Enabling ECN for Datacenter Networks with RTT Variations

Junxue Zhang[1], Wei Bai[2], Kai Chen[1,3]
[1]SING Lab, Hong Kong University of Science and Technology
[2]Microsoft Research [3]Peng Cheng Laboratory

ABSTRACT

ECN has been widely employed in production datacenters to deliver high throughput low latency communications. Despite being successful, prior ECN-based transports have an important drawback: they adopt a fixed RTT value in calculating instantaneous ECN marking threshold while overlooking the RTT variations in practice.

In this paper, we reveal that the current practice of using a fixed high-percentile RTT for ECN threshold calculation can lead to persistent queue buildups, significantly increasing packet latency. On the other hand, directly adopting lower percentile RTTs results in throughput degradation. To handle the problem, we introduce ECN$^\sharp$, a simple yet effective solution to enable ECN for RTT variations. At its heart, ECN$^\sharp$ inherits the current instantaneous ECN marking (based on a high-percentile RTT) to achieve high throughput and burst tolerance, while further marking packets (conservatively) upon detecting long-term queue buildups to eliminate unnecessary queueing delay without degrading throughput. We implement ECN$^\sharp$ on a Barefoot Tofino switch and evaluate it through extensive testbed experiments and large-scale simulations. Our evaluation confirms that ECN$^\sharp$ can effectively reduce latency without hurting throughput. For example, compared to the current practice, ECN$^\sharp$ achieves up to 23.4% (31.2%) lower average (99th percentile) flow completion time (FCT) for short flows while delivering similar FCT for large flows under production workloads.

CCS CONCEPTS

• **Networks → Data center networks**.

KEYWORDS

Datacenters, ECN, RTT Variations, AQM

ACM Reference Format:
Junxue Zhang, Wei Bai, and Kai Chen. 2019. Enabling ECN for Datacenter Networks with RTT Variations. In *The 15th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '19), December 9–12, 2019, Orlando, FL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3359989.3365426

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CoNEXT '19, December 9–12, 2019, Orlando, FL, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6998-5/19/12…$15.00
https://doi.org/10.1145/3359989.3365426

1 INTRODUCTION

Datacenter applications desire high throughput (e.g., data mining and storage) and low latency (e.g., web search and memory cache) communications. To achieve this, Explicit Congestion Notification (ECN) has been widely used, and many ECN-based transports, such as DCTCP [12] and DCQCN [39], have been proposed and adopted by industry [1, 2, 28, 35].

ECN-based transports require both ECN-aware rate control at the end host and ECN marking at the switch. ECN marking is accomplished by an active queue management (AQM) policy. To handle transient bursts which are common and harmful in production environments [38], datacenter AQM solutions usually mark packets aggressively based on instantaneous congestion states. For example, DCTCP [12] modifies the original RED[1] [20] to leverage the instantaneous queue length to mark packets, thus achieving good burst tolerance.

For instantaneous ECN marking, the choice of the marking threshold is critical as it directly affects the tradeoff between throughput and latency [12, 37]. Despite being successful, prior ECN-based transports have shared an important drawback: they only adopt a fixed RTT[2] value in calculating the marking threshold without considering RTT variations.

However, large RTT variations are common in datacenters as different flows traverse different processing components, e.g., network stack, hypervisor and middlebox. For example, compared to intra-service flows, inter-service flows experience large extra processing delays from the layer 4 load balancer [19, 33]. Furthermore, the processing delay of a given component also varies depending on workloads. With a very simple testbed (§2.2), we show these factors can easily lead to ~3× RTT variations, which also implies even larger RTT variations in real production.

We point out, through testbed experiments, that the current practice of using a fixed high-percentile RTT for ECN threshold calculation can lead to persistent queue buildup, significantly increasing packet latency. For example, our experiment results show that using 90th percentile RTT (as suggested by [37]) can achieve high throughput, but the latency of short flows is increased by over 50% due to queueing delay (§2.3). On the other hand, directly adopting lower percentile RTTs does not solve the problem either, as it results in throughput degradation.

Motivated by this, we seek an ECN marking scheme that can deliver high throughput and low latency simultaneously, in the presence of high RTT variations. To this end, we present ECN$^\sharp$ (§3), a simple yet effective solution to achieve our goal. At its heart, ECN$^\sharp$ marks packets based on both the instantaneous and persistent congestion states. On the one hand, ECN$^\sharp$ inherits the advantage of

[1]We call this version of RED as DCTCP-RED in this paper.
[2]In this paper, we use RTT to denote the base RTT without the queueing delay in datacenters.

current instantaneous ECN marking based on a high-percentile RTT to achieve high throughput and burst tolerance. On the other hand, to eliminate the persistent queueing caused by flows with small RTTs, ECN$^\sharp$ further marks packets when observing long-term switch queue buildups. This is performed conservatively in order not to affect throughput (§3.2).

