DiffECN: Differential ECN Marking for Datacenter Networks

Hanlin Huang, *Graduate Student Member, IEEE*, Ke Xu, *Fellow, IEEE, Member, ACM*, Tong Li, *Member, IEEE*, Zhuotao Liu, Xinle Du, and Xiangyu Gao

***Abstract*— ECN marking has been integrated into datacenter switches to enable high-throughput and low-latency transport. We observe that current marking schemes are coarse-grained: they blindly mark all flows when congestion occurs, causing large flows to occupy undeserved bandwidth and preventing newly arriving small flows from finishing quickly. In this paper, we propose DiffECN, a differential marking strategy that marks only the flows that are the culprits of congestion and protects the remaining flows from being limited. We have implemented it in the Barefoot Tofino switch and performed extensive evaluations via both physical testbed and large-scale simulations. The results show that DiffECN can restrain flows responsible for congestion successfully while providing desirable network performance. For instance, compared to the legacy way of ECN marking, DiffECN achieves up to 32.5% (40.1%) lower average (99th percentile) flow completion time (FCT) for small flows while delivering similar FCT for large flows under production workloads.**

***Index Terms*— ECN marking, congestion control, datacenter network.**

I. Introduction

DATACENTERS host a variety of applications and services with diverse network requirements. For example, web search [1] and distributed memory caches [2] require low latency, whereas others like cloud storage and parallel computing require high throughput. To accommodate these diverse requirements, the community designed multiple end-to-end congestion control algorithms [1], [3], [4], [5], [6], [7]. However, many of them need to be deployed on specialized hardware network interface cards (e.g., Swift [7], HPCC [4]) or require the support of infinite queues (e.g., pFabric [8]), incurring significant deployment effort in large-scale datacenters with heterogeneous fabrics [9]. Additionally, in multi-tenant datacenters, modifying the network protocol stack at the end-hosts imposes another layer of deployment challenges [10], [11].

To complement the host-based congestion control, researchers have explored in-network congestion signals [12], [13], [14]. One example is Explicit Congestion Notification (ECN), which is widely deployed in datacenters. A packet marked with the ECN bit indicates that congestion is imminent. Many production datacenter transport protocols (e.g., DCTCP [1], DECbit [15], DCQCN [3]) rely on ECN to observe congestion and adjust the sending rate accordingly.

However, existing ECN markings may negatively impact network performance, particularly by prolonging the flow completion times (FCTs) for small flows. Existing approaches about ECN marking can be roughly divided into two categories: *per-port* marking [13], [16], [17], [18] and *per-queue* marking [19], [20]. As shown in Fig. 1(a) and (b), all flows traversing the same outgoing port, or sharing the same queue, are marked without distinction. We argue that the granularity of per-port or per-queue marking is too coarse, resulting in unfair bandwidth allocation. When competing against a high-demand flow that is the culprit of congestion, new flows may fail to get their fair shares of bandwidth in time. Our preliminary experiments (see § III-A) show that it takes 15ms and 13ms for new flows to converge to their fair shares under per-port and per-queue marking, respectively, whereas only 0.5ms on an idle link. Small flows may not even receive their fair shares before they finish, leading to service-level agreement (SLA) violations [21].

Motivated by our observations, we propose DiffECN (Differential ECN marking), a simple but effective congestion marking policy that explores a more fine-grained and explicit congestion control. As shown in Fig. 1(c), the key idea of DiffECN is to adopt a more fine-grained scheme toward *per-flow* marking. Specifically, DiffECN only marks "congested" flows that occupy more than their fair shares of bandwidth. By reducing their rates in time, DiffECN helps to make room for the "non-congested" flows, so that these flows can receive their fair shares quickly.

To realize DiffECN, we need to address two primary challenges. The first one is to efficiently manage flows at a per-flow

Received 24 October 2023; revised 24 July 2024 and 23 September 2024; accepted 4 October 2024; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor K. Park. This work was supported in part by the National Key Research and Development Program of China under Grant 2022YFB3102301; in part by China National Funds for Distinguished Young Scientists under Grant 62425201; in part by the NSFC under Project 61932016, Project 62132011, Project 62221003, and Project 62202473; and in part by the China Computer Federation (CCF)-Huawei Populus Grove Fund. *(Corresponding authors: Tong Li; Ke Xu.)*

Hanlin Huang is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China (e-mail: hhl21@mails.tsinghua.edu.cn).

Ke Xu is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China, and also with the Zhongguancun Laboratory, Beijing 100094, China (e-mail: xuke@tsinghua.edu.cn).

Tong Li is with the Key Laboratory of Data Engineering and Knowledge Engineering, Renmin University of China, Beijing 100872, China (e-mail: tong.li@ruc.edu.cn).

Zhuotao Liu is with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China, and also with the Zhongguancun Laboratory, Beijing 100094, China (e-mail: zhuotaoliu@tsinghua.edu.cn).

Xinle Du is with Huawei, Shenzhen 518129, China (e-mail: duxinle1@huawei.com).

Xiangyu Gao is with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China (e-mail: gaoxy21@mails.tsinghua.edu.cn).

Digital Object Identifier 10.1109/TNET.2024.3477511

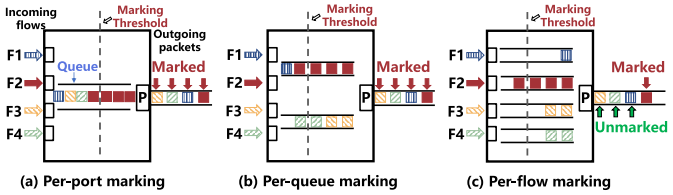


Fig. 1. Examples of different granularity of ECN marking.

granularity level. Given the existing multiple egress queues in modern programmable switches [22], [23], one strawman design is to assign an empty queue for each individual flow. However, given the finite number of available queues, this strawman approach is not scalable in datacenters with high level of flow concurrency and frequent bursts [24]. The second challenge is to precisely identify congested flows on the dataplane. A vanilla design to identify a congested flow is measuring the rate of the flow and comparing it with its fair share of the available bandwidth. However, real-time per-flow rate measurement is computationally intensive and is hardly available on programmable switches (due to the lack of sufficient memory and support of arithmetic division [25]).

To address these challenges, DiffECN designs two innovative components: the Queue Mapping Unit (QMU) and the Selective Marking Unit (SMU). In QMU, we initially track active flows only. We allocate a specific queue with high priority to implement controlled scheduling for flow enqueuing. And we use packet-triggered probing to prevent other queues from suffering starvation. This effectively addresses the first challenge. The second challenge is solved by the SMU. We reconstruct the ECN marking guideline based on fairness among active flows. By separately comparing the buffer occupancy of each active flow with the dynamic marking threshold, we can accurately identify congested flows and apply the necessary marking in time, thereby avoiding challenges associated with rate measurement. At a high level, DiffECN deploys per-flow marking when the number of active flows is within the limit of available queues. However, if the number of active flows exceeds the number of available queues, DiffECN smoothly switches to a modified per-queue marking scheme. To the best of our knowledge, DiffECN is the first work that approximates the goal of per-flow ECN marking in datacenters (see § IV).

We implement a prototype of DiffECN based on a Barefoot Tofino1 switch. The testbed experiments demonstrate that, given a realistic network workload, DiffECN can effectively mark flows with finer granularity and reduce the 99th percentile flow completion time (FCT) of small flows by up to 40.1% without impairing large flows. We further perform large-scale ns-3 [26] simulations. The quantitative results confirm that DiffECN advances state-of-the-art marking schemes in various aspects. Moreover, DiffECN can achieve good performance even in the presence of flow bursts because the queue length remains small.

The rest of the paper is organized as follows. We introduce the background in § II and insights into DiffECN in § III. § IV illustrates the design of our solution. § V describes the implementation. In § VI, we evaluate DiffECN in ns-3 and a small-scale testbed. § VII discusses details and limitations and § VIII surveys the related work. Finally, we conclude this paper in § IX.

II. Background

A. Congestion Marking for End-to-End Congestion Control

Congestion management, developed by Data Center Bridging Task Group [27], consists of two main modules: congestion detection and rate regulation. Generally, congestion detection is performed by switches, while the latter is handled by end-hosts. With the support of ECN, the switch marks CE instead of dropping packets [16]. When the average or instantaneous queue length exceeds the threshold set by the ECN, the switch responds by marking the CE bit as "11" according to the RED algorithm [28], to which the transport layer responds.

Many extant end-to-end congestion control algorithms rely on ECN marking to detect congestion. For example, DCTCP [1] and DCQCN [3], two commonly used protocols in production datacenters, both use RED with conventional ECNs at the switch and marked-packet aware algorithms at the end-host to regulate the rate. The precision of the congestion marking significantly affects the assessment of congestion level by end-to-end congestion control.

B. The Guideline for ECN Marking

In production datecenters, network operators commonly use instantaneous buffer occupancy to mark ECN, enabling faster reaction to traffic bursts [1], [29]. A departing packet is marked only when the instantaneous buffer occupancy surpasses a specific threshold, denoted as K. We consider an idealized scenario where several parallel long flows with the same round-trip time (RTT) share a bottleneck link with capacity C (in bits/s). Previous studies [19], [29] have shown that to maximize link capacity utilization while maintaining low latency, the ECN marking threshold K should be set as follows.

$$K = \lambda * C * RTT \tag{1}$$

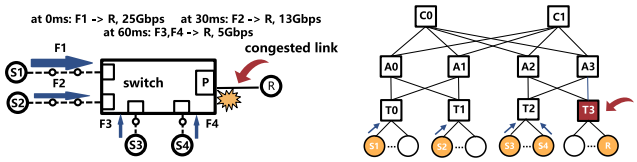
where λ is a parameter decided by congestion control algorithms (e.g., $\lambda = 0.17$ for DCTCP theoretically [30]).

Apart from the port-level setting, an alternative ECN marking scheme is based on per-queue. In this scheme, each egress queue is assigned its own marking threshold. For instance, MQ-ECN [19] implements varying marking thresholds based on the weight assigned to each queue. Rather than considering the port capacity, MQ-ECN utilizes the rate of the queue (denoted as R_i) and sets the threshold as $\lambda * R_i * RTT$. It is evident that regardless of the specific threshold setting method used, flows entering the same port or queue will share the same marking threshold. Consequently, they will experience the same congestion status.

III. Observation and Insight

A. Current Congestion Marking Is Unexplicit

When multiple flows share the same bottleneck link, per-port or per-queue marking policies may impose unexpected problems for flow control. In this section, we conduct fine-grained simulation experiments on a shared bottleneck link to examine the inaccuracy of current marking policies in the presence of flow competition.



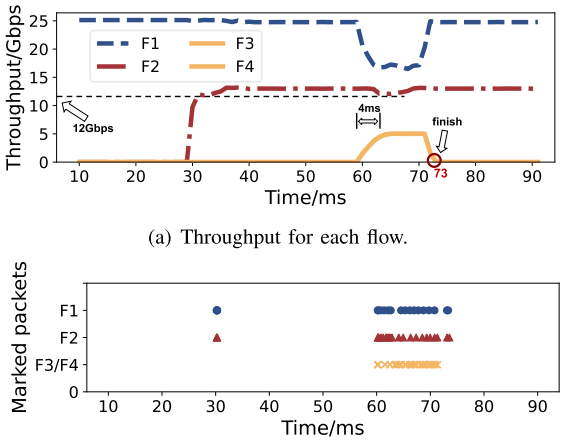
(a) Compact and typical network scenario. (b) Compact unit in Fat-Tree.

Fig. 2. Unfair marking phenomenon.

