traffic patterns.

Preemptive Deflection closely follows the performance of Selective Deflection with delay-based and sub-RTT congestion control protocols. In the first experiment, we deploy

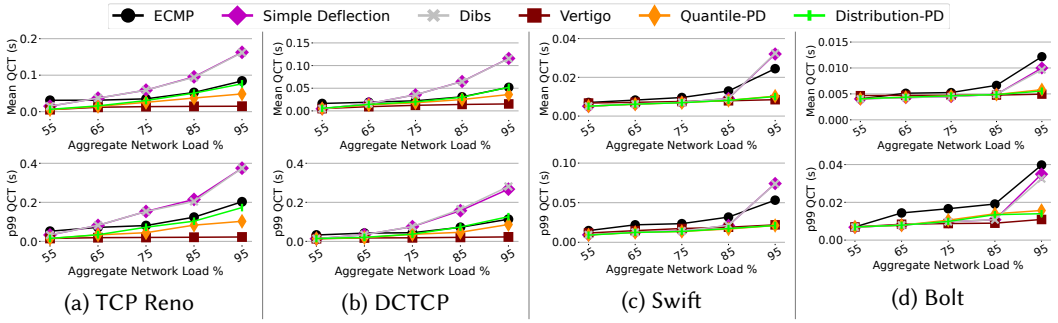
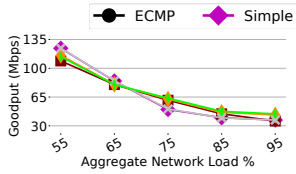


Fig. 6. Preemptive Deflection outperforms ECMP and Simple Deflection under load and achieves comparable latency to Vertigo with transport protocols that provide a fast reaction to congestion.

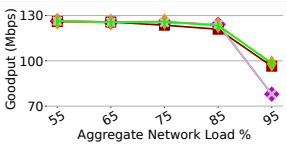
ECMP, DIBS, Vertigo, Simple Deflection, and Preemptive Deflection (PD) on top of TCP Reno and DCTCP [10], representatives of window-based transport protocols, Swift [54], a representative of delay-based protocols, and Bolt [13], a state-of-the-art protocol that supports congestion notification at sub-RTT granularity. We gradually increase the load by elevating the incast event arrival rate. Figure 6 illustrates that Simple Deflection successfully defuses bursts when the degree of the load is low. In particular, under 55% load, both Simple Deflection and DIBS achieve 51% and 71% lower mean QCT than ECMP under TCP and DCTCP, respectively. However, they break quickly under load. For instance, with DCTCP, Simple Deflection increases the average FCT and QCT by 57% and 2×, respectively, compared to ECMP under 95% load. The performance degrades at a much slower rate when Simple Deflection is used with Swift and Bolt due to their faster reaction to network congestion compared to TCP and DCTCP.

Unlike Simple Deflection, Selective Deflection preserves the low latency under load, irrespective of congestion control protocol, by deflecting the packets with the highest number of remaining bytes in their corresponding flows. However, the performance superiority of Selective Deflection comes at the cost of compromising its implementability. Figures 6a and 6b demonstrate that, while working with TCP or DCTCP, Preemptive Deflection achieves the middle ground between Selective and Simple Deflection. In particular, in conjunction with DCTCP, PD achieves 31% and 69% lower mean QCT and 24% and 69% lower 99^{th} percentile QCT than ECMP and Simple Deflection, respectively, under 95% load. While being effective under load, Preemptive Deflection is still prone to packet loss under extreme congestion due to its early deflection and drop paradigm and might occasionally deflect a packet that could have been enqueued considering the subsequent packets, resulting in higher latency than Selective Deflection. Concretely, under 95% load, PD+DCTCP incurs an extra 5% packet loss and 2.35× higher average query completion time than Vertigo+DCTCP. Accordingly, deploying Preemptive Deflection alongside congestion control schemes with fast reactions to congestion, such as Swift, significantly improves its effectiveness by reducing the number of packet drops. For instance, under 95% load, PD+Swift reduces the number of packet drops by 52× compared to PD+DCTCP which results in a 3.5× improvement in PD's mean QCT. We observe similar results for PD+Bolt. Particularly, PD+Bolt reduces the number of packet drops and the average QCT by 153× and 6×, respectively, compared to PD+DCTCP under 95% load. As a result of the reduction in the number of packet drops, Preemptive Deflection achieves a latency as low as Vertigo in conjunction with Swift and Bolt. Additionally, under 95% load, PD+Swift and PD+Bolt achieve 52% and 35% lower 99^{th} percentile FCT than ECMP+Swift and ECMP+Bolt, respectively, showing that packet deflection benefits Swift and Bolt by absorbing local bursts.

Distribution-PD achieves comparable performance to Quantile-PD under RTT and sub-RTT-based congestion control protocols. While implementing the distribution-based Preemptive Deflection



(a) DCTCP



(b) Swift

Fig. 7. Preemptive Deflection improves the goodput of elephant flows compared to Selective Deflection.

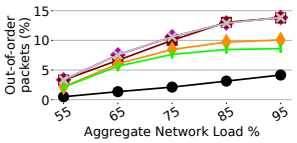


Fig. 8. Preemptive Deflection imposes lower degrees of reordering compared to Selective Deflection.

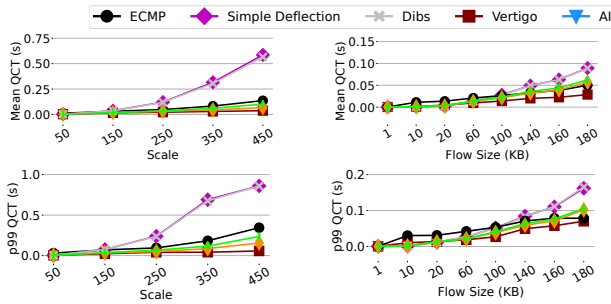
requires fewer resources than the quantile-based Preemptive Deflection (§4), the quantile-based paradigm has higher accuracy than the distribution-based scheme in calculating a packet's relative priority (Appendix C). Accordingly, Distribution-PD experiences higher query completion times than Quantile-PD under load. However, Figures 6c and 6d show that while deployed with Swift and Bolt, Distribution-PD is as effective as Quantile-PD under load. This is because Swift and Bolt react faster to congestion and cause fewer packet drops compared to TCP or DCTCP.

Simple Deflection performs similar to DIBS. Figure 6 also demonstrates that Simple Deflection effectively defuses bursts under light loads and closely follows the performance pattern of DIBS. In particular, with DCTCP and under 55% load, both Simple Deflection and DIBS outperform ECMP and Vertigo by achieving 3.5× and 5.25% lower mean QCT and 2.4× and 3.54% lower 99^{th} percentile QCT, respectively.