By nature, ECN$^\sharp$ works with both queue length and sojourn time (i.e., the amount of time a packet spends in the queue) as congestion signals. In our implementation we adopt sojourn time in order to be compatible with packet scheduler with traffic dynamics [16, 31].

We implement an ECN$^\sharp$ prototype with ~500 lines of P4 code and compile it to a Barefoot Tofino [5] switch using Barefoot Capilano SDE [9] (§4). Our implementation uses little switch resource, e.g., 5 32-bit register arrays and 2 64-bit register arrays. We address two practical challenges in the implementation: 1) To ensure the correctness, we use emulation to get a 32-bit microsecond-granularity system time; 2) To update switch states at line rate, we leverage match action tables to implement the complex control flow.

We build a small 10Gbps testbed with 8 servers connected to the above Tofino switch. Our experiments with realistic workloads [12, 22] show that, by eliminating persistent queue buildups, ECN$^\sharp$ can deliver up to 23.4% lower average FCT and 31.2% lower 99th percentile FCT for short flows compared to DCTCP-RED (where the ECN marking threshold is based on a high-percentile RTT according to current practice) while still maintaining comparable throughput for large flows. Furthermore, we find that instantaneous ECN marking is integral to ECN$^\sharp$ to tolerate bursts and reduce packet drops. For example, ECN$^\sharp$ outperforms CoDel [31] (which marks packets only based on persistent queue buildups instead of instantaneous queueing) by more than 50.0% FCT reduction for short flows.

To complement small-scale testbed experiments, we further perform larger-scale simulations to deep-dive into ECN$^\sharp$ (§5.3). Our simulation results further confirm the superior performance of ECN$^\sharp$. For example, compared to DCTCP-RED, ECN$^\sharp$ can achieve up to 37.9% lower FCT for short flows. Furthermore, from the microscopic view of switch queues, we show ECN$^\sharp$ can effectively eliminate queue buildups by keeping the switch queue length 95.6% lower than that of DCTCP-RED (from 182 packets to 8 packets). Finally, we show ECN$^\sharp$ is robust to parameter settings and arbitrary packet schedulers.

To make our work easy to reproduce, we have made our simulation code available at https://github.com/snowzjx/ns3-ecn-sharp.

2 BACKGROUND AND PROBLEMS

2.1 Instantaneous ECN Marking

ECN-based transports consist of two parts: ECN marking at the switch and ECN-aware rate control at the end host. ECN marking is accomplished by an active queue management (AQM) policy. To handle transient bursts which are common in production datacenters [38], most datacenter AQM solutions [12, 16, 39] aggressively mark packets based on the instantaneous congestion states, e.g., instantaneous queue length.

For instantaneous ECN marking, the choice of the marking threshold is important as it directly affects the tradeoff between throughput and latency [12, 37]. Most ECN-based datacenter congestion control algorithms [12, 36, 37] set the two thresholds of RED to the same value, $K_{min} = K_{max} = K$. Given the low statistical multiplexing of large flows in datacenter environments [12], to fully utilize link bandwidth while delivering low latency, the ideal ECN marking threshold K is given as follows [12, 18, 37]:

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

To calculate and configure the ideal ECN marking threshold at the switch, datacenter operators must get values for three parameters: λ, C and RTT. λ is a parameter determined by the congestion control algorithm at the end host. Different congestion control algorithms have different ECN reaction mechanisms, resulting in different λ. For example, ECN* (regular ECN-enabled TCP) cuts the window by half in the presence of ECN marks, thus having $\lambda = 1$. DCTCP reduces the window in proportion to the fraction of ECN marked packets, thus having a much smaller λ (0.17 in theory [13]). C is the bottleneck link capacity, which can be easily obtained for threshold calculation as datacenter operators have full knowledge of the network.

RTT is the base round-trip time and is the focus of this paper. Here the base RTT does not include switch queueing delay. In current practice, people use a fixed RTT value for calculation, implicitly assuming that base RTTs in datacenters are relatively stable due to the small cable length (low propagation delay variations), e.g., 200-300 meters [23], and high link capacity (low transmission delay variations). However, we show this does not hold in reality (§2.2) and causes problems (§2.3).