TABLE I

THE MAX-MIN FAIR BANDWIDTH FOR FLOWS (UNIT: GBPS)

Time(ms)	F1	F2	F3/F4	Time(ms)	F1	F2	F3/F4
30∼60	25	13	0	60∼70	17	13	5



(a) Throughput for each flow.

(b) Markings on flows.

Fig. 3. Throughput and marking under ECN.

TABLE II

THE TIME REQUIRED FOR CONVERGENCE

Scheme	Idle-link	ECN	MQ-ECN	DiffECN
Time(ms)	0.1	4	3.5	0.5

1) Simulation Setup: We use a topology as illustrated in Fig. 2(a). There are four senders S1-S4, and all of them have R as the shared destination. This is a basic topology unit in modern multi-rooted datacenters like Fig. 2(b) (e.g., Fat-Tree [31], Leaf-Spine [32]). In Fig. 2(a), we simulate the end-to-end transmission process in datacenters [33]. The switch corresponds to T3 in Fig. 2(b). The links, each with a speed of 40Gbps, exhibit a propagation delay of 13us for the link from senders to the switch, and one hop delay of 1us for the link from the switch to the receiver.[1] DCTCP is adopted in all end-hosts with recommended congestion control parameters [1]. For the ECN marking threshold, we refer to Eq. (1), consistent with the DCTCP setting.

In this case, S1 and S2 send a long-lived flow to R, and S3-S4 send shorter flows (with a size bigger than one bandwidth-delay product). We assume that the application layer sends data at a fixed rate. Suppose that F1 achieves 25Gbps at the beginning of the simulation. After 30ms, S2 starts F2 at 13Gbps.[2] At this moment, the aggregate transmission rate of F1 and F2 reaches 38Gbps, which is less than the link capacity. Port P has not yet experienced congestion. At 60ms, two concurrent flows, F3 and F4, start at 5Gbps simultaneously. By now, the total speed of the four flows is 48Gbps, exceeding the link capacity. Port P is encountering congestion and the queue starts to build up. In our experiments, all senders are set to have a fixed sending rate when no congestion happens. According to the definition of max-min [34], the fair shares of bandwidth are shown in Table I. When the four flows coexist, their fair shares are 17, 13, 5, and 5 Gbps. We can calculate these by using the water-filling algorithm in § III-B.

2) Throughput Evolution: Fig. 3(a) displays the throughput of these four flows over time. Initially, we observe that F2 takes 13ms to reach 13Gbps throughput from startup. After the arrival of F3 and F4, F2's throughput declines for 4ms. From the above phenomenon, we can see that F2 takes a long time to reach a fair bandwidth at startup. When new flows arrive, its fair bandwidth is compromised again. Initially, we observe that F2 maintains a steady throughput of 13Gbps until 60ms. After the arrival of F3 and F4, its throughput decreases to 12Gbps. But in terms of max-min fairness, when the flows share a 40Gbps bottleneck link, F2's fair share bandwidth should be 13Gbps. Hence, F2's throughput falls short of its fair share.

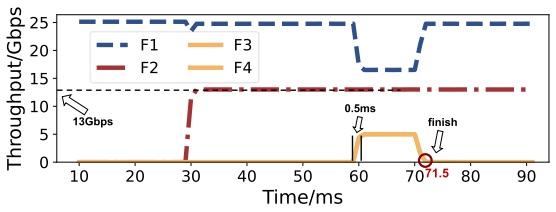
For F3 and F4, their throughput gradually increases until they achieve 5Gbps at around 64ms. It takes up to about 4ms to reach the target rate from startup. From Table II, it only needs 0.1ms to reach the desired bandwidth when the link is idle. The two flows experience a slow ramp-up in throughput, making it challenging to attain a fair share in a short time. Similarly, we also conduct the experiment using per-queue marking (MQ-ECN). The newly coming small flows need 3.5ms to converge to their fair share of bandwidth. It also has an unfair impact on the bandwidth of small flows, leading to longer FCT (see Fig. 15(c) and 15(d)).

3) Analysis: To investigate the cause of the aforementioned issue, we conduct a thorough analysis of the ECN marking of each flow in Fig. 3(b). Not surprisingly, when the congestion occurs on the link at 60ms, F2, F3 and F4 are also marked frequently besides F1. At this point, packets begin to accumulate in port P. When the length of the port exceeds the setting threshold, all packets are marked regardless of which flow the packet belongs to. The same holds for per-queue marking. As packets accumulate, packets in a queue would suffer from the same congestion state. Once marked, the end-host determines the degree of congestion based on the proportion of marked packets and reduces the sending window. This explains why F2's throughput is influenced and F3 and F4 take a comparatively long time to reach a steady state.

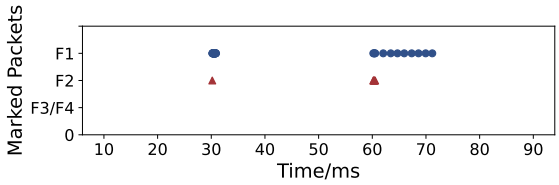
From the above analysis, we observe that all flows traversing the same bottleneck link share the congestion state. The current congestion marking policy cannot explicitly differentiate between flows and treats all packets equally once congestion occurs. Even flows with rates below fair share bandwidth are also marked, resulting in unfair bandwidth allocation. It becomes challenging to achieve the desired throughput promptly for new coming flows. In particular, bursty flows

[1]We set the delay according to PowerTCP [33].

[2]F2 can also include delay-sensitive short flows. For brevity, we set F2 to be a long-lived flow in this case study.



(a) Throughput for each flow.



(b) Markings on flows.

Fig. 4. Throughput and marking under DiffECN.

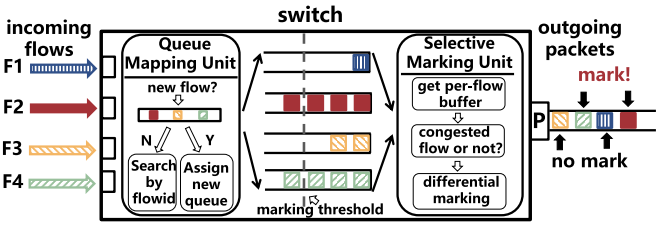


Fig. 5. The main components of DiffECN.

are frequent in datacenters [35], where the latency would be severely affected, leading to unacceptable SLA violations.

Fig. 4(a) depicts the results of using our strategy in the same experimental setting. When the link is congested, the throughput of F2 remains 13Gbps. Moreover, it takes only 0.5ms for F3 and F4 to attain a fair share bandwidth of 5Gbps, while F1 decreases to a fair share of 17Gbps in a shorter period. Finally, F3 and F4 finish about 1.5ms earlier compared to traditional ECN because DiffECN provides more precise control of each flow's rate. DiffECN accurately marks F1 and F2 and do not harm F3 and F4 whose rates are below their fair bandwidth. Deserved bandwidth is guaranteed while ensuring faster completion time for newly arriving flows.

B. It Is Time to Revisit Congestion Marking

Based on our observation in § III-A, we should first curb flows with the rate larger than the fair share. We define the two types of flows as follows.

- Congested flows: The demand rate exceeds the fair share of bandwidth.
- Non-congested flows: The demand rate does not exceed the fair share of bandwidth.

We can refer to the literature [36] to calculate the corresponding fair share allocation based on each flow's demand and the bottleneck link's capacity. The most commonly used calculation is an iterative water-filling algorithm [37]. Intuitively, this algorithm works by initializing all flows to be unconstrained with an allocated bandwidth of 0. For each iteration, it adds equal amounts to all unconstrained flows until at least one flow reaches its demand rate. It then continues to allocate bandwidth to the remaining flows. The algorithm stops iterating until all bandwidth on the bottleneck link has been allocated. In our example, when the four flows F1-F4 coexist, the fair share bandwidths of them are 17, 13, 5, and 5 Gbps, respectively.

In § III-A, both F1 and F2 were non-congested flows until the arrival of F3 and F4. Afterward, F1 becomes a congested flow and the remaining three flows are all non-congested flows. Since the switch treats all flows' states as the same, it marks them when congestion occurs, affecting bandwidth fairness and convergence speed.

To address this concern, the optimal solution is to prioritize the marking of congested flows since they consume more bandwidth than their fair shares. Reducing the rate of congested flows can rapidly release more bandwidth for the remaining flows. Simultaneously, it is crucial to minimize or avoid marking non-congested flows. These flows are not responsible for the congestion and should be protected to ensure they promptly receive their fair share of bandwidth.

IV. Design

Our objective is to develop a per-flow congestion marking scheme that addresses the mentioned issues by marking each flow in a distinct manner. The proposed approach should be straightforward to implement in production datacenters, capable of coexisting with existing requirements without requiring modifications to the current network stack. Furthermore, it should maintain high performance levels across different traffic loads and network sizes.

A. Overview

Fig. 5 illustrates the main components of DiffECN:

1) Queue Mapping Unit (QMU): To determine marking at per-flow granularity, QMU checks if a packet belongs to a new flow when it arrives. It then assigns the packet to an allocated queue or one of the empty outgoing queues on the egress port. This ensures that each flow occupies an empty queue as much as possible. However, a significant challenge arises when dealing with a large number of flows. While modern programmable switches like Tofino1 and Tofino2 offer 32 and 128 queues per port, respectively [22] and [23], the available queue space is still limited compared to the vast number of potential flows. In production datacenters, thousands of services may concurrently transmit traffic [24].

Given this scenario, how can we guarantee that an empty queue is allocated for each flow?

2) Selective Marking Unit (SMU): Before sending out packets of each flow, SMU decides whether to mark them based on the flow's buffer occupancy. The marking of each flow is independent and not influenced by other flows. A straightforward example to illustrate this process is shown in Fig. 5. The switch permits four flows F1-F4 and each flow is matched to a distinct queue on port P. The buffer occupancy of each queue varies due to the different flow rates. Prior to forwarding packets to the link, SMU assesses the need for marking based on the queue length of each flow. Consequently, packets belonging to F2 and F4 are marked, while the other two flows remain unaffected. Although all flows experience the same congestion port, DiffECN treats each flow differentially, marking only the

culprit congested flows and leaving the other flows' normal speed unaffected.

But how does DiffECN precisely identify the flow that needs to be marked, i.e., congested flows?

B. Queue Mapping Unit

To avoid state sharing and head-of-line (HOL) blocking across flows, we dynamically allocate a separate outgoing queue for each flow. When a new flow is detected, an available empty queue is selected to serve this flow. Subsequent packets belonging to the same flow will be directed to the same queue based on the flow's identification. Each flow is uniquely identified by a set of five tuples, namely, source IP, destination IP, source port, destination port, and protocol number, which together form the flow's identification. We determine which queues are empty by tracking the flow's buffer occupancy. As soon as the last packet in a queue is scheduled out, this queue status is reset.

Although the operation may appear straightforward, managing the enormous volume of datacenter traffic poses a significant challenge in providing an empty queue for each flow. To address the scalability limitations of the flow space, we unfold it in two steps:

1) Step1: Only Track Active Flows: Ideal per-flow control requires a substantial buffer to save per-connection state until the connection is disconnected, which is impractical in contemporary datacenter switches. Nonetheless, we observe that saving every individual connection state is unnecessary. Most of the time, the per-connection state is for flows that have no packets queued on the switch, so controlling them is unnecessary.