Preemptive Deflection achieves higher goodput than Vertigo. By constantly extracting the packets of large flows from queues and deflecting them, Selective Deflection throttles their send rates, leading to diminished goodput, *i.e.,* application-level throughput. Conversely, Preemptive Deflection adopts an alternative approach by refraining from packet extraction once they are enqueued. This strategy sporadically benefits large flows, thus, enhancing the goodput compared to Selective Deflection. To show this, we compare the goodput of elephant flows, *i.e.,* flows larger than 10 MB [6], under various deflection-based paradigms with DCTCP and Swift as the congestion control protocol. Figure 7 shows that PD consistently achieves superior goodput for elephant flows in comparison to Vertigo. For instance, with DCTCP under 95% load, PD improves the goodput of elephant flows by 20% compared to Vertigo. Additionally, we observe that Simple Deflection's goodput closely parallels that of DIBS.

Preemptive Deflection imposes lower degrees of reordering compared to Selective Deflection. We also compare the degree of reordering imposed by different techniques. To this end, we measure the reordering percentage, *i.e.,* number of out-of-order packets received by receiver hosts divided by the total number of packets, under various degrees of load. Figure 8 presents the results. We observe that applying deflection increases the degree of reordering. However, the adverse effects of this heightened reordering on performance can be mitigated by deploying an ordering component, responsible for restoring the correct packet order, at the receiver hosts [6, 35, 37, 40]. For instance, with DCTCP under 95% load, Vertigo and PD manage to achieve 70% and 31% lower average QCT than ECMP despite causing 3.3× and 2.4× more reordering, respectively. Since Preemptive Deflection uses FIFO queues instead of Shortest Remaining Processing Time (SRPT) scheduled queues, it imposes lower degrees of reordering compared to Selective Deflection.

Preemptive Deflection is resilient to the scale and the flow size of incast events. Next, we gradually increase the load from 50% to 95% by increasing the scale of incast events, *i.e.,* the number of requests sent per incast query. To this end, we set the incast event arrival rate to 4000 QPS and the size of each flow in the event to 40 KB and change the scale from 50 to 450 requests per query. We use DCTCP as the congestion control protocol. Figure 9a illustrates that while Simple Deflection breaks as the scale goes over 150 requests per query, Preemptive Deflection remains



(a) Different incast scales (b) Different incast flow lengths

Fig. 9. Preemptive Deflection remains resilient to incast scale and works best when incast flow sizes are shorter than 100 KB.

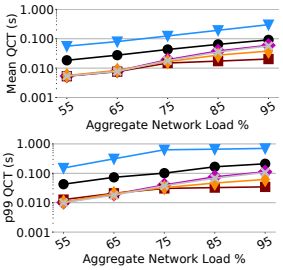


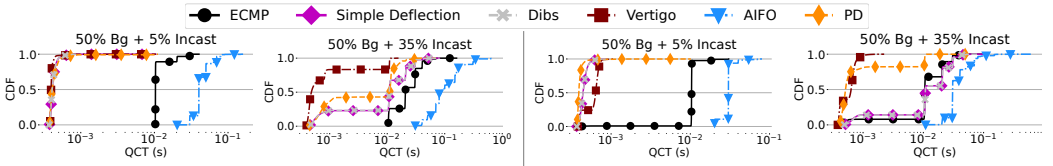
Fig. 10. PD achieves lower QCTs than ECMP, Simple Deflection, and AIFO in a fat-tree topology.

resilient and performs closely to Selective Deflection (*e.g.,* PD outperforms ECMP and DIBS by achieving 2× and 4× lower 99^{th} percentile FCT, respectively, under 450 requests per incast query). Due to its higher accuracy for calculating relative priority, Quantile-PD finishes the incast queries 34% faster than Distribution-PD when the scale of incast queries is 450.

To evaluate the resiliency of distinct techniques against the incast flow size, we generate 4000 queries per second, set the scale to 100 flows per query, and change the size of each flow from 1 KB to 180 KB. Figure 9b presents the results. When the size of incast flows is under 100 KB, Preemptive Deflection outperforms ECMP and Simple Deflection. In particular, with 100 KB incast flows, PD results in 28% and 24% lower 99^{th} percentile QCT than ECMP and DIBS, respectively. As the size of incast flows goes over 100 KB, PD results in higher latency than ECMP. This is due to prioritizing packets of background flows over incast flows as more than 50% of the background flows in our experiments are smaller than 100 KB [64]. We believe the higher latency of Preemptive Deflection at very large flows would not be an issue since previous studies of data center traffic argue that around 70% of the flows in large datacenters are smaller than 100 KB [11, 13, 64].

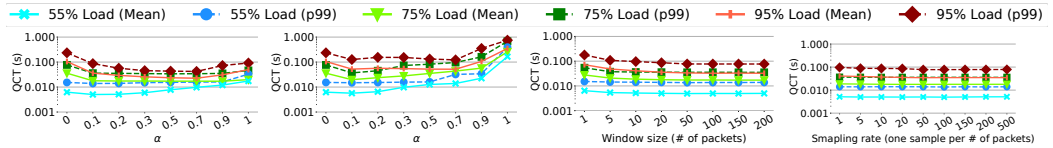
Deflection-based techniques defuse bursts more effectively in a three-tiered topology. We further evaluate different deflection-based paradigms by simulating an 8-ary fat-tree topology [7] with 32 edge switches, 32 aggregate switches, 16 core switches, and 128 servers. The bandwidth of all the connection links is set to 10 Gbps [6, 82]. For this set of experiments, we simulate ECMP, AIFO, DIBS, Vertigo, Simple Deflection, and PD on top of DCTCP. Figure 10 illustrates that AIFO's early drop mechanism is inefficient when faced with bursts of high-priority packets. Particularly, AIFO completes queries 3×, 10×, and 10× slower than ECMP, Simple Deflection, and PD under 55% load. We also observe that, unlike our experiments with 2-tier leaf-spine topology, ECMP does not outperform Simple Deflection under load. In particular, DIBS and Simple Deflection complete 4% more queries and achieve 2× lower 99^{th} percentile QCT than ECMP under 95% load. This is due to the extra 33% buffer capacity and 20% more choices available for packet deflection in fat-tree topology compared to a 2-tier leaf-spine topology with 4 spines and 8 leaf switches. However, Simple Deflection shows a more drastic degradation of performance than ECMP as the load increases. Using the extra resources, while also taking the remaining bytes of the flows into consideration, Preemptive Deflection outperforms Simple Deflection and achieves 33% and 42% lower mean and 99^{th} percentile QCTs under 95% load, respectively.

Preemptive Deflection effectively defuses bursts in networks with higher bandwidth. Next, we simulate a 2-tier leaf-spine topology with 100 Gbps links [53, 72]. We change the arrival rate of incast queries to generate different degrees of burstiness. Figure 11 illustrates the performance of different techniques in conjunction with DCTCP and Swift under low and high degrees of burstiness. We observe that similar to the network with lower link capacities, Simple Deflection effectively



(a) DCTCP (b) Swift

Fig. 11. As we increase the link rate to 100 Gbps, Simple and Preemptive Deflection effectively defuse bursts under low and high degrees of burstiness, respectively.