2.2 RTT Variations in Datacenters

The base RTT consists of three parts: transmission delay, propagation delay, and processing delay. As discussed above, transmission delay and propagation delay are small inside datacenters. The transmission time of a 1.5KB packet on a 10Gbps link is only ~1.2μs. The propagation delay of a 1KM cable is only 3.3μs. However, the processing delay can be as high as tens (or even hundreds) of μs, dominating the base RTT. Thus, the variation of the base RTT is actually caused by the variation of processing delay.

In fact, the processing delay in datacenters varies vastly as different flows traverse different processing components, e.g., network stack, hypervisor, and middlebox. The more processing components a flow traverse, the larger delay it experiences. For example, in our production datacenters, we use a layer-4 software load balancer (SLB) [33] to process and balance the inter-service traffic (e.g., traffic from compute to storage)[3]. Inter-service *inbound* traffic is first delivered to the SLB Multiplexer (a scalable set of dedicated servers), then forwarded to the application server, thus experiencing extra processing delay compared to intra-service traffic. Furthermore, the processing delay of a given component also varies depending on the loads.

We use testbed experiments to demonstrate the variations of processing delay. In our testbed, 3 hosts are connected to a Mellanox SN2100 switch via 100 Gbps links. Each server is equipped

[3]A large portion of inter-service traffic stays within the same datacenter [33].

RIGHTSLINK

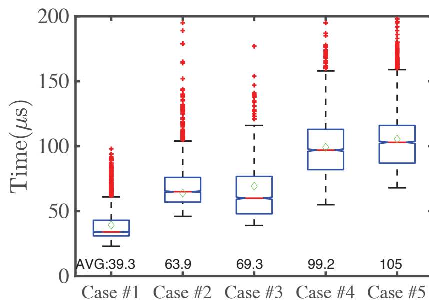


Figure 1: [Testbed] RTT variations. Varying processing delay has enlarged RTT variations up to 2.68 times.

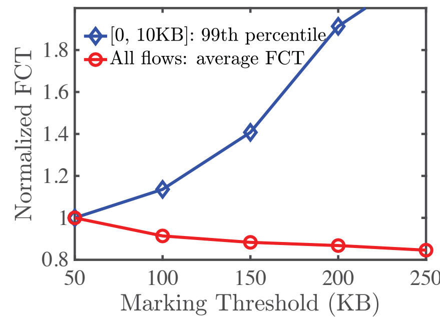


Figure 2: [Testbed] Instantaneous marking cannot achieve high throughput and low latency simultaneously.

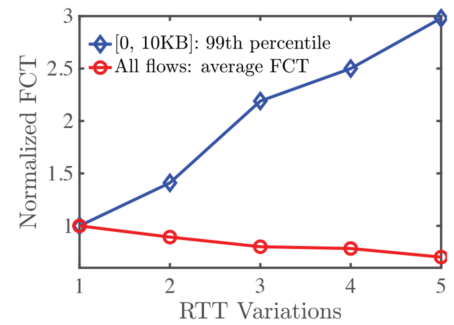


Figure 3: [Testbed] Larger RTT variations cause more degradation to performance.

with a 100 Gigabit Ethernet Adapter. We enable DCTCP [12] at the end host. A host installed with Apache HTTP Server works as the web server. We generate HTTP requests using ApacheBench on another host to fetch a 1-byte web page. On the client host, we install a modified TCP Probe to record smoothed RTTs (srtt) measured by Linux kernel. For each experiment, we generate 1000 requests and get ~3000 RTT samples. Note that a new request is sent when we receive the previous response. Since there is no congestion and hop difference, the RTT variation is caused by processing delay. We consider the following five cases and show corresponding results in Figure 1 and Table 1.

1. Network Stack: In this experiment, the requests from the client are directly sent to the server. Therefore, processing delay mainly comes from network stacks of the client and server. As shown in Table 1, the average RTT and standard deviation are 39.3μs and 12.2μs, respectively. We think the variation is due to multiple factors, e.g., kernel scheduling, advanced NIC techniques such as TSO, GSO, GRO and TCP delayed ACK mechanism.