We define an active flow as one that has one or more packets queued in the switch. In practice, the number of active flows traversing a switch is small [38]. As described in previous subsection, modern Barefoot programmable switches are equipped with numerous queues (32 in Tofino1, 128 in Tofino2), allowing for the allocation of an empty queue for each active flow in general. According to the literature [6], in the Google datacenter, the number of active flows only significantly surpasses the number of queues available under extreme high load. We need to avoid using FIFO queuing strategy for a port because the number of active flows will be large due to a single big flow blocking a multitude of small flows. Instead, the packet scheduler uses deficit round robin [39] to implement fair queuing among active queues (containing packets), but packets within a queue are forwarded in FIFO order. We have conducted a simple test in fat-tree topology with 128 hosts to compare the difference between the traditional and our schemes. Fig. 6 shows the statistics under both schemes over a period of time. The average number of active flows is 25, only about 20% of that of connections. By tracking only active flows, we can drastically reduce the flow space that needs to be maintained.

With dynamic queue assignment, each active flow is allocated a separate queue. We can use the length of the queue as the buffer usage of the flow, which can be directly obtained from the interface provided by the programmable switch [40]. There is no need to frequently update the size for each flow as packets enter and leave the switch, thereby reducing computational overhead. The queue length of a flow solely pertains to its own packets, and DiffECN determines whether to mark the flow based on its queue length, independent of others. Further details regarding the marking policy will be discussed in the subsequent subsection.

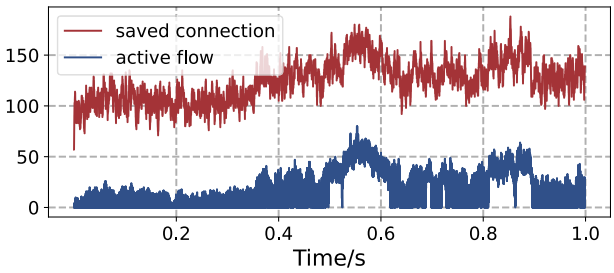
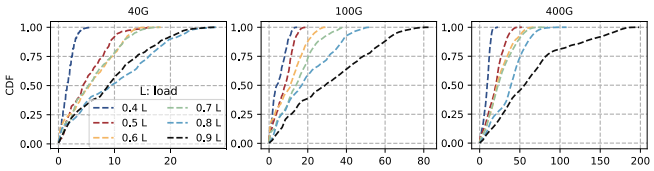


Fig. 6. The number of connections and active flows.

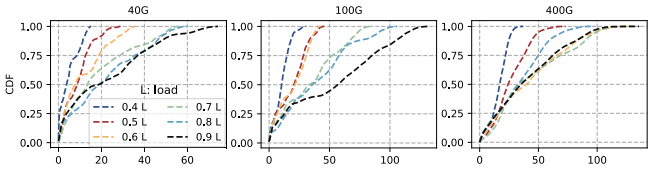
2) Step2: Customize for Bursty Flows: Despite the significant reduction in the burden on available queues by tracking only active flows, the issue of insufficient queues could still arise. Two main factors contribute to this problem. On one hand, the datacenter contains various services. The number of available queues on a switch port may be less than the physical number, e.g., some queues are used for other purposes like traffic isolation [41]. On the other hand, particularly in the case of incast, the number of active flows is likely to surpass 32 (proved in our experiment § VI-C.3), not supported by Tofino1. This further increases the likelihood that the number of available queues is less than the number of active flows.

To address the lack of available queues, we set aside a queue that is granted high priority. When a new flow arrives and all other queues are occupied, it will enter the high-priority queue. Packets in this queue will be dequeued first, prioritized over packets in other queues. However, the introduction of a high-priority queue raises potential starvation of other queues during bursts, thereby prolonging the completion time for the remaining flows. We summarize two primary reasons why starvation may occur, and present the corresponding solutions.

One is that the high-priority queue may be utilized frequently if the number of active flows exceeds the actual number of available queues. We analyze that, unlike the traditional connection-preserving flow counting approach, the probability of queue undersupply can be significantly reduced by considering only active flows as described in the previous subsection. We have tested the active flows in the core switch port under 40G, 100G, and 400G links in the classic spine-leaf architecture. Fig. 7(a) and 7(b) show the results under WebSearch and DataMining workload, respectively. The general trend is that the higher load corresponds to a higher number of active flows. However, there are a few exceptions, like 400G link in Fig. 7(b). In some scenarios, a high load contains more large flows, while a low load contains more small flows. Thus, it is possible that the number of active flows is greater at a low load than that at a high load. It can be seen that the number of active flows can exceed the number of queues only under high link bandwidth and high load (like above 80% load). This result is consistent with the results shown by BFC [6]. In most cases, the switch has enough queues for active flows, especially for Tofino2 (128 queues in a port). Consequently, with the assistance of new hardware, it is uncommon for the high-priority queue to remain occupied.



(a) Active flows under WebSearch worklaod.



(b) Active flows under DataMining worklaod.

Fig. 7. The number of active flows at a port.

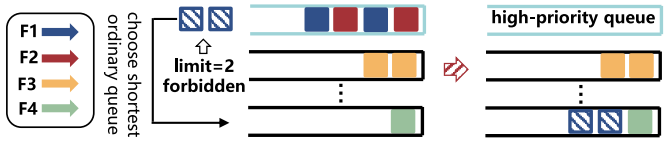


Fig. 8. An example for flows in the high-priority queue.

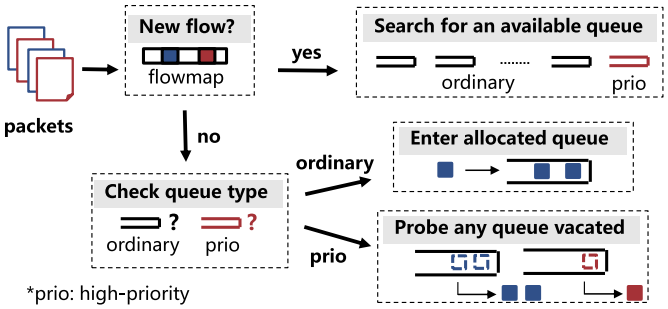


Fig. 9. Queue assignment when enqueuing.

The second reason is that the size of the flow entering the high-priority queue may be large, and the queue will be occupied for a long time until the long flow finishes. For this concern, instead of allowing flows to remain in the queue, we treat the high-priority queue as a "temporary buffer region." Because active flows change dynamically, we monitor the availability of queues whenever a packet belonging to a flow in the high-priority queue enters the switch. When an empty queue is available, the flow is reassigned to the empty queue. So if the flows in the high-priority queue do not continue to send packets, the rest queues have a chance to be emptied. The packet-triggered probing method greatly increases the likelihood that a new flow will enter an empty queue and reduces the frequency of using the high-priority queue.

Nevertheless, considering the worst case where no empty queue is ever vacated in each turn, we cannot allow long flows to continuously occupy the high-priority queue. We set a "*pass limit*" for each active flow entering the high-priority queue, i.e., each flow's limit is independent. When the size of a flow exceeds the limit, it will be moved out of the high-priority queue and choose the shortest queue. We use an example to illustrate it in Fig. 8. In this case, let's assume that the "*pass limit*" is set to 2 packets. As a result, for F2, it can be entirely processed in the high-priority queue. For F1, after the first two packets, it is moved out and the subsequent packets are directed to the shortest queue. It is worth noting that when a flow is moved out, it will affect the original flow in the new queue regardless of which queue it enters. The effect is also unavoidable in the per-queue and per-port schemes. But we can minimize this effect by choosing the shortest queue for the following two reasons. (1) It can prevent the length of a reassigned queue from easily reaching the ECN marking threshold due to the arrival of a new flow, which unfairly affects the original flow. (2) A queue with a shorter length is more likely to contain a short flow (proven in § IV-C.2). So it will complete quickly, allowing a new flow to get access to an empty queue in the shortest possible time. By doing so, we ensure that bursty short flow can finish quickly while avoiding large flows from taking up resources for a long time. Fig. 19 shows that our scheme can achieve better results in large-scale incast.

Therefore, the high-priority queue acts as a "temporary buffer region" to ensure that bursty short flows can finish quickly. And the packet-triggered probing increases the chances that a new flow will enter an empty queue when the available queue is insufficient. If the probing phase is skipped and the shortest queue is chosen directly, it will cause multiple short flows to suffer from HOL problems and prolong their completion time, which is unfriendly to datacenters that emphasize tail latency.

3) Integrated Enqueuing Logic: Based on the above description, we take steps 1 and 2 to address the flow space scalability issue. We only track active flows to shrink the flow space significantly, and then reserve a high-priority queue to respond to bursty flows. With controlled scheduling, the chances of a bursty flow entering an empty queue are increased. Fig. 9 illustrates the queue allocation process.

First, the five-tuple of a packet is extracted and used as the flow identification (FID). Then the allocated flowmap is searched for a corresponding allocated queue. If such a queue doesn't exist, we check the available queues to decide whether to allocate an empty queue or to be conceded to the high-priority queue. Otherwise, the allocated queue is checked to see whether it is the high-priority queue. If it is an ordinary queue, the packet will directly enter it. If the allocated queue is of high priority, the decision to enter the probing phase or choose the shortest queue is based on the number of bytes that the flow has passed through at this moment. During the probing phase, if any empty queue is vacated, it will be assigned to the flow. Once the assignment is made, the queue length is increased by the packet size. In the extreme case where all the memory in the switch is already occupied, the incoming packet will simply be dropped. We use a bitmap to record whether the queue is empty or not, and reset the corresponding bit if the queue length is zero.

The total number of entries in our flowmap is 512 times the number of available queues. It ensures that when the number of flows is less than the number of available queues, the probability of index hash conflicts does not exceed 0.2%. The probability of conflict decreases as the size of the flowmap increases. If the memory of the switch supports it, the conflict will be easily avoided.

C. Selective Marking Unit

When it is time to drain out a packet, we determine whether the flow is congested or not. If it is congested, DiffECN sets the CE field of the packet to 11, otherwise no action is taken. In this way, even in the case of congestion, DiffECN can distinguish each flow individually and mark only congested flows. Then the sender of a congested flow releases excess bandwidth so that small coming flows can finish swiftly. Therefore, how to distinguish between congested and non-congested flows becomes a key issue for marking ECN.

1) How to Identify Congested Flows: Intuitively, the most reliable way to determine whether a flow is congested is by measuring the speed of the flow and comparing it with the fair share. However, in practice, precisely measuring the speed of a flow at a switch is very challenging because the data plane does not support division operations. While it is possible to perform such measurements with the help of the control plane, doing so in real-time is difficult. Additionally, the number of flows is large and speed measurement for each flow would consume a lot of storage resources, which is difficult to be supported by the switch [25].

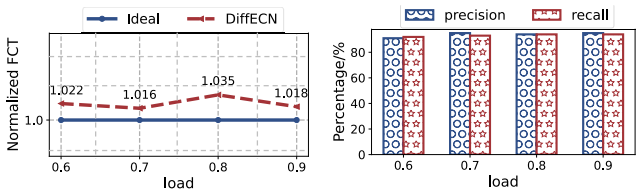
Consider that congested flows are those whose rates exceed fair shares, which are higher than those of non-congested flows. In our scheme, each flow enters a separate FIFO queue. Thus high-demand flows naturally generate more packets in the same amount of time, resulting in longer queues. This phenomenon is demonstrated in the experiment § VI-C.2. As a result, under per-flow management, the more buffer a flow occupies, the more likely it is to be a congested flow. This assures us that it is reasonable to identify congested flows by queue length. In particular, although we react quickly to the congested flow, we avoid curbing it excessively. Given the fairness of bandwidth, we no longer mark congested flows when their rates successfully drop to fair shares or when other flows reach the demand bandwidth.