(a) α for Quantile-PD (b) α for Distribution-PD (c) Window size (d) Sampling rate

Fig. 12. Measuring the performance of Preemptive Deflection under various parameter settings.

defuses low degrees of bursts and performs similarly to DIBS. Also, Preemptive Deflection effectively absorbs bursts and outperforms ECMP and Simple Deflection under load. Particularly, with Swift, PD achieves 6×, 18×, and 7× lower mean QCT than ECMP, AIFO, and Simple Deflection under 85% load, respectively. By providing a faster reaction to congestion, Swift improves the Preemptive Deflection's effectiveness in absorbing bursts under load. In Appendix F, we evaluate Simple and Preemptive Deflection under other combinations of background load and incast traffic patterns with both 10/40 Gbps and 100 Gbps links. Our findings are in alignment with our earlier observations.

5.4 Calibrating the parameters for Preemptive Deflection

Adjusting the aggressiveness of Preemptive Deflection. The Preemptive Deflection algorithm sets a dynamic threshold on the destination queue occupancy using the destination queue capacity, the relative priority of the incoming packet, and a user-defined parameter, *i.e.,* α, that indicates the aggressiveness in deflecting and dropping packets. Larger α values indicate a more aggressive packet deflection and drop paradigm. For Preemptive Deflection, there is a trade-off between choosing small and large values for α. Aggressively deflecting and dropping packets increases the chance of dropping the first few packets of small flows, *i.e.,* packets with the largest number of remaining bytes in a small flow, under the incast traffic pattern while small values of α might result in late reaction to congestion and reduce the effectiveness of Preemptive Deflection. To investigate the impact of α on PD's performance, we simulate quantile-based and distribution-based Preemptive Deflection with various α values in a two-tier leaf-spine topology under 55%, 75%, and 95% load. Figures 12a and 12b illustrate that, under 55% and 75% load, using small non-zero α values ($\sim$ 0.1) is preferable. For instance, under 55% load, setting α to 0.1 improves the average QCT of Quantile-PD by 19% and 71% compared to setting it to 0 and 1, respectively. This is due to the fact that setting α to zero eliminates relative priority calculation in the Preemptive Deflection algorithm and thus hampers the performance while setting it to a large value increases its sensitivity, thus increasing the chance of deflecting high-priority packets that contribute to transient bursts. Under extreme loads, on the other hand, Preemptive Deflection should be more sensitive to congestion, therefore, we need higher values of α ($\sim$ 0.5). In particular, under 95% load, setting α to 0.5 results in 50% and 40% better 99^{th} percentile QCT for Quantile-PD than setting it to 0.1 and 0.9, respectively. Datacenter operators can adjust α values based on the average load level in the network clusters.

Tuning the window size and sampling rate. To calculate a packet's relative priority, quantile-based Preemptive Deflection compares the priority of the newly arrived packet to previously enqueued packets. Due to resource limitations in Tofino, when implementing Quantile-PD, we

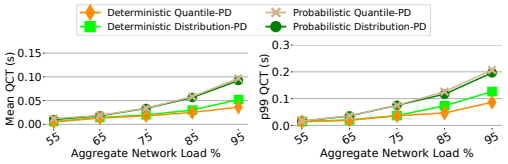


Fig. 13. Using a deterministic algorithm for PD outperforms the probabilistic approach under load.

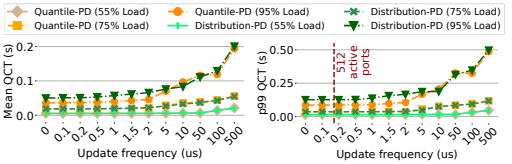


Fig. 14. PD provides low latency as long as the gap between queue occupancy updates is less than $2\mu s$.

calculate the quantile compared to only a subset of recently enqueued packets stored in a circular window. In Figure 12c, we report the impact of the window size on Quantile-PD's performance. Our findings show that we need larger subsets to achieve better latency as the load increases. In particular, while the QCTs converge when the window size goes over 5 packets under 55% load, we need more than 50 packets for the latency to converge under 95% load. Unfortunately, calculating the quantile compared to 50 packets is not feasible in programmable hardware such as Tofino 1. However, by setting the window size to 20 packets, we achieve the middle ground between implementability and performance under various loads. In particular, with a window size of 20 packets, Quantile-PD remains implementable in programmable hardware, requiring 11 processing stages, and achieves comparable QCT to setting the window size to 200 packets under 55% and 75% load while increasing the 99^{th} percentile QCT by only 13% under 95% load.

In addition to limiting the window size, we also use packet sampling to reduce Quantile-PD's implementation complexity (§4). In particular, as the packets pass the switch, we store samples of their priorities for quantile calculation. Figure 12d shows that, unlike the cases with α and window size, changing the sampling rate has a marginal effect on the QCTs under Quantile-PD.

6 DISCUSSION

Alternative Preemptive Deflection algorithms. Early congestion detection paradigms deployed at the core of the network can be probabilistic [29, 42, 52, 70] or deterministic [10, 33, 80]. Probabilistic techniques are more sensitive to parameters but little changes in the queue occupancy slightly change the probability of signaling congestion. Deterministic techniques, on the other hand, require less parameter tuning [10] but might be vulnerable to slight fluctuations as they set a hard threshold on the queue occupancy. In addition to our original design, we also evaluate Preemptive Deflection (PD) with a probabilistic algorithm inspired by Random Early Detection [29]. With the probabilistic Preemptive Deflection algorithm, when a packet arrives, we enqueue it if the queue occupancy is less than or equal to a user-defined threshold, min_{th}. Otherwise, we probabilistically deflect the packet. The deflection probability is calculated as $R_{pkt}[\frac{q-min_{th}}{Q-min_{th}}]$ where q, Q, and R_{pkt} are the destination queue occupancy, the destination queue capacity, and the relative priority of the newly-arrived packet, respectively. We set the min_{th} parameter according to [10]. Figure 13 compares PD's original design with its probabilistic counterpart. Compared to the deterministic scheme, while performing similarly under low degrees of load, the probabilistic paradigm results in a 2.7× and 2.3× jump in PD's mean and 99^{th} percentile QCT under 95% load, respectively.

The recirculation overhead. While implementing Simple and Preemptive Deflection, we use *control* packets to transfer information, such as buffer occupancy, from the egress to the ingress pipeline. Unlike data packets, that pass the switch pipeline once, *control* packets continuously recirculate inside the switch. Since *control* packets solely go through the recirculation ports, they do not contribute to the queueing at other ports and leave a negligible throughput footprint. Our testbed measurements using four server machines, generating 10 million packets per second (Mpps) on aggregate, show no throughput footprint when we enable *control* packet recirculation. Additionally, in our testbed, we observe that, with up to 64 active ports, every 72-byte *control* packet

undergoes ∼ 1.7 million recirculations per second (Appendix D), resulting in a bandwidth overhead of less than 1 Gbps. Considering that Intel Tofino 1 switches offer a robust aggregate throughput of up to 6.4 Tbps [4], the throughput overhead of recirculating 64 *control* packets remains below 1%. It is worth noting that as the number of active ports exceeds 64, the average recirculation frequency of *control* packets decreases due to queueing at the recirculation port.