2. Network Stack + SLB: In this experiment, we add a layer 4 SLB between the client and the server. SLB is widely used to process inter-service traffic inside the datacenters [33]. Our SLB is another host installed with Linux Virtual Server (LVS)[11]. The requests are first sent to the SLB. Then SLB forwards them to the backend server. The responses are directly returned to the client, without traversing the SLB. With SLB, the average RTT and standard deviation are increased to 63.9μs (1.62X) and 18.3μs, respectively. The processing latency imposed by the SLB is ~24.6μs.

3. Network Stack + Hypervisor: In this experiment, we use a quad-core virtual machine (VM) as the web server. Therefore, compared to the first experiment, request and response packets experience extra processing latencies on the hypervisor. On the hypervisor, we only add a necessary bridge rule without installing other complex rules, which is much simpler than the configuration in production environments. The average RTT and standard deviation are 69.3μs (1.76X) and 18.8μs, respectively. The extra latency caused by the hypervisor is ~30μs.

4. Network Stack + SLB + Hypervisor: In this experiment, we combine the above two experiments. With the SLB and hypervisor, the average RTT is increased from 39.3μs to 99.2μs (2.52X), compared to the first one.

5. Network Stack (High Load) + SLB + Hypervisor: In this experiment, to emulate the high load, we use stress to run 4 workers spinning on `sqrt()` on the web server VM. As a result, the average RTT reaches 105.5μs, which is around 2.68X larger than that in the first experiment.

Summary: We use the above simple experiments to demonstrate the causes of RTT variations in datacenters. Network components, such as Network Stack, SLB, Hypervisor, and etc., add varying and unpredictable variations to the RTT, especially when they are under different loads. Furthermore, different flows may traverse different numbers of components, making RTT variations more severe. Note that all of the experiments are conducted in very simple settings, e.g., only a single bridge rule in the hypervisor, and very low loads (except for the last one). We believe that the actual RTT variations in production environment would be much larger due to complex configurations, high system loads, and more processing components. For example, Rohan Gandhi et al. [21] showed that the SLB in Microsoft datacenter adds a median latency of 196us to packets while the 90th percentile can reach as high as 1ms. Moreover, the base RTT varies across both flows and time. As the last two experiments show, the base RTT of a given flow also varies when the load of a processing component changes.

2.3 Current Practice and Problems

Existing solutions [12, 16, 18, 37] only adopt a fixed RTT value for ECN marking threshold calculation while overlooking RTT variations, which can cause severe problems. For example, operators get RTT distributions using tools such as PingMesh [24] and in current practice they use a high percentile RTT (e.g., 90th percentile [37]) to derive the threshold. While this approach can achieve high throughput for flows with different RTTs in general, for flows with small RTTs, it results in persistent queue buildups, causing queueing delay. On the other hand, directly using low percentile RTTs (e.g., 50%) or average RTT can mitigate such queueing

RIGHTSLINK

Combination of different processing components	Mean (μs)	STD (μs)	90 percentile(μs)	99 percentile(μs)
Networking Stack	39.3	12.2	59.0	79.0
Networking Stack + SLB	63.9	18.3	87.0	121.0
Networking Stack + Hypervisor	69.3	18.8	91.0	130.0
Networking Stack + SLB + Hypervisor	99.2	23.0	129.0	161.0
Networking Stack(high load) + SLB + Hypervisor	105.5	23.6	138.0	178.0

Table 1: [Testbed] RTT statistics. Different combinations of processing components result in up to 2.68× RTT variations in a very simple testbed setting, implying that the actual RTT variations in production datacenters would be even larger.

latency, but cause flows with high RTTs to suffer from throughput degradation.

Since we only have three 100Gbps adapters, we build a new testbed to demonstrate this dilemma. We use 8 endhosts with 10Gbps Ethernet adapters, connecting to the Mellanox SN2100 switch. The cable speed is 10Gbps (with QSFP+ to SFP+ Adapter). We generate traffic according to the web search workload [12] (Figure 5). The average load of the bottleneck link is 50%. To emulate RTT variations, we use `netem` [6]. `netem` is a Linux traffic control component which can emulate increased delay, packet loss, duplication, and etc.. It allows adding delays in microseconds. We use `netem` to add extra delay at senders to emulate RTT variations.

Assume the minimum RTT and maximum RTT within the network are RTT_{min} and RTT_{max} respectively, we define RTT variation as RTT_{max}/RTT_{min} here.