Thus, it is crucial to set the threshold of buffer occupied as a boundary to distinguish congested flows from non-congested flows. The setting of the marking threshold will then be examined in order to more precisely control high-demand flows and safeguard low-rate flows.

2) How to Set the Marking Threshold According to Buffer Occupancy: As mentioned earlier, before dequeuing each packet, it must compare the queue length to the marking threshold. However, calculating a threshold for each flow is impractical because it requires complex computations and affects fairness (which we will discuss later). For the sake of simplicity and deployment, we decide to set the same threshold for each queue of active flows and use this threshold to distinguish congested flows from non-congested ones.

We observe that the traditional ECN marking schemes using Eq. (1) has achieved good results [1], [29], making full use of the link bandwidth while achieving low latency. Under the per-flow marking idea, we treat the rate separately from the flow's perspective instead of the port and queue. Combined with the scenario where N flows share a port at the same time, each flow should receive a fair share of bandwidth C/N. Therefore, we reconstruct the marking threshold as follows.

$$K_p = \lambda * RTT * C/N_{flow} \tag{2}$$



(a) Normalized AVG FCT (b) Precision and recall

Fig. 10. The comparison of DiffECN with the optimal solution.

where N_{flow} represents the number of active flows. As the flow enters and leaves the switch, N_{flow} also changes dynamically. C/N_{flow} ensures that each active flow has a fair port bandwidth, so it represents the congestion boundary when the bandwidth of each flow reaches a fair share.

Intuitively, we could set separate marking threshold for each flow, based on different flow rates C_i instead of a unified C/N. But it would compromise fairness. In this case, for flows with smaller C_i, their marking thresholds would also be smaller. Then when congestion occurs, they will suffer from marking even if the queue length is low. Whereas for flows with larger C_i, they will have more buffer space due to a larger marking threshold. Thus the culprit high-speed flows are not slowed down in time while innocent low-speed flows are victimized, which is not what we hope to see. Instead, under Eq. (2), small flows can be sped up in time, while flows with larger C_i will be curbed first to free up more bandwidth. In this way, we successfully relate the flow rate to the queue length to avoid the difficulties of measuring the flow rate.

To further justify our judgment of congested flows based on their buffer occupancy, we compare it with the ideal scheme. In the ideal scheme, we know the fair share bandwidth of each flow in advance and determine whether a flow is congested by measuring the speed of the flow in the simulator. In our scheme, only flows whose buffer occupancy exceeds K_p are identified as congested flows. Fig. 10 demonstrates our results. The experiments show that our FCT results are very close to the ideal value. We also measure the key classification assessment metrics, i.e., accuracy and recall [42], as follows:

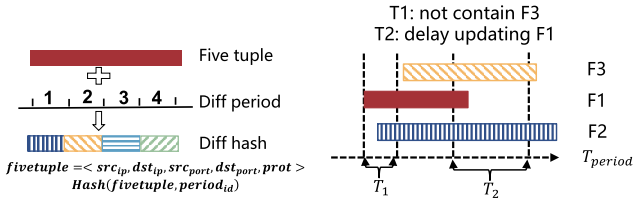
$$Accuracy = \frac{Correct\ predictions}{All\ predictions} \tag{3}$$

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives} \tag{4}$$

These metrics are above 90%, which proves that our identification scheme is effective.

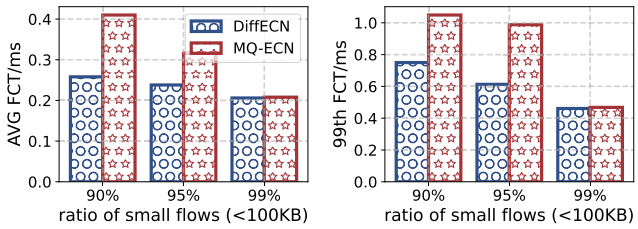
3) How to Count the Number of Active Flows: After determining the method for setting the marking threshold, we need to consider how to count the number of active flows (N_{flow}), a key parameter that affects performance. In our scheme, DiffECN maintains a flow counter. Due to the dynamic nature of when a flow comes and leaves, we need to count the number of flows in real-time. So we need to recount and update it every period. We define the beginning of a flow as a five-tuple packet that first appears in the switch within each timeout-sized period.

However, a problem arises when a flow spans more than one period. How do we distinguish the same flow in different periods? To this end, we use the id of the period and the five-tuple of packets together as the hash key. For each turn,



(a) Flow Counter (b) Different period

Fig. 11. An example of how flow count works.



(a) AVG FCT (b) 99th FCT

Fig. 12. DiffECN under most small flows.

the period id is incremented by 1, thus distinguishing the same flow for different periods. Fig. 11(a) illustrates this process. This new arrival counting method helps to avoid bias in case of the beginning or the end of a flow. It determines whether a flow is new by checking the bitmap maintained by each port. If it is, the flow counter is added by 1. For each period, the bitmap and N_{flow} are reset and recirculated. We count N_{flow} by considering the flows of the previous period (denoted by N_{flow}^{last}) and the N_{flow}^{now} measured in the current period. Then it yields $N_{flow} = max(N_{flow}^{last}, N_{flow}^{now})$. Because the flow counter always starts from 0, we cannot take N_{flow}^{now} as the current N_{flow}.

Furthermore, we also need to consider the upper limit of the number of queues. Under few cases link heavy incast or many active large flows, the number of active flows may exceed the total number of queues (Q_{num}). In these cases, we set N_{flow} to Q_{num}. DiffECN will fall back to a modified per-queue without side effects. We have executed the experiments in a scenario with almost all small flows, where the number of active flows is above the number of queues. The topology settings are consistent with § VI-C.5. We set the load of the bottleneck link to 90%. Fig. 12 shows when the number of active flows exceeds the number of queues, DiffECN falls back to a modified per-queue scheme (MQ-ECN). When large flows exist, DiffECN can maintain better performance for small flows because they can use the high-priority queue or choose the shortest queue when the limit is exceeded. When almost all flows are small, DiffECN behaves similarly to per-queue because there are no more large flows to be sacrificed at this point.

V. Implementation

We have implemented DiffECN in Barefoot Tofino1 programmable switch, including 12 MAU stages, 120Mbit SRAM and 6.2Mbit TCAM per pipeline, 32 queues per egress port, supporting stateful packet manipulation. Ingress/egress ports are statically assigned to pipelines. Our solution can be fully implemented in the data plane without the involvement of the control plane. The development effort on the switch requires about 900 lines of $P4_{16}$ code, and a total of 9 stages are consumed in the switch. Algorithm 1 shows the logic of our data-plane processing. We present two core implementations, namely Queue Assignment and Flow Count.

Algorithm 1 Data Plane Implementation.

Ingress logic (Queue Assignment):
$FID \leftarrow ExtractFiveTuple(packet)$
$P_{index} \leftarrow FID[IndexSize:0]$
if $flowmap[P_{index}]$ not None **then**
  $in_q = flowmap[P_{index}]$
  if in_q is $Priority_Q$ **then**
    switch $PassedSize$ **do** ▷ The number of bytes this flow has passed in the high-priority queue
      case $\leq limit$
        $ProbeEmptyQ(queues)$
      case $> limit$
        $ReassignShortestQ(lengths)$
  end if
else
  $key \leftarrow CombineLength(queues)$
  $emptyQ \leftarrow TernaryMatch_{emptyQ}(key)$ ▷ Simulate looping empty queue by ternary match
  if $emptyQ$ is None **then**
    $flowmap[P_{index}] = Priority_Q$
    $in_q \leftarrow Priority_Q$
  else
    $flowmap[P_{index}] = emptyQ$
    $in_q \leftarrow emptyQ$
  end if
end if
Send the packet to traffic manager.
Egress logic (Flow Count):
if $T_{now} - T_{start} > T_{period}$ **then**
  $Update_{Seq_{interval}(+1)}$ ▷ Increase the interval sequence to distinguish periods
  $Update_{N_{flow}^{last}}(N_{flow}^{now})$
  $Reset_{N_{flow}^{now}}(0)$ ▷ Reset the current period flow number
  $Reset_{T_{start}}(Time_{now})$
else
  $Hash_{new} \leftarrow CaculateHash(FID, Seq_{interval})$
  $P_{index} \leftarrow Hash_{new}[IndexSize:0]$
  $Hash_{old} \leftarrow Read\&Update_{Bitmap}(P_{index}, Hash_{new})$
  if $Hash_{new} != Hash_{old}$ **then** ▷ Find a new flow by comparing current and last hash values
    $N_{flow}^{now} \leftarrow N_{flow}^{now} + 1$
  end if
end if
$N_{flow} \leftarrow max(N_{flow}^{last}, N_{last}^{now})$
$Mark_{threshold} \leftarrow Match\&Read(N_{flow})$ ▷ Different number of flows corresponds to different marking threshold
if $Queue_{len} > Mark_{threshold}$ **then**
  $MarkECN(packet)$
end if
Send out the packet.

A. Queue Assignment

When a packet arrives, DiffECN needs to know if it belongs to a new flow. We maintain a *flowmap* for flows. It records the queue identification assigned to each flow, indexed by the five-tuple hash of the flow. Specially, we take the first $IndexSize$ bits of the hash as the index. For an arriving packet, we extract its five tuple and then get the queue identification based on the hash index.

If the queue id can not be found in the *flowmap*, then it belongs to a new flow and needs to allocate an empty queue for it (lines 14-21). Since P4 does not support loop operation, we cannot traverse all queues. Fortunately, ternary matching can simulate the process of looping by matching the entries in order, with the more preceding entries having higher priority. Specifically, we input the current lengths of all queues and combine them in order as keys. DiffECN then looks for an empty queue starting from queue 0 until 31 (the last queue id). Once an empty queue $emptyQ$ is found, it exits the match process (the priority of current matching entry is higher than the rest) and updates the *flowmap*. That is, ternary matching always finds the empty queue with the smallest sequence number.

For flows that have been assigned the high-priority queue, we need to enter the probing phase. The packet-triggered probing detects if there is an empty queue. If an empty queue is still not found before the size this flow has passed exceeds the "*pass limit*," we reassign it with the shortest queue (lines 4-11).

It is important to note that queue assignment needs to be finished at ingress because enqueuing is done in the traffic manager and the queue identifier needs to be specified before. We implement it by using recirculate now. Although the operation causes overhead to the data plane, it can be reduced by periodic updates. Moreover, Tofino2 has already provided a built-in functionality to avoid the overhead. We have discussed these in § VII.

B. Flow Count

DiffECN needs to know how many active flows are passing through the switch's port at any time. As described in previous section, the switch only needs a bitmap and a counter to achieve this functionality. However, in a pipeline, a stage can only operate on one register, not an array of bitmap registers. To solve this problem, DiffECN uses a time period sequence counter and a write-after hash operation to calculate the number of active flows. The lines 26-39 of the algorithm show this process, which is implemented in egress.

For an arrived packet, it firstly check whether the current time T_{now} exceeds the last update time T_{start} more than T_{period}. We define exceeding this period as a timeout for ease of presentation. If timeout occurs, we need to reset the value of the register. For example, the $Seq_{interval}$ for counting time period is increased by 1 to distinguish between different periods. The current N_{flow}^{now} is assigned to the number of flows counted in the last period (i.e., N_{flow}^{last}). And N_{flow}^{now} is reset to 0. Finally, T_{start} is updated to the current time T_{now}, which is used as the start time of the new statistical interval.