The queue occupancy information update frequency. In our implementations, the queue occupancy information in the ingress pipeline is updated periodically using *control* packets. The gap between updates impacts the effectiveness of our proposed algorithms in making deflection decisions. To investigate this, we evaluate the performance of Quantile-PD and Distribution-PD with various frequencies for updating the queue occupancy information. In the experiments with Distribution-PD, we use the same frequencies for updating the average priority information. For this set of experiments, we simulate the 2-tier leaf-spine datacenter, used in §5, and DCTCP as the congestion control protocol. Figure 14 illustrates that the QCTs of Preemptive Deflection stay steady as long as the time gap between updates is smaller than $2\mu s$ which is, in fact, larger than the gap observed in our testbed experiments under various numbers of *control* packets. Particularly, in Appendix D, we illustrate that under 120 Gbps load with 512 control packets recirculating in our testbed switch (512 active ports), the time gap between queue occupancy updates, *i.e.,* average time taken by each control packet to go through the switch pipeline and be recirculated, is around $1.7\mu s$.

Future directions. Our work builds on a few assumptions. First, similar to other proposals that exploit packet prioritization [6, 8, 58, 71, 74, 80], this paper focuses on private datacenters where security is enforced at the periphery of the network (e.g., via gateways and firewalls connecting the datacenter to the Internet) and the hosts inside the networks are assumed to be trusted. However, in untrusted networks that deploy packet prioritization, an adversary can starve other flows by sending high-priority packets. Exploring the deployability of priority-based deflection techniques in untrusted networks is a promising direction for future work.

Moreover, while we focus on output-queued switches with static queue size configurations, shared memory switches are also popular in datacenters because of their ability to absorb bursts using dynamic queue size thresholds [26, 61, 63]. Under this architecture, we can preemptively deflect packets by setting the queue capacity in Equation 1 to a constant value. However, this is not optimal for shared memory architectures due to imposing false positives and false negatives when applying Preemptive Deflection. Lastly, we rely on extensive experimental results to evaluate our approximation paradigms similar to the techniques that they are approximating, *i.e.,* DIBS [82] and Vertigo [6]. We leave tailoring Preemptive Deflection for shared memory switches and deriving performance guarantees for packet deflection through mathematical modeling for future work.

7 RELATED WORK

Managing traffic bursts. Many proposals attempt to mitigate burstiness. Traffic shaping, which regulates the flow of traffic into the network by smoothing out the bursts, helps prevent congestion. To this end, several congestion control techniques [23, 54, 55] and end-host packet schedulers [65] employ pacing during congestion periods. While effective in reducing the bursty transmissions that can overwhelm intermediate buffers, they fail to prevent the synchronized arrival of packets that lead to incast traffic patterns and cause packet loss in the last hop. Load balancing, distributing traffic across multiple paths to avoid congestion on a single path, is another area that has been studied with network traffic burstiness in mind [9, 37, 69]. However, these techniques also struggle to address the last-hop congestion. Finally, sub-RTT transport protocols [13] employ precise congestion signals to react to congestion events faster, therefore reducing the queue buildups that can lead to packet loss. We show that reactive techniques, such as packet deflection, can be practically combined with these transports and greatly enhance performance by locally absorbing bursts.

Exploiting packet prioritization. Aiming for low latency, several works [11, 14, 58, 74] prioritize packets of latency-sensitive flows by deploying packet scheduling. Particularly, pFabric [11] attempts to achieve low flow completion times by designing a flow control mechanism at the end-hosts and applying the SRPT scheduling at the core of the network. Following the same motivation, PIAS [14] attempts to improve latency while avoiding the need for flow size information. To this end, PIAS suggests LAS scheduling in which the priority of a flow decreases as it sends more packets into the network. Homa [58] advocates multi-level prioritization by applying SRPT scheduling between flows in sender hosts and between packets at the core of the network. In addition to imposing additional implementation complexities [8, 71, 80], such proposals fail to prevent packet loss under bursts. To address this, priority-aware deflection paradigms, such as Selective and Preemptive Deflection, offer better resilience against bursts of high-priority packets.

Deflection in networks. Deflection routing has been commonly used to reduce hardware implementation costs by leveraging bufferless network-on-chip designs [27, 28]. In larger-scale networks, deflection can be used to ensure that packets that arrive at congested [43] or failed links [32, 51, 77] do not experience drops, by directing the traffic away from the affected paths. While effectively used as a failover technique to recover from link failures, such techniques take more than the lifetime of datacenter bursts [84] to apply (e.g., around 50ms in MPLS fast re-route [78]). More recently, packet-level re-routing has been proposed to recover from switch buffer congestion in datacenter networks [6, 70, 82]. DIBS [82] deflects packets that arrive at a full destination port buffer to random neighboring switches. To prevent congestion collapse under high degrees of burstiness, Vertigo [6] introduces the notion of Selective Deflection by prioritizing packets of short flows and deflecting the packets of large flows with higher probability. Unfortunately, the above solutions face implementability challenges in resource-limited switches.

Programmble networks. Implementing packet deflection is a challenge due to the limited resources available in the network core. The emergence of programmable fabric enables stateful analysis of network congestion and opens the way for implementing more intelligent forwarding decisions. For example, programmable networks enable access to instantaneous buffer utilization information [80], in-network telemetry [18, 24], realizing or approximating various packet scheduling paradigms [8, 71, 74, 80], and burstiness measurements [46, 66]. The initial proposals on Simple Deflection [6, 82] rely on prototype FPGA implementations or software switches. On the other hand, in this paper, we propose an approximation of Simple and Selective Deflection on PISA pipelines that offer more flexible programming interfaces.

8 CONCLUSION

In this paper, we highlight and address the practicality challenges of packet deflection in datacenters. In particular, we propose a random port selection algorithm and the concept of Preemptive Deflection (PD) to approximate Simple and Selective Deflection, respectively. We implement and evaluate Simple and Preemptive Deflection using large-scale simulations and in a testbed consisting of Tofino switches. Our results illustrate that our Simple Deflection approximation effectively defuses bursts under light loads and performs closely to DIBS. Additionally, we observe that PD is resilient against load and reduces the 99^{th} percentile latency by 60% and 71% compared to ECMP and Simple Deflection, respectively, while achieving comparable performance to Selective Deflection.

9 ACKNOWLEDGMENTS

We would like to thank our shepherd and the anonymous CoNEXT reviewers for their insightful feedback. We would also like to thank Xin Jin for his equipment support and Sina Sharifi for his input and feedback on our paper. This project was partially supported by an Intel Fast Forward award, a Facebook faculty research award, and NSF NeTS grant 2313164.