Observation 1: Instantaneous ECN marking cannot achieve high throughput and low latency simultaneously. First, we let the RTT variation to be 3× (70μs to 210μs[4]) and choose different ECN marking thresholds. We change the marking threshold from 50KB to 250KB. The FCT statistics are shown in Figure 2. The results have been normalized to the FCT achieved by the threshold 50KB. If we choose high percentile RTTs to derive the threshold, such as $90th$ percentile (250KB) according to the current practice, short flows suffer from 119.2% longer 99th percentile FCT (581μs to 265μs). On the contrary, setting threshold based on low percentile RTTs leads to throughput loss. For example, using average RTT ($\sim$ 100KB) causes 8% throughput degradation (3701μs to 3426μs). Thus, none of the values in the range can simultaneously achieve high throughput and low latency.

Observation 2: Larger RTT variations enlarge performance loss. Second, we try to understand how the RTT variations affect the performance. We change the RTT variation from 2× (70μs to 140μ) to 5× (70μs to 350μ) and set the ECN marking threshold based on the average and 90th percentile RTTs accordingly. The FCT statistics are shown in Figure 3, normalized to the result achieved by average RTT. When the variation is 2×, the threshold with the average RTT leads to 6.7% throughput loss (3375μs to 3163μs) compared to that with the 90th percentile. The gap increases to 29.8% (4464μs to 3439μs) when the RTT variation reaches 5×. Similarly, for 99th percentile FCT of short flows, the performance degradation caused by current practice has increased from 41.1% (404μs to 287μs) to 198.0% (995μs to 334μs) when the RTT variations rise from 2× to 5×. Thus, larger RTT variation has enlarged the performance degradation.

Summary: Instantaneous ECN marking cannot deliver both high throughput and low queueing delay under RTT variations. The performance degrades more as the RTT variation grows.

[4] The emulated RTT denotes end-to-end base RTT.

3 ECN$^\sharp$

3.1 Design Choice

Given the above dilemma, we seek to develop a solution to achieve high throughput and low queueing delay in datacenters even with high RTT variations. In the meanwhile, the solution should be resilient to traffic burstiness.

As shown in §2.3, the crux of existing instantaneous ECN marking is that the configured static threshold cannot adapt to the base RTTs of active flows. Given the emerging reconfigurable switching chips [5], a straightforward solution is a radical switch AQM that tracks the base RTTs of active flows (both spatially and temporally) and dynamically adjusts the instantaneous threshold on each individual switch correspondingly. For example, we can use the average base RTTs of all active flows to compute the threshold in Equation 1. However, it is challenging to measure real-time base RTTs. The reason is that, to exclude the noise of queueing delay, we first need to measure real-time processing delay of each component in both directions and embed this information into packet headers. This is a huge burden for operators as they need to modify all network processing components in datacenters.

Therefore, we take one step back and seek a lightweight solution. Our lesson learned from §2.3 shows that the current practice of using a high percentile RTT to derive the instantaneous threshold is able to achieve high throughput and burst tolerance, but suffers from persistent queues when the base RTTs of active flows are small. Inspired by this, our design choice is to inherit the current practice for throughput and burst tolerance, and augment it with a scheme to detect and eliminate persistent queues. As a result, we introduce ECN$^\sharp$: 1) inherits instantaneous ECN marking based on a high percentile RTT to handle bursts and maintain high throughput; and 2) further enables ECN marking upon presistent queue buildups to eliminate unnecessary queueing delay without incurring throughput loss.

RIGHTSLINK

Configuration Parameters	
pst_target	persistent queueing target
pst_interval	interval to determine persistent queueing
ins_target	threshold for instantaneous marking
Variables	
sojourn_time	sojourn time of the current packet
detected	if persistent queueing is detected
marking_state	if ECN♯ has marked packets
marking_count	number of marked packets
marking_next	the next marking time
first_above_time	when sojourn time starts to exceeds *pst_target*

Table 2: Parameters and variables used in ECN♯.

3.2 ECN♯ Mechanism

ECN♯ is a new AQM solution. Following the above rationale, it marks packets based on both instantaneous and persistent congestion states. A packet is marked when either one of the following conditions decides to mark it:

- When there is a large instantaneous queue, ECN♯ *aggressively* marks packets to avoid buffer overflow.
- When there is a persistent queue, ECN♯ *conservatively* marks packets to reduce queueing delay.

By nature, ECN♯ works with both queue length and sojourn time (i.e., the amount of time a packet spends in the queue) as congestion signals. According to Equation 1, the equivalent sojourn time based ECN marking threshold is:

$$T = \frac{K}{C} = \frac{\lambda \times C \times RTT}{C} = \lambda \times RTT \quad (2)$$

In our design and implementation, we adopt sojourn time over queue length in order to be compatible with packet scheduler with traffic dynamics [16, 31].