If no timeout happens, it read the current period sequence counter. Do hash operation together with the FID of packets and $Seq_{interval}$ to get $Hash_{new}$, as the index P_{index} of the lookup storage array. The most critical step is that we read the hash $Hash_{old}$ at the P_{index} position, and update the current register to $Hash_{new}$. Then compare whether $Hash_{new}$ and $Hash_{old}$ are equal. If they are not equal, then a new flow is detected and the counter N_{flow}^{now} (lines 35-36) is updated. The $Seq_{interval}$ is used for hash of a packet because we want to treat a flow in different periods as a different flow, to count the number of active flows in real-time. In our implementation, the size of the bitmap array used to store the hash values is 2^{14} uints. The number of active flows is finite, so the probability of different flows' hashing conflict is low and does not affect performance much.

Finally, P4 does not support multiplication and division operations. So we need to calculate the dynamic thresholds K_p in advance, which is affected by N_{flow}, and store them in the table. Before the packet is dequeued, the length $Queue_{len}$ of the corresponding queue is read. Then the comparative relationship between $Queue_{len}$ and K_p determines whether it needs to be marked (lines 40-43).

C. Implementation Experience

In one packet processing pipeline, multiple tables can be stored in a stage, but only one table can be modified (i.e., a register can only be modified once). We can modify registers located in several different stages. Therefore, tables that need to be modified in the same pipeline must be placed in different stages (corresponding to multiple registers). For example, in our implementation, *flowmap*, *update_now_flow_count*, *window_seq*, *flowhash*, etc. are placed in different stages, while tables that are not in the once pipeline, like *read_window_seq* and *update_window_seq*, can be placed in the same stage.

VI. Evaluation

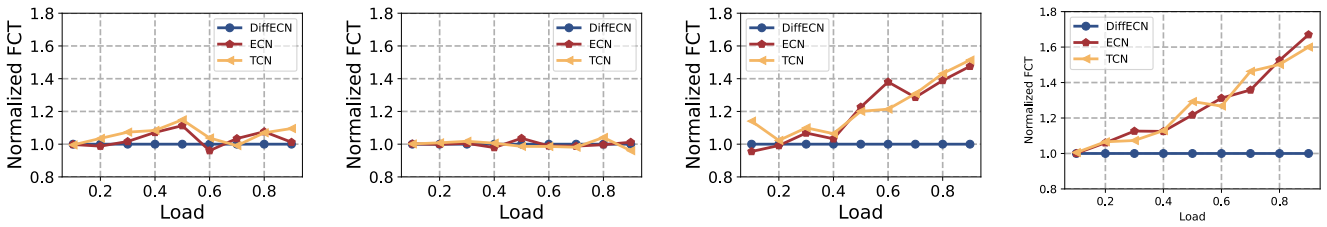
In this section, we perform testbed experiments and ns-3 simulations to answer the following key questions:

- How does DiffECN perform in practice?
- Does DiffECN scale to large datacenter topologies?
- Why can DiffECN guarantee fairness among different flows?
- How tolerant is DiffECN to bursty traffic?

A. Methodology

1) Schemes Compared: We compare DiffECN with the following five schemes: (a) ECN [testbed, simulation]: We use the typical per-port marking according to Eq. (1). (b) MQ-ECN [simulation]: MQ-ECN marks flows on a per-queue basis, and we set all queues with the same weight. (c) TCN [testbed, simulation]: By detecting packet sojourn time $T_{sojourn}$ in the switch, TCN compares it with $\lambda * RTT$ to determine congestion, referring to [17]. (d) ECN# [simulation]: ECN# [18] marks packets based on both instantaneous and persistent congestion. (e) CoDel [simulation]: CoDel is also a per-queue scheme and tracks minimal queueing over an interval to mark packets [20].

2) Benchmark Traffic: We generate traffic based on two real workloads in production datacenters: WebSearch [1] and DataMining [43]. According to [19], both workloads are heavy-tailed. For example, in webSearch, 70% of the flows



(a) Overall AVG (b) Large Flow AVG (>10M) (c) Small Flow AVG (<100K) (d) Small Flow 99th (<100K)

Fig. 13. [Testbed] FCT statistics with WebSearch workload.

are less than 1MB and 97% are less than 10MB. While in DataMining, 91% of the flows are less than 1MB and 95% are less than 10MB. In the testbed experiment, we use an open-source traffic generator [44] to generate benchmark traffic. Referring to the common practice in academia [18], [45], [46], we also generate traffic based on Poisson assumptions to capture the statistical temporal traffic patterns (burstiness on different time scales). And we use the same scheme to generate flows in our simulation experiments.

3) Network Load: Under all-to-one communication mode, the network load refers to the load of the bottleneck link between the switch and the receiver. Under all-to-all, it refers to the load on the links between the switches adjacent to servers and the upper layer switches (e.g., spine-leaf links). Most datacenter networks operate at loads less than 50% [47] and the instantaneous load becomes high when bursty traffic comes. So consistent with classic practice [13], [48], we explore DiffECN's performance under different loads (from 10% to 90%) separately.

4) Metric: We use flow completion time (FCT) as the main metric. In addition to the overall average FCT, we also decompose the FCT results across different flow sizes (small (<100K) and large (>10M) flows). To evaluate the tail latency, we also show the 99th percentile FCT for small flows. All results are normalized according to the results achieved by DiffECN.

B. Testbed Experiments

Testbed Setup: We build a small-scale testbed with 5 servers connected to a Barefoot Tofino1 switch with DiffECN's implementation. Each egress port has 32 queues. There are four senders and one receiver. Each sender starts 10 services, simulating a 40-to-1 request totally. Each server is with a 4-core Intel Xeon E5620 2.4GHz CPU, 16G memory, and a 160GB hard disk. All servers are running Linux kernel 4.15.0 and DCTCP is enabled in the host. The base RTT is 0.19ms and all links are 10Gbps.

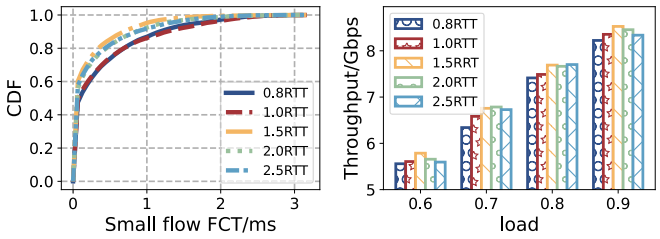
1) Realistic Workload: We first verify the functionality of DiffECN in a real traffic scenario. For generality, we test the FCT under different network load from 10% to 90%. The threshold of ECN is set as 510 cells according to Eq. (1). In Tofino switches, queue lengths are in units of cells, where a cell represents 80 bytes [49]. The $T_{sojourn}$ of TCN is set to 0.03ms. We calculated all dynamic thresholds for DiffECN in advance according to Eq. (2) and put them in a match table. For example, the threshold is set as 510 cells when there is one active flow and 255 cells under two. We generate traffic based on the real workload WebSearch. Figure 13 shows the FCT results. They are a breakdown in terms of overall FCT, FCT of large flows (>10MB) and FCT of small flows (<100KB). All results have been normalized to FCT achieved by DiffECN.

a) Small flows: The network operators are most concerned about the performance of small flows. From subfigures (c) and (d), it is easy to see that when the network load is low, the gap among these schemes is not big. When the load becomes higher, DiffECN's advantage starts to emerge. And the higher the load, the better results achieved by DiffECN. Specifically, compared to current practice (i.e., ECN), DiffECN can achieve up to 32.5% lower average FCT at 90% load. In the 99th percentile FCT, DiffECN can achieve up to 40.1% lower FCT. Compared to TCN, DiffECN maintains a similar advantage over it.

The reason is that when the load is high, ECN and TCN do not differentiate flows and will mark ECN for small ones as well, preventing them from getting enough bandwidth. DiffECN, on the other hand, will accurately curb high-speed flows and not mark small ones, providing them with the deserved bandwidth. Therefore small flows can finish earlier compared to the other schemes. And the higher the load, the more obvious it is that traditional schemes hurt small flows. Because they can even cause packet loss, which can seriously affect performance.

b) Large flows and overall: From subfigure (b), we see that DiffECN achieves similar results as ECN and TCN. There is only a small difference under high load. For example, ECN achieves up to 1.4% lower FCT and TCN achieves up to 2.9% lower FCT at 90% load. Nevertheless, these side effects are negligible. This is because DiffECN curbs the bandwidth of large flows in order not to impair small flows. However, the duration of the congestion is short and does not seriously affect the throughput of large flows. In general, DiffECN achieves good overall performance among the three schemes (subfigure (a)). It further demonstrates that DiffECN can fully utilize link capacity and maintain the throughput of all flows.

2) Reasonable Period to Count Active Flows: In the testbed, we count and update the number of active flows at each period, so the setting of T_{period} is crucial for performance. First, it must be greater than a round-trip time, otherwise, the estimate of N_{flow} will be less than the ground truth, because all flows need at least one RTT passing through the switch. However, it should not be too large either. If the number of newly arriving flows increases, the update of N_{flow} will be delayed. From Fig. 11(b), T_1 is smaller than one RTT while T_2 is too large. It cannot count F3 under T_1 because F3 arrives at the switch later. For T_2, although it counts all flows



(a) CDF of FCT. (b) Throughput under high load

Fig. 14. [Testbed] DiffECN performance under different T_{period}.

accurately, it does not update in time when F1 ends. Through extensive experiments, it shows that setting T_{period} to 1.5RTT is reasonable. We have tested the performance of DiffECN by setting T_{period} to 0.8RTT, RTT, 1.5RTT, 2.0RTT, 2.5RTT respectively. The parameter settings and traffic patterns are all similar to the previous subsection.

Figure 14(a) shows the distribution of small flows' FCT. 1.5RTT for T_{period} achieves the best results, with 97% of small flows completing within 1ms. 2.0RTT and 2.5RTT also perform well, with little difference from 1.5RTT. But 0.8RTT and 1.0RTT perform poorly. This is due to the fact that when T_{period} is greater than one RTT, it ensures that all active flows pass through the switch. Conversely, the estimation of N_{flow} is small and the threshold K_p is large, resulting in a long queue, which has a negative impact on small flows.

Figure 14(b) shows the throughput gains for various T_{period} settings. When the network load is high, 1.5RTT can also maintain high link utilization. 2.0RTT and 2.5RTT are slightly worse in both FCT and throughput because updating N_{flow} is delayed. It can lead to the estimated N_{flow} being larger or smaller than the ground truth. With the above analysis, we believe that setting T_{period} to 1.5RTT is a good choice for practical production. Operators can customize it by applying a similar analysis on their networks' traffic traces.

C. Simulation Experiments

In this section, we use ns-3 simulator to evaluate the performance of DiffECN in multi-hop datacenter networks. DCTCP is still adopted as the default transport protocol, unless specified. Unlike the testbed, we can accurately count active flows in real-time, eliminating the bias on estimating N_{flow}. Also, we will answer why DiffECN can guarantee fairness by fine-grainedly analyzing the buffer occupied by flows with different demands. We further explore the performance of DiffECN under bursty flows. Finally, DiffECN is also proven to cooperate well with DCQCN in lossless fabric.

1) All-to-All Communication: To complement the small-scale testbed, we evaluate DiffECN in a production datacenter with the multi-hop spine-leaf topology.

a) Setup: We simulate a 128-host leaf-spine topology with 8 leaf switches and 8 spine switches. Each leaf switch has 16 40Gbps links to hosts and 8 40Gbps links to spine switches. The latency of the link is 15us. We use ECMP for load balance. K in ECN is set to 65KB and $T_{sojourn}$ in TCN is set to 11us. For CoDel, we set the interval to 150us and the target to 10us. These settings are suggested in Eq. (1) and literature [18].