REFERENCES

[1] 2020. *INET Framework. https://inet.omnetpp.org/*.
[2] 2020. *OMNeT++ Simulator. https://omnetpp.org/*.
[3] 2020. *Open Tofino. https://github.com/barefootnetworks/Open-Tofino*.
[4] 2023. *Intel Tofino 1 products. https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino/products.html*.
[5] 2023. *Tofino 2. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html*.
[6] Sepehr Abdous, Erfan Sharafzadeh, and Soudeh Ghorbani. 2021. Burst-Tolerant Datacenter Networks with Vertigo. In *CoNEXT*.
[7] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*.
[8] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. 2020. SP-PIFO: Approximating Push-in First-out Behaviors Using Strict-priority Queues. In *NSDI*.
[9] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. 2014. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *SIGCOMM*.
[10] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *SIGCOMM*.
[11] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pFabric: Minimal Near-optimal Datacenter Transport. In *SIGCOMM*.
[12] Mark Allman and Ethan Blanton. 2005. Notes on Burst Mitigation for Transport Protocols. *SIGCOMM CCR* (2005).
[13] Serhat Arslan, Yuliang Li, Gautam Kumar, and Nandita Dukkipati. 2023. Bolt: Sub-RTT Congestion Control for Ultra-Low Latency. In *NSDI*.
[14] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. 2015. Information-agnostic flow scheduling for commodity data centers. In *NSDI*.
[15] Wei Bai, Li Chen, Kai Chen, and Haitao Wu. 2016. Enabling ECN in Multi-service Multi-queue Data Centers. In *NSDI*.
[16] Sandro Bassi, Maurizio Decina, Paolo Giacomazzi, and Achille Pattavina. 1994. Multistage shuffle networks with shortest path and deflection routing for high performance ATM switching: The open-loop Shuffleout. *IEEE Transactions on Communications* (1994).
[17] R. Ben-Basat, X. Chen, G. Einziger, and O. Rottenstreich. 2018. Efficient Measurement on Programmable Switches Using Probabilistic Recirculation. In *ICNP*.
[18] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. 2020. PINT: Probabilistic In-band Network Telemetry. In *SIGCOMM*.
[19] Theophilus Benson, Aditya Akella, and David A Maltz. 2010. Network Traffic Characteristics of Data Centers in the Wild. In *IMC*.
[20] Alberto Bononi, Fabrizio Forghieri, and Paul R Prucnal. 1993. Analysis of One-buffer Deflection Routing in Ultra-fast Optical Mesh Networks. In *INFOCOM*.
[21] Flaminio Borgonovo, Luigi Fratta, and Joseph Bannister. 1993. Unslotted Deflection Routing in All-optical Networks. In *GLOBECOM*.
[22] Flaminio Borgonovo, Luigi Fratta, and Joseph A Bannister. 1994. On the Design of Optical Deflection-routing Networks. In *INFOCOM*.
[23] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-based Congestion Control. *ACM Queue* (2016).
[24] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. 2020. BeauCoup: Answering Many Network Traffic Queries, One Memory Update at a Time. In *SIGCOMM*.
[25] Yang Chen, Hongyi Wu, Dahai Xu, and Chunming Qiao. 2003. Performance Analysis of Optical Burst Switched Node With Deflection Routing. In *ICC*.
[26] A.K. Choudhury and E.L. Hahne. 1998. Dynamic queue length thresholds for shared-memory packet switches. *IEEE/ACM Transactions on Networking* (1998).
[27] Chris Fallin, Chris Craik, and Onur Mutlu. 2011. CHIPPER: A Low-complexity Bufferless Deflection Router. In *HPCA*.
[28] Chris Fallin, Greg Nazario, Xiangyao Yu, Kevin Chang, Rachata Ausavarungnirun, and Onur Mutlu. 2012. MinBD: Minimally-Buffered Deflection Routing for Energy-Efficient Interconnect. In *NOCS*.
[29] Sally Floyd and Van Jacobson. 1993. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM ToN* (1993).
[30] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *OSDI*.

[31] D. Gaiti and N. Boukhatem. 1996. Cooperative congestion control schemes in ATM networks. *IEEE Communications Magazine* (1996).
[32] Igor Ganichev, Bin Dai, P Brighten Godfrey, and Scott Shenker. 2010. YAMR: Yet Another Multipath Routing Protocol. *SIGCOMM CCR* (2010).
[33] Yixiao Gao, Yuchen Yang, Tian Chen, Jiaqi Zheng, Bing Mao, and Guihai Chen. 2018. DCQCN: Taming Large-Scale Incast Congestion in RDMA over Ethernet Networks. In *IEEE ICNP*.
[34] Nadeen Gebara, Alberto Lerner, Mingran Yang, Minlan Yu, Paolo Costa, and Manya Ghobadi. 2020. Challenging the Stateless Quo of Programmable Switches. In *HotNets*.
[35] Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, and Mohammad Alizadeh. 2016. Juggler: a practical reordering resilient network stack for datacenters. In *EuroSys*.
[36] 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 *IMC*.
[37] Soudeh Ghorbani, Zibin Yang, P Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. 2017. DRILL: Micro Load Balancing for Low-latency Data Center Networks. In *SIGCOMM*.
[38] Prateesh Goyal, Preey Shah, Kevin Zhao, Georgios Nikolaidis, Mohammad Alizadeh, and Thomas Anderson. 2022. Backpressure Flow Control. In *NSDI*.
[39] Soroush Haeri and Ljiljana Trajković. 2014. Intelligent Deflection Routing in Buffer-less Networks. *IEEE Transactions on Cybernetics* (2014).
[40] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. 2015. Presto: Edge-based Load Balancing for Fast Datacenter Networks. In *SIGCOMM*.
[41] Ching-Fang Hsu, Te-Lung Liu, and Nen-Fu Huang. 2002. Performance Analysis of Deflection Routing in Optical Burst-switched Networks. In *INFOCOM*.
[42] Alshimaa H Ismail, Ayman El-Sayed, Zeiad Elsaghir, and Ibrahim Z Morsi. 2014. Enhanced Random Early Detection (ENRED). *IJCA* (2014).
[43] Sundar Iyer, Supratik Bhattacharyya, N Taft, and C Diot. 2003. An Approach to Alleviate Link Overload as Observed on an IP Backbone. In *INFOCOM*.
[44] Hao Jiang and Constantinos Dovrolis. 2003. Source-Level IP Packet Bursts: Causes and Effects. In *IMC*.
[45] Hao Jiang and Constantinos Dovrolis. 2005. Why is the Internet Traffic Bursty in Short Time Scales? *SIGMETRICS Perform. Eval. Rev.* (2005).
[46] Raj Joshi, Ting Qu, Mun Choon Chan, Ben Leong, and Boon Thau Loo. 2018. BurstRadar: Practical Real-Time Microburst Monitoring for Datacenter Networks. In *APSys*.
[47] Rajgopal Kannan and Sibabrata Ray. 2000. A fair and efficient multicast ATM switch based on deflection routing. In *Proceedings ISCC 2000. Fifth IEEE Symposium on Computers and Communications*.
[48] Rishi Kapoor, Alex C Snoeren, Geoffrey M Voelker, and George Porter. 2013. Bullet Trains: a Study of NIC Burst Behavior at Microsecond Timescales. In *CoNEXT*.
[49] Kazuki Kawanabe and Tatsuro Takahashi. 2007. Effective Deflection Control Method in Optical Packet Switching Networks With Shared Buffers. *Electronics and Communications in Japan (Part I: Communications)* (2007).
[50] T. Khan, S. Rashidi, S. Sridharan, P. Shurpali, A. Akella, and T. Krishna. 2022. Impact of RoCE Congestion Control Policies on Distributed Training of DNNs. In *HOTI*.
[51] S Kini, S Ramasubramanian, A Kvalbein, and A F Hansen. 2009. Fast Recovery from Dual Link Failures in IP Networks. In *INFOCOM*.
[52] Jahon Koo, Byunghun Song, Kwangsue Chung, Hyukjoon Lee, and Hyunkook Kahng. 2001. MRED: a New Approach to Random Early Detection. In *ICOIN*.
[53] Ashok V. Krishnamoorthy, Hiren D. Thacker, Ola Torudbakken, Shimon Müller, Arvind Srinivasan, Patrick J. Decker, Hans Opheim, John E. Cunningham, Ivan Shubin, Xuezhe Zheng, Marcelino Dignum, Kannan Raj, Eivind Rongved, and Raju Penumatcha. 2017. From Chip to Cloud: Optical Interconnects in Engineered Systems. *Journal of Lightwave Technology* (2017).
[54] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M G Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. 2020. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *SIGCOMM*.
[55] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. 2019. HPCC: High Precision Congestion Control. In *SIGCOMM*.
[56] Zhonghai Lu, Mingchen Zhong, and Axel Jantsch. 2006. Evaluation of On-chip Networks Using Deflection Routing. In *GLSVLSI*.
[57] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and