ECN marking based on instantaneous queue: To achieve burst tolerance, ECN♯ employs instantaneous ECN marking. When a packet dequeues, ECN♯ compares the sojourn time of the packet with the instantaneous marking threshold, $T = ins_target$, which can be calculated based on Equation 2 with high percentile RTTs. If the sojourn time exceeds the threshold, the packet gets ECN marked.

ECN marking based on persistent queue buildups: First, ECN♯ tracks the minimal queueing over an interval to detect persistent queue buildups. Then, it performs conservative marking to eliminates those queues. Here, two parameters are involved: 1) *pst_interval*: the time interval used to observe the queue before deciding if there is persistent queueing; 2) *pst_target*: the persistent queue threshold used to compare against the packet sojourn time to control the long-term queueing. The algorithm mainly contains the following two parts as illustrated in Algorithm 1. Table 2 shows all parameters and variables.

Persistent queue buildups detection. ECN♯ measures whether the sojourn time has exceeded *pst_target* for at least one *pst_interval* to determine long-term queueing. ECN♯ only uses one variable

Algorithm 1: ECN marking upon persistent congestion

```
   input  : Coming packet pkt
   output: Whether the pkt should be marked
1  Procudure ShouldPersistentMark(pkt)
2      detected = IsPersistentQueueBuildups(pkt)
3      if marking_state == true then
4          if detected == false then
5              marking_state = false
6              return false
7          else if now() > marking_next then
8              marking_count ++
9              marking_next += pst_interval / sqrt(marking_count)
10             return true
11         end
12     else
13         if detected == true then
14             marking_state = true
15             marking_count = 1
16             marking_next = now() + pst_interval
17             return true
18         end
19         return false
20     end
21 Procudure IsPersistentQueueBuildups(pkt)
22     if pkt.sojourn_time < pst_target then
23         first_above_time = 0
24         return false
25     end
26     if first_above_time == 0 then
27         first_above_time = now()
28         return false
29     else if now() > first_above_time + pst_interval then
30         return true
31     else
32         return false
33     end
```

first_above_time to record the timestamp that sojourn time exceeds *pst_target* for the first time (first_above_time = `now()`). If sojourn time is lower than *pst_target*, which means the queue expires, ECN♯ will reset *first_above_time* to 0. If ECN♯ finds it has been over one *pst_interval* since *first_above_time* (`now()` > first_above_time + *pst_interval*), it confirms there is persistent queueing and triggers the conservative marking logic.

Conservative marking. After detecting the persistent queue buildups, ECN♯ conservatively marks one packet in each *pst_interval* in order not to adversely affect throughput. In the meanwhile, *pst_interval* is a time-varying interval that adapts to real workloads. If the sojourn times of packets continuously exceed the threshold, ECN♯ reduces the interval of next marking to increase the marking probability (marking_next += *pst_interval* / `sqrt`(marking_count)), thus reducing the buffer occupancy.

3.3 Why ECN♯ works?

We discuss why ECN♯ can achieve all the design goals.

High throughput & low queueing delay: Instead of directly using a low instantaneous ECN marking threshold to aggressively bring down the buffer occupancy at the cost of throughput loss, ECN♯ uses a more conservative mechanism. By tracking the minimal queueing over a time interval, ECN♯ only marks packets after it determines persistent queue buildups. These queue buildups

RIGHTSLINK

are caused by providing excessive buffer space for flows with small RTTs, which do not contribute to the throughput but only increase the queueing delay. Eliminating these queues can benefit latency-sensitive short flows without throughput degradation. We carefully choose the interval to be around one worst case base RTT because TCP needs one RTT to react to ECN marking. Furthermore, the marking itself is conducted in a conservative way to mitigate throughput loss. Only after detecting persistent queue buildups, ECN$^\sharp$ marks one packet and schedules next marking after an interval. Afterwards, ECN$^\sharp$ checks again to see whether the queue expires or not. If not, ECN$^\sharp$ increases the marking frequency by reducing the interval. Through both persistent queueing detection and conservative ECN marking, ECN$^\sharp$ can achieve both high throughput and low queueing delay simultaneously.