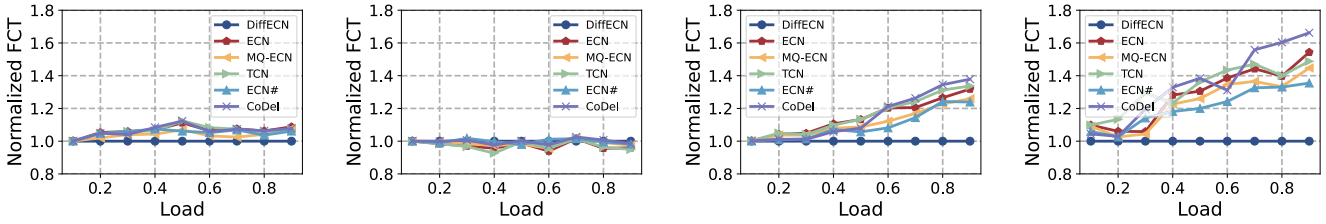
b) Results: Figure 15 shows the FCT results across different flow sizes under WebSearch. For the overall FCT, DiffECN slightly outperforms another five schemes, which proves that it can maintain good throughput in production. For the small flows' average FCT, DiffECN achieves up to 16.1% lower FCT than ECN and up to 25.9% lower FCT than MQ-ECN. As expected, DiffECN achieves larger improvements at the 99th percentile: the 99th FCT of DiffECN (2.13ms) has a reduction ranging from 26.0% (ECN#, 2.88ms) to 39.8% (CoDel, 3.54ms) at 90% load. For larger flows, this has a negligible increase in the FCT of DiffECN (128.8ms), ranging from 1.5% (MQ-ECN, 126.9ms) to 4.9% (ECN, 122.5ms) at 90% load. Under the advantage of significant gains obtained for small flows, the impact on large flows is not notable. This demonstrates that DiffECN can outperform other marking algorithms in real workload scenarios. The result is also consistent with the testbed.

We also test DiffECN's performance under DataMining workload. From the Figure 16, DiffECN maintains a similar advantage. The difference is that the performance superiorities of small flows is slightly degraded under high load compared with WebSearch's. This is due to the fact that the number of long flows in DataMining is quite small, while long flows are the congested ones that really need to be limited. So some small flows with larger size are marked ECN. From the average FCT in subfigure 16(a), DiffECN can still maintain a stable throughput. From the results under the above two workloads, DiffECN is able to achieve good performance in production datacenters, where the majority of flows are short flows but most of the bytes come from large flows [50].

2) Fairness: In § IV-C, we distinguish between congested and non-congested flows based on queue length. This is based on the premise that congested flows tend to have higher rates and longer queues. We have confirmed it in § IV-C.1. To further illustrate this basis, we measure the queue length of each flow using DiffECN during the period when congestion occurs in the preliminary test in § III-A. Fig. 17(a) illustrates this result. As we expected, when the four flows coexist and reach the stable state, the larger flows F1 and F2, with 17Gbps and 13Gbps, have longer queue length. While F3 and F4's queues are shorter because they are small flows with lower rates, i.e., 5Gbps. Thus, it is not hard to find that the queue length of every flow is proportional to its rate.

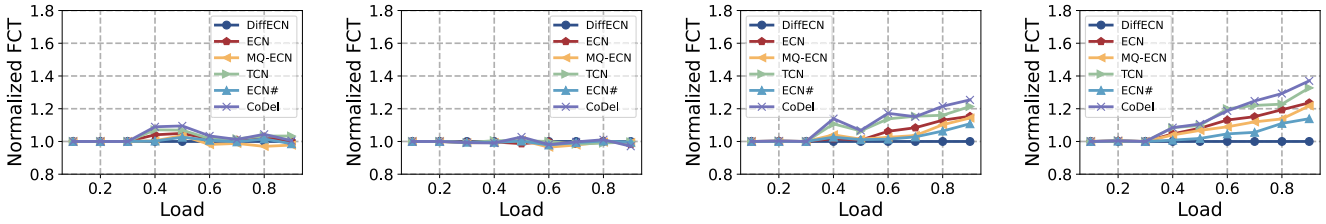
With the same experimental setup, we further explore whether the bandwidth can be fairly distributed when flows with the same demand compete for the link. We set the demands of F1-F3 to be 15Gbps and F4 to be 5Gbps. Figure 17(b) shows that F1-F3 can get the same bandwidth before the arrival of F4. When F4 arrives, it can still get its fair share despite its less demand. This proves that DiffECN can provide fair share of bandwidth for flows regardless of their demand rates. Moreover, they can converge quickly whenever a new flow joins or leaves.

3) Incast Absorption: In the presence of an incast, the number of active flows passing through the switch link is likely to exceed the number of queues available on the port. To demonstrate the performance of DiffECN under bursty traffic, we add an additional 5% incast and create 60% network load in all-to-all communication. The incast level is 16-to-1 at



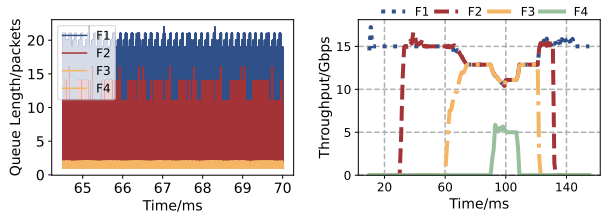
(a) Overall AVG (b) Large Flow AVG (>10M) (c) Small Flow AVG (<100K) (d) Small Flow 99th (<100K)

Fig. 15. [Simulation] FCT statistics with WebSearch workload.



(a) Overall AVG (b) Large Flow AVG (>10M) (c) Small Flow AVG (<100K) (d) Small Flow 99th (<100K)

Fig. 16. [Simulation] FCT statistics with DataMining workload.



(a) Queue length in preliminary test of § III-A. (b) Fairness among all flows.

Fig. 17. [Simulation] (a) Flows with higher rates have longer queues. (b) DiffECN provides fair share of bandwidth for each flow.

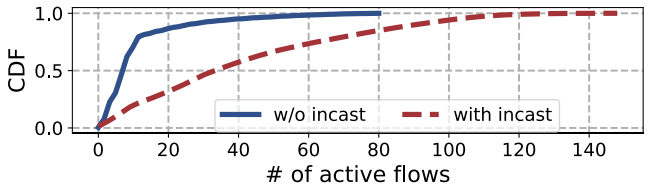
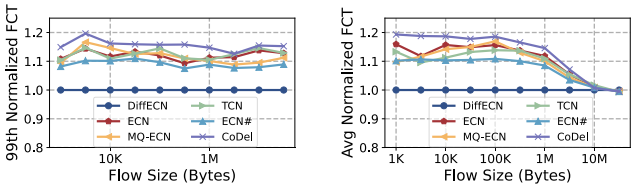


Fig. 18. [Simulation] The number of active flows.



(a) 99th Normalized FCT (b) Avg Normalized FCT

Fig. 19. [Simulation] DiffECN's incast tolerance.

each pod[3] where one of hosts in each pod suffers an incast. Hence for a leaf (ToR) switch within one pod, there will be at most 240 (16*15) connections created by incast, assuming they all arrive at the same time.

Figure 18 shows the distribution of active flows for a port in a leaf switch with and without incast. DataMining workload is used here. Under no incast, there are no more than 20 active flows for 90% of the time. Only in a few cases does the number of active flows exceed 32, supported by Tofino. When a large-scale incast comes, the number of active flows is increased. But it hardly exceeds 128, supported by Tofino2. Therefore, in practical production, we have the ability to provide a separate queue for each active flow.

In fact, even if the number of active flows exceeds the number of available queues due to bursts, DiffECN can still achieve good results. In our experiments, each switch port is assigned 32 queues. No separate queue can be provided for each flow in this case. Figure 19 shows FCT on different flow sizes. From the results, we can see that the 99th percentile FCT and the average FCT of DiffECN are the lowest. For large flows (>10MB), DiffECN can achieve similar performance to other schemes. This is mainly due to that when incast arrives, N_{flow} becomes large. According to Eq. (2), the marking threshold K_p decreases rapidly. In turn, it quickly curbs large flows passing through switches and keeps all queues' length low. Then even if a small flow randomly enters the queue where a large flow locates, there will be no HOL problem.

Through the above experiments and analysis, we learn that in real production, the switch has the ability to provide a separate queue for each active flow with the presence of incast. Even if there are not enough available queues to serve bursty traffic, DiffECN can still achieve good results by keeping the queue low.

4) Impact of Available Queues: In the previous experiments, we use 32 physical egress queues, supported by Tofino1. In production datacenters, it is possible that operators may reserve some queues for other services, resulting in less

[3]Datacenters build groups of servers in unit of pod. One pod has 16 servers in our all-to-all communication mode.

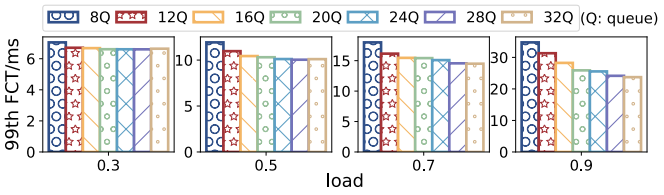


Fig. 20. [Simulation] Available queues.

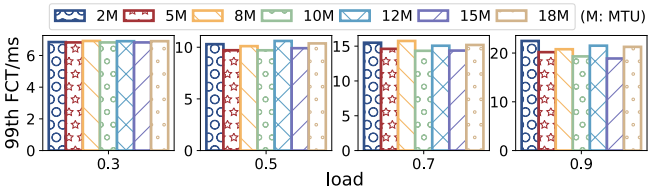
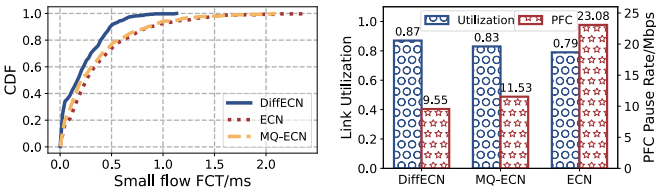
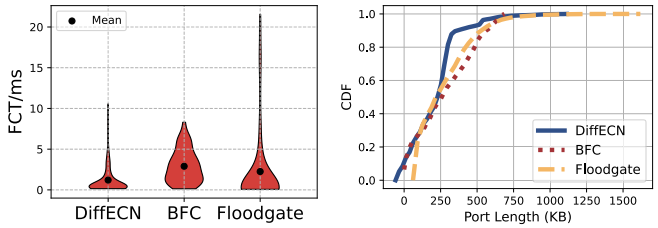


Fig. 21. [Simulation] "*Pass limit*" in the high-priority queue.



(a) CDF of FCT (b) Link Utilization and PFC PAUSE

Fig. 22. [Simulation] DCQCN with DiffECN.



(a) Overall FCT (b) CDF of queue length

Fig. 23. [Simulation] Comparison with BFC and Floodgate.

than 32 queues available. For example, some queues are used for traffic isolation. To explore the dependence of DiffECN on available queues, we change the number of egress queues in switches, from 8 to 32. Other experimental settings are consistent with all-to-all communication.