Amin Vahdat. 2019. Snap: A Microkernel Approach to Host Networking. In *SOSP*.
[58] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A Receiver-driven Low-latency Transport Protocol Using Network Priorities. In *SIGCOMM*.
[59] Dheevatsa Mudigere et al. 2022. Software-Hardware Co-Design for Fast and Scalable Training of Deep Learning Recommendation Models. In *ISCA*.
[60] Aisha Mushtaq, Radhika Mittal, James McCauley, Mohammad Alizadeh, Sylvia Ratnasamy, and Scott Shenker. 2019. Datacenter Congestion Control: Identifying What is Essential and Making it Practical. *SIGCOMM CCR* (2019).
[61] Eugene Opsasnick. 2011. Buffer management and flow control mechanism including packet-based dynamic thresholding.
[62] Jae-Hyun Park, Hyunsoo Yoon, and Heung-Kyu Lee. 1999. The deflection self-routing Banyan network: A large-scale ATM switch using the fully adaptive self-routing and its performance analyses. *IEEE/ACM transactions on networking* (1999).
[63] Vinod Rajan and Yul Chu. 2005. An enhanced dynamic packet buffer management. In *IEEE ISCC*.
[64] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *SIGCOMM*.
[65] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. 2017. Carousel: Scalable Traffic Shaping at End Hosts. In *SIGCOMM*.
[66] Erfan Sharafzadeh, Sepehr Abdous, and Soudeh Ghorbani. 2023. Understanding the Impact of Host Networking Elements on Traffic Bursts. In *NSDI*.
[67] Naveen Kr. Sharma, Antoine Kaufmann, Thomas Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. 2017. Evaluating the Power of Flexible Packet Processing for Network Resource Allocation. In *NSDI*.
[68] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. 2018. Approximating Fair Queueing on Reconfigurable Switches. In *NSDI*.
[69] Weiguang Shi, Mike H MacGregor, and Pawel Gburzynski. 2005. A Scalable Load Balancer for Forwarding Internet Traffic: Exploiting Flow-level Burstiness. In *ANCS*.
[70] X. Shi, L. Wang, F. Zhang, K. Zheng, and Z. Liu. 2017. PABO: Congestion Mitigation via Packet Bounce. In *IEEE ICC*.
[71] Vishal Shrivastav. 2019. Fast, Scalable, and Programmable Packet Scheduler in Hardware. In *SIGCOMM*.
[72] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. 2019. Shoal: A Network Architecture for Disaggregated Racks. In *NSDI*.
[73] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. 2020. 1RMA: Re-Envisioning Remote Memory Access for Multi-Tenant Datacenters. In *SIGCOMM*.
[74] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. Programmable Packet Scheduling at Line Rate. In *SIGCOMM*.
[75] Renata Teixeira, Aman Shaikh, Tim Griffin, and Jennifer Rexford. 2004. Dynamics of hot-potato routing in IP networks. In *International Conference on Measurement and Modeling of Computer Systems*.
[76] Vojislav Đukić, Sangeetha Abdu Jyothi, Bojan Karlaš, Muhsen Owaida, Ce Zhang, and Ankit Singla. 2019. Is Advance Knowledge of Flow Sizes a Plausible Assumption?. In *NSDI*.
[77] Fábio L. Verdi and Gustavo V. Luz. 2023. InFaRR: In-network Fast ReRouting. *IEEE TNSM* (2023).
[78] Junling Wang and Srihari Nelakuditi. 2007. IP Fast Reroute with Failure Inferencing. In *INM*.
[79] Siyu Yan, Xiaoliang Wang, Xiaolong Zheng, Yinben Xia, Derui Liu, and Weishan Deng. 2021. ACC: Automatic ECN Tuning for High-Speed Datacenter Networks. In *SIGCOMM*.
[80] 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 *SIGCOMM*.
[81] Yifan Yuan, Omar Alama, Jiawei Fei, Jacob Nelson, Dan RK Ports, Amedeo Sapio, Marco Canini, and Nam Sung Kim. 2022. Unlocking the Power of Inline Floating-Point Operations on Programmable Switches. In *NSDI*.
[82] Kyriakos Zarifis, Rui Miao, Matt Calder, Ethan Katz-Bassett, Minlan Yu, and Jitendra Padhye. 2014. DIBS: Just-in-time Congestion Mitigation for Data Centers. In *Eurosys*.
[83] Rachid Zarour and HT Mouftah. 1993. Bridged shuffle-exchange network: A high performance self-routing ATM switch. In *Proceedings of ICC'93-IEEE International Conference on Communications*.
[84] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. 2017. High-resolution Measurement of Data Center Microbursts. In *IMC*.

A ALTERNATIVE PRIORITIZATION PARADIGMS FOR PREEMPTIVE DEFLECTION

The majority of internal workloads in datacenters are RPC style [13]. In these types of workloads, that are controlled by the service provider, flow sizes are typically known [58]. This makes Vertigo [6] and Preemptive Deflection a good fit for RPC workloads. While using the remaining bytes of flows for packet scheduling and deflection is shown to improve the overall latency [6, 11, 60, 76], the flow size information might not always be available. Accordingly, we also simulate Preemptive Deflection (PD) using the flow aging paradigm for packet prioritization. We use the Least Attained Service (LAS) paradigm [14] for this purpose. In particular, the packets are marked with the number of bytes successfully sent from their corresponding flows. Table 1 presents the 99^{th} percentile QCT achieved by different techniques under various degrees of load. For other priority-based techniques, such as AIFO and Vertigo (Selective Deflection), we use SRPT prioritization as their flow aging counterparts result in higher latency. Our results show that while using flow aging, Preemptive Deflection outperforms ECMP, DIBS, and AIFO by achieving 33%, 73%, and 88% lower 99^{th} percentile QCT under 95% load.