Burst tolerance: ECN$^\sharp$ leverages the instantaneous ECN marking to achieve good burst tolerance. As discussed above, the instantaneous marking scheme enables fast response to bursty traffic, such as incast. Once the queue exceeds the threshold, it aggressively marks the packets to tame the subsequent bursts from senders and bring down the buffer usage. This scheme ensures a upper bound of buffer occupancy to avoid packet loss. Furthermore, ECN$^\sharp$'s instantaneous marking threshold is derived base on a high percentile RTT, which provides sufficient buffer without causing throughput loss.

3.4 Parameter Setting

ECN$^\sharp$ has 3 key parameters: *ins_threshold*, *pst_target* and *pst_interval*. We provide a rule-of-thumb to tune those parameters while leaving the optimality analysis as future work. We also conduct parameter sensitivity analysis in §5.4 to show that ECN$^\sharp$ can achieve good performance by following the rule-of-thumb.

The instantaneous marking threshold, *ins_target*, can be calculated based on Equation 2. As discussed in §2.3, we use high percentile RTTs (e.g., 90th percentile) to derive a high instantaneous marking threshold to avoid throughput loss.

Both *pst_interval* and *pst_target* are critical for identifying and mitigating persistent queue buildups. *pst_interval* determines the time that ECN$^\sharp$ spends tracking a queue before it confirms there is a persistent queue buildup. It has been recommended that the interval has to be around one RTT in order to accurately determine the persistent queueing [31]. Thus, we set *pst_interval* to be around the high percentile RTT to ensure that ECN$^\sharp$ can precisely detect the persistent queue buildups caused by flows with smaller RTTs. However, in real environment where traffic is very bursty, we can set a smaller value for *pst_interval* to increase the marking rate to achieve fast reaction to those dynamics.

For *pst_target*, theoretically, we can set a very small value for it. The reason is that by observing the queue for an interval, we can detect persistent queue buildups, and a lower target will further bring down those unnecessary queues. However, in datacenters, the queue oscillation is large due to some network settings (MTU settings, etc.) or NIC offloading (GSO, LSO, etc.) [12]. Thus, we also recommend setting it to be a more conservative value, which is larger than or equal to $\lambda \times \overline{RTT}$, where λ is determined by the transport protocol and $\overline{RTT}$ is the average RTT. This setting can reduce the buffer occupancy to an acceptable value without risking throughput loss.

3.5 Discussion

ECN$^\sharp$ vs CoDel: ECN$^\sharp$ is partially inspired by CoDel [31], an AQM solution to address the bufferbloat problem in Internet. CoDel uses the minimal queueing over an interval to distinguish good queues (queues that contribute to throughput) and bad queues (queues that do not contribute to throughput, such as the persistent queueing mentioned above). After detecting a bad queue, CoDel gradually reduces the marking interval to mitigate persistent queue buildups while not degrading throughput. However, such conservative marking makes CoDel react slowly to transient bursts, resulting in excessive packet losses. In contrast, ECN$^\sharp$ leverages the instantaneous ECN marking to keep good burst tolerance. Hence, ECN$^\sharp$ can greatly outperform CoDel in incast scenarios. We further analyze it through simulations in §5.4.

ECN$^\sharp$ vs TCN: TCN [16] combines the sojourn time and instantaneous ECN marking. However, under high RTT variations, it is challenging to decide a proper sojourn time threshold for TCN. Current practice with high percentile RTTs leads to persistent queue buildups. In contrast, ECN$^\sharp$ can conservatively mark packets based on persistent congestion state, thus delivering low latency.

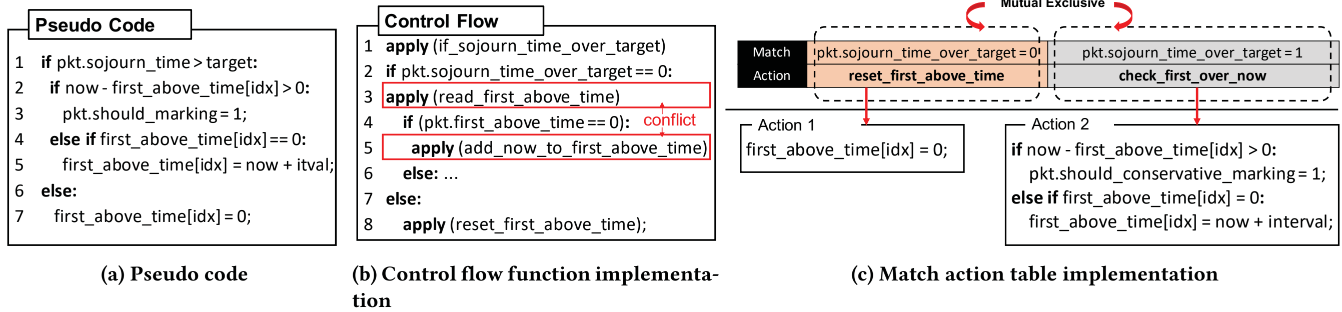
Probabilistic instantaneous marking: Some transport protocols, e.g. DCQCN [39], need probabilistic instantaneous marking to ensure fairness. Two instantaneous marking thresholds, K_{min} and K_{max} are used, packets are marked with probabilities from 0 to 1 when queue length resides from K_{min} to K_{max}, instead of "cut-off" behaviors (mark all packets if queue length exceeds a single threshold). One possible solution for ECN$^\sharp$ to work with probabilistic marking is to change the original cut-off marking into probabilistic marking, and keep the marking based on persistent congestion unchanged since it is conducted in a probabilistic way. Detailed analysis of ECN$^\sharp$ with probabilistic marking is beyond the scope of this paper.