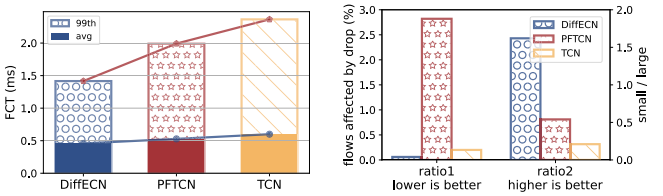
Fig. 20 shows that the more queues, the lower 99th percentile FCT of small flows is. More queues increase the probability of controlling each flow individually. Note that under the low load (like 0.3), the number of queues is sufficient to support active flows. Hence increasing more queues will not bring more gains. However, we can also see that after the number of queues exceeds 20, DiffECN can already achieve good performance, with only a slight difference from 32 queues'. This coincides with what is demonstrated in Figure 18, where the number of active flows in the switch port at the same moment is below 20 in most cases. We draw a conclusion that to achieve good performance, it is unnecessary for all queues to serve DiffECN, where additional queues can be reserved for other services.

As depicted in § IV-B, DiffECN sets a limit for each flow in the high-priority. For an active flow, when there is still no empty queue after the "probing" phase, it will be downgraded to the shortest queue due to the limit. Although it has been analyzed earlier that this situation rarely occurs, we still choose a better limit through experiments for robustness. We generate flows of various sizes based on WebSearch workload. Figure 21 illustrates the impact of the flow size limit. The results show that setting this limit to 10 MTU is a good choice. Small flows tend to enter non-empty queues when the number of bytes allowed per flow is small. So they can not finish quickly. If the value is large, the high-priority queue becomes busy. It will cause the rest queues to starve while large flows will take up more resources on the high-priority queue. In extreme workload scenarios, we can reduce the amount of time a flow spends in the high-priority queue and reduce the impact on the rest of the queue by controlling the size of the limit. So an appropriate limit in the high-priority queue can guarantee good performance for small flows, preventing the rest of the queue from starving.

5) DCQCN With DiffECN: In addition to DCTCP, DCQCN is also a commonly used congestion control algorithm based on ECN marking. We conduct case studies to show how DiffECN incorporates it. We construct a 40-to-1 topology with 40Gbps links, consisting of 40 senders and one receiver. The link between the switch and the receiver is the bottleneck with 60% load. The size of both large and small flows are generated based on WebSearch workload. In the switch, we adopt three marking policies: DiffECN, ECN, and MQ-ECN, respectively. The parameter settings are kept the same principle as the previous experiments. We measure the FCT of small flows (<100KB), bottleneck link utilization and PFC pause rate.

Figure 22(a) shows the FCT results of small flows, where DiffECN provides faster completion time. With DiffECN, all small flows are mostly completed within 1ms. Yet ECN and MQ-ECN have obvious long-tail effects, they provides enough buffer space for each flow, resulting in a long queue. Moreover, DiffECN provides priority service for small flows without assigned empty queues, which can cope with bursty traffic and trigger fewer pauses in PFC. From Figure 22(b), PFC pause rate in DiffECN is 9.55Mbps, lower than another two policies'. At the same time, triggering more PFC also prolongs the flow completion time, consistent with the results in the left subfigure. Under less PFC, DiffECN achieves higher link utilization because upstream senders would not be frequently paused. With the above analysis, DiffECN can still work well with the widely deployed DCQCN even in lossless networks.

With the above setup, we compare the state-of-art BFC [6] and Floodgate [51], where Floodgate also uses DCQCN as the host's control. Fig. 23(a) shows that DiffECN can provide lower average FCT (2.15ms), which is 33.6% lower than BFC (3.24ms) and 32.2% lower than Floodgate (3.17ms). BFC has the lowest tail latency and Floodgate has the largest tail latency. We carefully analyze the port buffer occupancy under the three schemes in Fig. 23(b). DiffECN provides a lower average port length and Floodgate has a longer length. This is because BFC sends PFC at per-queue granularity. Although it keeps the 99th length low, small flows are blocked by large flows, resulting in a higher overall average latency. Whereas Floodgate is based per-dst flow control, the port of



(a) Small flow FCT (b) Ratios of affected flows and buffer occupation

Fig. 24. [Simulation] Comparison with TCN.

the bottleneck link will have packet accumulation due to the large initial sending window. DiffECN minimizes harming small flows by marking large flows first. When the number of queues is not enough, it guarantees that the small flows finish quickly through the high-priority queue. Ultimately, DiffECN significantly improves the performance of small flows and the overall performance with slight harm on large flows.

6) In-Depth Comparison With TCN: In the original TCN [17], which is a port-level marking scheme, there is still the problem of large and small flows sharing the same congestion state. In order to extend the TCN, we explore its effect based on per-flow. Namely, whether each flow is marked or not does not depend on the queue length, but on the sojourn time of its packets in the switch. When sojourn time is greater than $\lambda * RTT$, we decide to mark the packet. We refer to the per-flow based TCN scheme as PFTCN for short. We add a 20-to-1 burst on the basis of the experimental setup of § VI-C.5. We measure the FCT, packet loss, and buffer occupancy.

Fig. 24(a) illustrates the average and 99th percentile FCT for small flows under the three schemes. DiffECNmaintains the lowest latency in both. Where the 99th percentile delay is 28.9% and 40.1% lower than PFTCN and TCN, respectively. Also, we see that PFTCN can improve the original TCN scheme by providing lower latency. We deeply analyze the reasons behind the above results in Fig. 24(b). Fig. 24(b) shows two ratios, where ratio1 refers to the percentage of flows affected by packet loss, and ratio2 refers to the ratio of the buffer occupancy of small flows to the buffer occupancy of large flows during burst. We find that PFTCN is more affected by packet loss, which is mainly due to the inability of PFTCN to curb congested flows in time. Although per-flow isolates the congestion state, sojourn time can only be detected when the packet dequeues. Take F2 in Fig. 5 as an example, F2 has already occupied more buffer at this point, but it is only marked as congested when its fourth packet dequeues. While DiffECNcan determine the congested flow in the first packet in time according to the queue length. Therefore less buffer is also provided to small flows under PFTCN, which puts pressure on the shallow buffer switch due to the overcommitment of large flows. This problem will be more significant in end-to-end transmission with longer RTT, which corresponds to a larger marking threshold and more lagged congestion response.

VII. Discussion, Limitations, and Future Work

A. Fairness Guarantee

According to § III-A, our goal is to slow down congested flows in time when congestion occurs through flow-distinguished marking while ensuring that the rest of traffic is not affected. In this way, each flow gets its fair share of bandwidth. In fact, the process of marking each flow according to its queue length is the inverse water-filling algorithm [37]. As we keep marking the flows with more buffer occupancy, the released bandwidth is gradually given to the "hungry" non-congested flows. Suppose there are three active flows competing for an outgoing port at the same time, and their bandwidth requirements are the same. Due to the difference in reach time, F2 occupies more buffer; F1 and F3 occupy less buffer. In this case, F2 is marked and its queue length gradually decreases until it reaches the fair queue length. F1 and F3, on the other hand, are not marked. So they can get more bandwidth and quickly increase the sending rate in a short time, with their queue length reaching the same as F1. The allocated bandwidth for high-speed flows gradually rises to a fair share, and the queue length gradually decreases to a fair level.

B. Count Active Flows Periodically

DiffECN only records flows that are currently in the queue, i.e., active flows. The state of a flow is updated for each cycle T_{period} (we have discussed parameter sensitivity about T_{period} in § VI-B.2). If a flow does not send packets consistently, its subsequent packets are likely to be treated as a new active flow, depending on the update period T_{period}. In other words, if the flow is stalled for a time greater than T_{period}, then its subsequent packets are treated as a new flow, and a new queue is reassigned. Otherwise, it is proved that the queue is still occupied by this flow and will not be allocated. Therefore, the probability of two active flows coexisting in the same queue is low.

C. Overhead in Design

In our approach, we need to consider the overhead from two operations. The first is the probing overhead described in § IV-B, where a packet from a flow in a high-priority queue triggers probing to see if there is an empty queue at that time before the flow's passed size exceeds the "*pass limit*." In the Tofino switch, we use ternary matching to find empty queues to avoid loop operations. The matching operation is natively supported by the data plane and does not affect the line-speed processing. Besides, we get the length of the queue by recirculating at ingress. Note that it will put pressure on the data plane. To minimize the overhead, in the future, we can update the queue length periodically, e.g. every few packets. But it will introduce a loss of precision, due to the infrequent updating. Thus we face a tradeoff between precision and overhead in the testbed implementation. Fortunately, Tofino2 has already implemented this feature [6]. It has an inbuilt feature tailored for fetching queue length in ingress and does not consume any additional ingress cycles or bandwidth. So we think this problem can be solved by upgrading the device.

D. Heavy Incast

Theoretically, active flows on a link will exceed the number of queues on the port under heavy incast. In this case, DiffECN will fall back to a modified per-queue scheme to ensure that no side effects occur. We have tested the performance

in a scenario with almost small flows, where the number of active flows exceeds the number of queues in Fig. 12. It shows that DiffECN can achieve almost similar performance with per-queue. In fact, under real workload and typical network bandwidth, it is often hardly possible for the active flows of a switch link to exceed the number of queues. Fig. 7(a) and 7(b) demonstrate that only in rare cases (like 90% load with 400G links) will the number of queues be exhausted. Moreover, according to [47], the average load of datacenters does not exceed 50%. Therefore, we believe that DiffECN can handle most cases when providing a queue for each active flow.

E. Universality

In § III, we use end-to-end transmission to illustrate the problems with traditional schemes. DiffECN seems to be more effective in long RTT scenarios because congested flows cannot be contained in time. However, congestions due to incast-like traffic patterns are frequent in datacenters, which can lead to longer total RTTs. In this case, innocent short flows still suffer long-term harm and DiffECN can alleviate the problems in traditional schemes.

VIII. Related Work

A. ECN-Based Transport in Datacenters

Generally, ECN-related works in datacenters are vast [3], [13], [17], [18], [19], [29], [12]. ECN* [29] is more sensitive than DCTCP and a lower ECN marking threshold can greatly affect the throughput. To adapt to the varying queue capacity caused by packet schedulers, TCN [17] and ECN# [18] are proposed to use instantaneous sojourn time. ACC [13] leverages deep reinforcement learning to adjust ECN parameters dynamically. These arts fall into the category of per-port marking and fail to guarantee fairness among flows passing through the same port. CoDel [20] and MQ-ECN [19] move one step forward by applying the per-queue marking. That is, setting separate ECN thresholds for each queue, in which multiple flows may still share the same threshold. Although FQ-CoDel [52] can do per-flow by creating 1024 egress queues, it has deployment challenges practically. Compared to per-queue marking, DiffECN can better handle bursty datacenter workloads by approximating the more fine-grained per-flow ECN marking.

B. Marking Threshold Settings in AQM

The most well-known rule of buffer sizing shows that the minimum marking threshold should be $C * RTT/\sqrt{N}$ when there are a large number of N long-lived TCP Reno flows [53]. The rule-of-thumb $C * RTT$ has been applied for regular TCP. Our DiffECN also follows this principle, except that we dynamically adjust the settings in proportion to the number of active flows. Ideal ECN-based datacenter congestion control algorithms [1], [29], [38] usually set the two thresholds of RED to the same value, i.e., $K_{min} = K_{max} = K = \lambda * C * RTT$. In addition, the sojourn time is also set, from the above rule, to determine if ECN marking is required sometime. The instantaneous sojourn time in TCN [17] and ECN# [18] are both set to $\lambda * RTT$.

IX. Conclusion

This paper aims at approximating per-flow marking for a bursty datacenter. We propose DiffECN based on the classical ECN scheme to mark flows differentially, guaranteeing timely rate curtailment for congested flows and deserved bandwidth for non-congested ones. Extensive experiments have validated its performance and burst tolerance. With the valid implementation and without any changes at the end-host, we envision that DiffECN has the potential to be deployed in modern production datacenters.