Technique →	ECMP	DIBS	AIFO	Vertigo	PD	
Load ↓					SRPT	LAS
55%	0.033	0.013	0.151	0.014	0.014	0.016
65%	0.042	0.034	0.161	0.018	0.019	0.034
75%	0.044	0.076	0.311	0.018	0.036	0.044
85%	0.072	0.164	0.342	0.021	0.047	0.055
95%	0.113	0.281	0.641	0.023	0.086	0.076

Table 1. 99^{th} percentile QCT (seconds) for Preemptive Deflection with SRPT and LAS prioritization paradigms, compared to other in-network alternatives.

B STATISTICAL DISTRIBUTION MAPPING FOR DISTRIBUTION-PD

To decide on the distribution that closely fits the packets' priorities, we run Selective Deflection (Vertigo [6]) under different combinations of background and incast workloads, record the priority of packets that arrive at switch ports, and calculate the Sum Square Error (SSE) while fitting different distributions to them. Table 2 presents the SSE for the uniform, normal, exponential, and Pareto distributions. We observe that fitting an exponential distribution to the packet priorities results in the lowest SSE under various workloads. For instance, under 35% load, fitting exponential distribution results in 46.18%, 34.70%, and 21.98% lower errors than the uniform, normal, and Pareto distribution, respectively.

Distribution →	Uniform	Normal	Exponential	Pareto
Workload ↓	$\times 10^{-15}$	$\times 10^{-15}$	$\times 10^{-15}$	$\times 10^{-15}$
25% background + 10% incast	135.9082	112.0216	73.14878	93.75750
25% background + 30% incast	553.0061	514.3600	366.0715	453.0258
25% background + 60% incast	810.8242	758.0309	483.1348	687.8197

Table 2. Sum square error of different distributions mapped to the priority of the packets observed by a switch port under different workloads

C ACCURACY VS. RESOURCE CONSUMPTION TRADE-OFF

Both quantile-based and statistical methods introduce some approximation errors in estimating the relative priority of an incoming packet. Figure 15 uses the sequence of packets presented in Figure 2 to demonstrate the difference between using an exponential distribution and the quantile calculation for determining the relative priority. In this example, we assume that the queue can

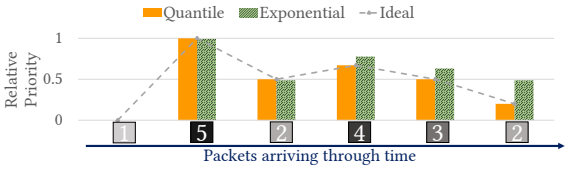


Fig. 15. Quantile-based preemptive Deflection estimates the relative priority more accurately than its distribution-based counterpart.

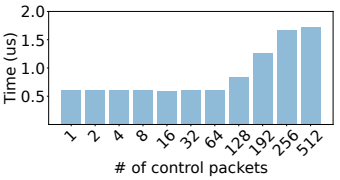


Fig. 16. The time between queue occupancy information updates is less than $2\mu s$ with various number of *control* packets.

accommodate six packets and estimate the quantile of every incoming packet by comparing its priority to the window of four recently enqueued packets. We observe that using distribution mapping results in more approximation errors compared to quantile calculation. For instance, while only one packet has a higher priority than the last packet in the sequence, the distribution-based technique determines its relative priority as 0.49, assigning it a 2× lower priority than the ideal case. Meanwhile, the quantile-based method estimates the relative priority as 0.25, resulting in 25% approximation error, compared to the ideal case. However, calculating the quantile of a packet is more resource-intensive than mapping a statistical distribution. In particular, calculating a packet's quantile (using only 20 previously enqueued packets) in a Tofino switch requires 4× more processing stages than the distribution-based method. Selecting the ideal method depends on the switch memory and processing resources, such as the number of available stages in PISA architecture.

D QUEUE OCCUPANCY UPDATE FREQUENCY IN THE PHYSICAL TESTBED

It takes some time for a *control* packet to recirculate inside the switch and transfer the most recent queue occupancy information, and average priority information in case of distribution-based Preemptive Deflection, from the egress to the ingress pipeline. To investigate the update frequency, we measure the average time gap between queue occupancy information updates for a single port of our testbed switch with various numbers of *control* packets, *i.e.,* the number of active switch ports, under 120 Gbps load. Figure 16 illustrates that with up to 64 *control* packets, the recirculation causes negligible overhead in the update frequency. However, as the number of active ports increases from 64 to 512, the average time taken for updating the queue information increases by 2.8×. As we observed in §6, Preemptive Deflection performs effectively as long as the time gap between information updates is less than $2\mu s$, which is the case even with 512 active ports.

E SIMPLE DEFLECTION APPROXIMATION

While implementing Simple Deflection, to randomly choose a port toward neighboring switches, we generate a random number r between 0 and the number of neighboring switches and deflect the packet to the first port whose corresponding bit in the queue occupancy bitmap is 0 and is located after the r'th bit of the bitmap. Here, we prove that this technique approximately selects the ports for packet deflection uniformly at random.

Proof. To prove that our selection is performed uniformly at random, we show that the probability of deflecting a packet to port i, given that there exists at least one non-congested port toward the neighboring switches (Y), $P(X = i|Y)$, is equal to $\frac{1}{n}$, n being the number of neighboring switches.

$$P_r(X = i|Y) = \frac{P(X = i, Y)}{P(Y)} \tag{2}$$

Assume that $P_f(i)$ and $P_r(i)$ represent the probability of the queue of the i'th port being full, and thus its corresponding bit in the bitmap being 1, and the probability of the randomly generated

number, r, being equal to i, respectively. Accordingly, the probability of deflecting a packet to port i, while at least one of the ports is not congested, $P(X = i, Y)$, is:

$$\begin{aligned} P(X = i, Y) = (1 - p_f(i))[&P_r(i) + P_r(i-1)P_f(i-1) + P_r(i-2)P_f(i-2)P_f(i-1) + ... \\ &+ P_r(1)P_f(1)P_f(2)...P_f(i-1) + P_r(n)P_f(n)P_f(1)P_f(2)...P_f(i-1) \\ &+ P_r(n-1)P_f(n-1)P_f(n)P_f(1)...P_f(i-1) + ... \\ &+ P_r(i+1)P_f(r+1)P_f(r+2)...P_f(n)P_f(1)...P_f(i-1)] \end{aligned} \quad (3)$$

In particular, the probability of port i being chosen for packet deflection is calculated as the probability of port i not being congested and either $r = i$ or $r \neq i$ and all the bits after the r'th bit till the i'th bit in the bitmap are 1. Note that we are implementing a circular paradigm for choosing the first 0 in the bitmap after the r'th bit. For instance, assume that our bitmap (B) indicates 10011 for the ports toward the neighboring switches and $r = 4$. In this example, the sequence at which we search for the non-congested port is $B[4], B[5], B[1], B[2]$ and send the packet to the port corresponding to $B[2]$.