References

[1] M. Alizadeh et al., "Data center TCP (DCTCP)," in *Proc. ACM SIGCOMM Conf.*, Aug. 2010, pp. 63–74.

[2] R. Nishtala et al., "Scaling memcache at Facebook," in *Proc. 10th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2013, pp. 385–398.

[3] Y. Zhu et al., "Congestion control for large-scale RDMA deployments," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 523–536, Aug. 2015.

[4] Y. Li et al., "HPCC: High precision congestion control," in *Proc. ACM Special Interest Group Data Commun.*, Aug. 2019, pp. 44–58.

[5] R. Mittal et al., "TIMELY: RTT-based congestion control for the datacenter," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 537–550, 2015.

[6] P. Goyal, P. Shah, K. Zhao, G. Nikolaidis, M. Alizadeh, and T. E. Anderson, "Backpressure flow control," in *Proc. 19th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2022, pp. 779–805.

[7] G. Kumar et al., "Swift: Delay is simple and effective for congestion control in the datacenter," in *Proc. ACM SIGCOMM Conf. Appl. Technol. Archit. Protocols Comput. Commun. (SIGCOMM)*, 2020, pp. 514–528.

[8] M. Alizadeh et al., "pFabric: Minimal near-optimal datacenter transport," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 435–446, 2013.

[9] M. Al-Fares, "Change management in physical network lifecycle automation," in *Proc. USENIX ATC*, 2023, pp. 635–653.

[10] B. Cronkite-Ratcliff et al., "Virtualized congestion control," in *Proc. ACM SIGCOMM Conf. (SIGCOMM)*, 2016, pp. 230–243.

[11] K. He et al., "AC/DC TCP: Virtual congestion control enforcement for Datacenter networks," in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 244–257.

[12] W. Bai, S. Hu, K. Chen, K. Tan, and Y. Xiong, "One more config is enough: Saving (DC)TCP for high-speed extremely shallow-buffered datacenters," *IEEE/ACM Trans. Netw.*, vol. 29, no. 2, pp. 489–502, Apr. 2021.

[13] S. Yan, X. Wang, X. Zheng, Y. Xia, D. Liu, and W. Deng, "ACC: Automatic ECN tuning for high-speed datacenter networks," in *Proc. ACM SIGCOMM Conf. Appl. Technol. Archit. Protocols Comput. Commun. (SIGCOMM)*, 2021, pp. 384–397.

[14] Y. Zhang, Y. Liu, Q. Meng, and F. Ren, "Congestion detection in lossless networks," in *Proc. ACM SIGCOMM Conf.*, Aug. 2021, pp. 370–383.

[15] K. K. Ramakrishnan and R. Jain, "A binary feedback scheme for congestion avoidance in computer networks," *ACM Trans. Comput. Syst.*, vol. 8, no. 2, pp. 158–181, May 1990.

[16] S. Floyd, D. K. K. Ramakrishnan, and D. L. Black, *The Addition of Explicit Congestion Notification (ECN) to IP*, document RFC 3168, Internet Requests for Comments, Internet Engineering Task Force, Sep. 2001. [Online]. Available: https://www.rfc-editor.org/info/rfc3168

[17] W. Bai, K. Chen, L. Chen, C. Kim, and H. Wu, "Enabling ECN over generic packet scheduling," in *Proc. 12th Int. Conf. Emerg. Netw. Exp. Technol.*, Dec. 2016, pp. 191–204.

[18] J. Zhang, W. Bai, and K. Chen, "Enabling ECN for datacenter networks with RTT variations," in *Proc. 15th Int. Conf. Emerg. Netw. Exp. Technol.*, Dec. 2019, pp. 233–245.

[19] W. Bai, L. Chen, K. Chen, and H. Wu, "Enabling ECN in multi-service multi-queue data centers," in *Proc. 13th USENIX Symp. Netw. Syst. Design Implement.*, 2016, pp. 537–549.

[20] K. Nichols and V. Jacobson, "Controlling queue delay," *Commun. ACM*, vol. 55, no. 7, pp. 42–50, 2012.

[21] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 50–61, 2011.

[22] (2023). *Intel Tofino*. [Online]. Available: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html

[23] (2023). *Intel Tofino2*. [Online]. Available: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html

[24] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. 10th ACM SIGCOMM Conf. Internet Meas.*, Nov. 2010, pp. 267–280.

[25] S. Ibanez, G. Antichi, G. Brebner, and N. McKeown, "Event-driven packet processing," in *Proc. 18th ACM Workshop Hot Topics Netw.*, Nov. 2019, pp. 133–140.

[26] (2023). *Network Simulator 3*. [Online]. Available: https://www.nsnam.org/

[27] (2010). *IEEE 802.1 QAU—Congestion Notification*. [Online]. Available: http://www.ieee802.org/1/pages/802.1au.html

[28] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Trans. Netw.*, vol. 1, no. 4, pp. 397–413, Aug. 1993.

[29] H. Wu, J. Ju, G. Lu, C. Guo, Y. Xiong, and Y. Zhang, "Tuning ECN for data center networks," in *Proc. 8th Int. Conf. Emerg. Netw. Experiments Technol.*, Dec. 2012, pp. 25–36.

[30] M. Alizadeh, A. Javanmard, and B. Prabhakar, "Analysis of DCTCP: Stability, convergence, and fairness," in *Proc. ACM SIGMETRICS*, 2011, pp. 73–84.

[31] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 63–74, 2008.

[32] M. Alizadeh et al., "CONGA: Distributed congestion-aware load balancing for datacenters," in *Proc. ACM Conf. SIGCOMM*, Aug. 2014, pp. 503–514.

[33] V. Addanki, O. Michel, and S. Schmid, "PowerTCP: Pushing the performance limits of datacenter networks," in *Proc. 19th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2022, pp. 51–70.

[34] J.-Y. L. Boudec. (Nov. 2021). *Rate Adaptation Congestion Control and Fairness: A Tutorial*. [Online]. Available: https://leboudec.github.io/leboudec/resources/tutorial.html

[35] D. Shan, F. Ren, P. Cheng, R. Shu, and C. Guo, "Micro-burst in data centers: Observations, analysis, and mitigations," in *Proc. IEEE 26th Int. Conf. Netw. Protocols (ICNP)*, Sep. 2018, pp. 88–98.

[36] J.-Y. Le Boudec, "Rate adaptation, congestion control and fairness: A tutorial," *Web Page*, vol. 4, pp. 1–54, Nov. 2005.

[37] E. Danna et al., "Upward max min fairness," in *Proc. IEEE INFOCOM*, Mar. 2012, pp. 837–845.

[38] B. Vamanan, J. Hasan, and T. N. Vijaykumar, "Deadline-aware datacenter TCP (D2TCP)," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 115–126, 2012.

[39] M. Shreedhar and G. Varghese, "Efficient fair queueing using deficit round Robin," in *Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, Oct. 1995, pp. 231–242.

[40] (2023). *P4 Open Source Programming Language*. [Online]. Available: https://p4.org

[41] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized 'zero-queue' datacenter network," in *Proc. ACM SIGCOMM*, 2014, pp. 307–318.2023.

[42] A. Tharwat, "Classification assessment methods," *Appl. Comput. Informat.*, vol. 17, no. 1, pp. 168–192, Jan. 2021.

[43] A. Greenberg et al., "VL2: A scalable and flexible data center network," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 51–62, Aug. 2009.

[44] (2015). *Empirical Traffic Generator*. [Online]. Available: https://github.com/datacenter/empi rical-traffic-gen

[45] V. Paxson and S. Floyd, "Wide area traffic: The failure of Poisson modeling," *IEEE/ACM Trans. Netw.*, vol. 3, no. 3, pp. 226–244, Jun. 1995.

[46] M. S. Taqqu, W. Willinger, and R. Sherman, "Proof of a fundamental result in self-similar traffic modeling," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 27, no. 2, pp. 5–23, Apr. 1997.

[47] A. Roy, Z. Hongyi, B. Jasmeet, P. George, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proc. ACM SIGCOMM*, 2015, pp. 123–137.2015.

[48] H. Zhang, J. Zhang, W. Bai, K. Chen, and M. Chowdhury, "Resilient datacenter load balancing in the wild," in *Proc. ACM SIGCOMM*, Aug. 2017, pp. 253–266.

[49] Intel. (Apr. 2021). *P416 Intel Tofino Native Architecture—Public Version*. [Online]. Available: https://raw.githubusercontent.com/barefootnetworks/open-tofino/master/public-native-arch.pdf

[50] C. Gao, S. Chu, H. Xu, M. Xu, K. Ye, and C. Xu, "Flash: Joint flow scheduling and congestion control in data center networks," *IEEE Trans. Cloud Comput.*, vol. 11, no. 1, pp. 1038–1049, Jan. 2023.

[51] K. Liu et al., "Floodgate: Taming incast in datacenter networks," in *Proc. 17th Int. Conf. Emerg. Netw. Exp. Technol.*, Dec. 2021, pp. 30–44.

[52] T. Hoiland-Jorgensen, P. McKenney, T. Dave, J. Gettys, and E. Dumazet, *The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm*, document RFC 8290, Internet Requests for Comments, Internet Engineering Task Force, Jan. 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8290

[53] G. Appenzeller, I. Keslassy, and N. McKeown, "Sizing router buffers," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 4, pp. 281–292, Aug. 2004.

Hanlin Huang (Graduate Student Member, IEEE) received the B.E. degree from the Software College, Nankai University, Tianjin, China, in 2021. He is currently pursuing the Ph.D. degree with Tsinghua University. His research interests include data center networks and AQM.

Ke Xu (Fellow, IEEE) received the Ph.D. degree from Tsinghua University, Beijing, China. He is currently a Full Professor with the Department of Computer Science, Tsinghua University. He has published more than 200 technical articles and holds 11 U.S. patents in the research areas of next-generation internet, blockchain systems, the Internet of Things, and network security. He is an ACM Member. He serves as the Steering Committee Chair for IEEE/ACM IWQoS. He has guest-edited several special issues for IEEE and Springer journals. He is an Editor of IEEE INTERNET OF THINGS JOURNAL.

Tong Li (Member, IEEE) received the B.E. degree from the School of Computer Science, Wuhan University, China, in 2012, and the Ph.D. degree from the Department of Computer Science and Technology, Tsinghua University, China, in 2017. He was a Visiting Scholar with the School of Computer Science and Electronic Engineering, University of Essex, U.K., in 2014 and 2016. He was a Chief Engineer with Huawei in 2022. He is currently an Associate Professor with Renmin University of China. His research interests include networking, distributed systems, and big data.

Zhuotao Liu received the Ph.D. degree from the University of Illinois at Urbana–Champaign, USA. He is currently an Associate Professor with Tsinghua University. Before joining Tsinghua University, he was a Technical Lead with Google, managing one of the world's largest software-defined datacenter networks. His research interests include data/AI security and privacy, blockchain and applied cryptography, and secure internet architecture.

Xinle Du received the B.E. degree from the Department of Computer Science and Technology, Xidian University, Xi'an, China, in 2018, and the Ph.D. degree from the Department of Computer Science and Technology, Tsinghua University, Beijing, China, in 2023. He has been the Chief Engineer with the Computer Network and Protocol Laboratory, Huawei Technologies, since 2023. His research interests include networking and LLM systems.

Xiangyu Gao received the B.S. degree from the Taishan College, Shandong University, Qingdao, China, in 2021. He is currently pursuing the Ph.D. degree with Tsinghua University, Beijing, China. His research interests include data center networking and programmable dataplane.