For our approximation, we assume that the load is evenly distributed through the network. Accordingly, every port in a switch has the same probability (p) of being congested.[9]

$$P_f(1) = P_f(2) = ... = P_f(n-1) = P_f(n) = p \quad (4)$$

Since we choose r uniformly at random, the probability of choosing each bit ($P_r(i)$) is equal to $\frac{1}{n}$. Considering equation 4, we can change equation 3 as below:

$$\begin{aligned} P(X = i, Y) &= (1-p)[(\frac{1}{n}) + (\frac{1}{n})p + (\frac{1}{n})p^2 + ... + (\frac{1}{n})p^{i-1} + (\frac{1}{n})p^i + (\frac{1}{n})p^{i+1} + ... + (\frac{1}{n})p^{n-1}] \\ &= (1-p)[\frac{1}{n}\sum_{j=0}^{n-1} p^j] = (1-p)(\frac{1}{n})(\frac{1-p^n}{1-p}) = \frac{1}{n}(1-p^n) \end{aligned} \quad (5)$$

Using equation 4, the probability of having at least one non-congested port can be written as $P(Y) = 1 - P(\text{all ports being congested}) = 1 - p^n$.

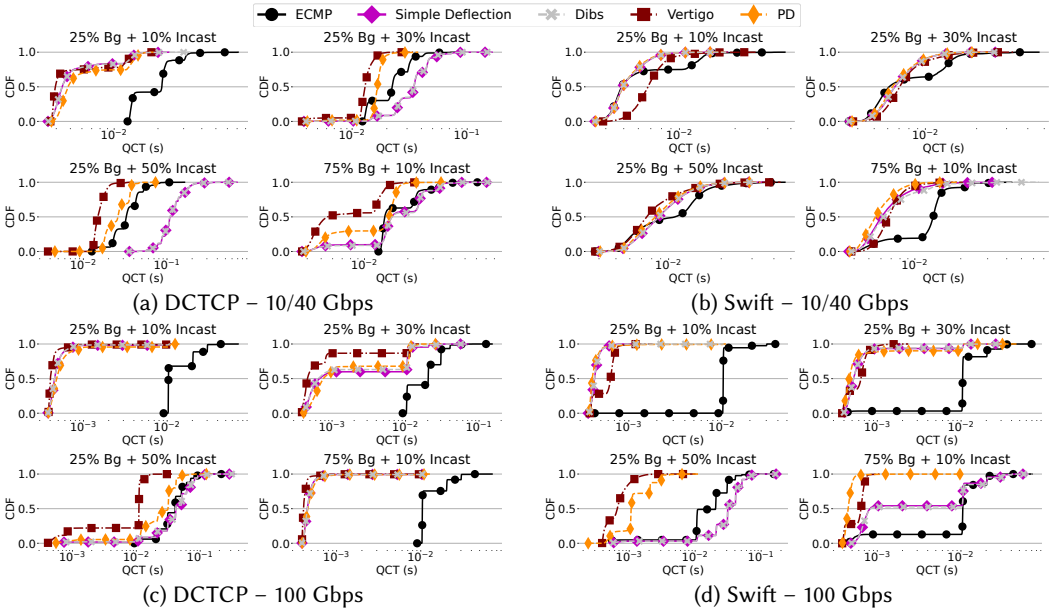
$$P_r(X = i|Y) = \frac{\frac{1}{n}(1-p^n)}{1-p^n} = \frac{1}{n} \quad (6)$$

According to 6, using a circular paradigm while searching for the first zero after the r'th bit in the bitmap, the probability of selecting any of the ports toward the neighboring switches is $\frac{1}{n}$. This shows that our technique of selecting a port for packet deflection is uniformly random. □

F DEFLECTION UNDER OTHER WORKLOAD COMBINATIONS

Figure 17 illustrates the performance of various deflection-based techniques in conjunction with two widely deployed congestion control protocols, DCTCP and Swift, under four different combinations of background load and incast traffic pattern. For these experiments, we simulate a two-tier leaf-spine topology with two sets of link rates: 10/40Gbps links [6, 82] and 100 Gbps links [53, 72, 73, 79]. We observe that performance patterns under distinct workloads are consistent with our observations in §5. Specifically, Simple Deflection performs similarly to DIBS under various workloads. Additionally, when deployed with a window-based congestion control paradigm, such as DCTCP, Preemptive Deflection (PD) outperforms ECMP and Simple Deflection but results in a higher latency than Selective Deflection under load. In particular, on average, PD takes 68% longer than Vertigo to complete the incast queries under 75% load. With Swift, on the other hand,

[9]There are scenarios, in which the likelihood of congestion varies across individual ports, such as with uneven traffic distributions. However, our empirical findings demonstrate that our approximation of Simple Deflection delivers performance on par with that of DIBS [82]. This holds true across various workloads, link rates, and network topologies.



(a) DCTCP – 10/40 Gbps (b) Swift – 10/40 Gbps

(c) DCTCP – 100 Gbps (d) Swift – 100 Gbps

Fig. 17. Preemptive Deflection's performance under other workload combinations with 10/40 Gbps and 100 Gbps links is consistent with our previous observations.

Preemptive Deflection absorbs bursts more effectively. For instance, PD+Swift results in 34% and 9% lower 99^{th} percentile QCT than ECMP+Swift and Vertigo+Swift, with 40/10 Gbps links under 75% load. Similarly, with 100 Gbps, Swift improves Preemptive Deflection's query performance compared to DCTCP due to its faster reaction to congestion. In particular, PD+Swift achieves 17% and 30% lower mean and 99^{th} percentile QCT than PD+DCTCP, respectively, under 85% load.

G PAPER ARTIFACTS

The evaluations in this paper were carried out using network simulations and an Intel Tofino testbed. You can access our codebase for the network simulations and the hardware implementation via our GitHub repository at the following link: https://github.com/hopnets/practical_deflection.git. This appendix provides a brief overview of each module and for more detailed information, we direct readers to the README files available in our repository.

Network simulations. For our network simulations, we utilized Ubuntu 18.04, Omnetpp-5.6.2 [2], and the INET framework [1]. In the README file provided in our repository, you will find a comprehensive guide covering the following essential aspects: 1) Installing project dependencies, 2) Setting up the Omnet++ simulator, 3) Building the project modules, and 4) Executing the simulations and extracting the results.

Hardware implementation. We also publish the code for implementing Simple and Preemptive Deflection on Intel Tofino 1 switches [4]. Our repository comprises separate sub-directories for each specific technique. Particularly, both the data plane and control plane implementations for Simple Deflection, quantile-based Preemptive Deflection, and distribution-based Preemptive Deflection can be found in their respective sub-directories inside the public artifact repository. Building the hardware artifacts requires Intel Tofino SDE.

Received July 2023; revised September 2023; accepted October 2023