4 IMPLEMENTATION

We have implemented an ECN$^\sharp$ prototype in the egress pipeline of Barefoot Tofino switch [5]. ECN$^\sharp$ is essentially a *stateful* data plane algorithm as the per-packet processing involves updates of multiple switch states. Our implementation has around 500 lines of P4 [10] code and is compiled to Barefoot Tofino switch using Barefoot Capilano SDE 6.1.1 [9].

Resource requirement: In our implementation, we use 7 match action tables in total. The number of table entries is less than 10 (most of match action tables use default actions, so no entry is needed explicitly). To store switch updates on all the 128 ports, we use 5 32-bit register arrays and 2 64-bit register arrays. The register memory consumption is only ~37KB. Each packet needs 124-bit metadata. Our ECN$^\sharp$ prototype only uses little switch resource, leaving more for other network functions.

RIGHTSLINK



(a) Pseudo code (b) Control flow function implementation (c) Match action table implementation

Figure 4: Different control flow implementations. Control flow implementation that directly interprets the pseudo code into control flow cannot be compiled to Tofino as it causes multiple accesses to the same register. Hence, we use match action table to optimize the ECN$^\sharp$ control flow instead.

We meet two main challenges in our implementation: 1) how to obtain the precise system time, and 2) how to update multiple switch states at line rate. In the following, we will describe our efforts to address them.

4.1 Emulate Precise System Time

As shown in Algorithm 1, ECN$^\sharp$ requires current system time to determine persistent queue buildups and mark packets. In the egress pipeline, each packet carries a metadata `egress_global_tstamp`, which has 64 bits and expresses current egress pipeline time in nanoseconds. However, we cannot directly use this 64-bit metadata. This is because we need to compare the current time with other states, e.g. `first_above_time`, using algorithm logical units (ALUs), but ALUs in Tofino only accept 32-bit input. Thus, we need an approximation method to derive a 32-bit system time. To solve this problem, there are several potential solutions:

- Use the lower 32 bits of `egress_global_tstamp`. The time is accurate enough but wrap around in every ~4 seconds (2^{32} nanoseconds). When the time wraps around, we will have a very small *marking_next* (line 16 of Algorithm 1), resulting in aggressive marks. A serious performance degradation in every 4 seconds is unacceptable.
- Use the higher 32 bits of `egress_global_tstamp`. The time is not accurate enough (around 4-second granularity) for low latency datacenters.
- Right shift `egress_global_tstamp` by 10 bits and then get lower 32 bits. The time is accurate enough (microsecond granularity) for datacenters and only wraps around in every 4295 seconds (2^{32} microseconds), which is more than 1 hour. However, the operation `shift_right` in Barefoot Capilano SDE can only support 32-bit input data.

Faced with above problems, we decide to *emulate* a 32-bit microsecond-granularity system time. Our key idea is given as follows. We first use the lower 32 bits of `egress_global_tstamp`, then right shift the 32 bits by 10 bits to get a 22-bit microsecond-granularity time. For the remaining higher 10 bits, we increase it by 1 whenever we observe the lower 22 bits wrap around. Finally, we attach the emulated 32-bit microsecond-granularity time to the packet as a metadata. Algorithm 2 gives the pseudo code. In our implementation, we use two 32-bit registers to store values of the lower 22 bits and the higher 10 bits, respectively.

Algorithm 2: Emulate Precise System Time

Input : pkt.egress_global_tstamp
Output: pkt.current_time
1 tmp_tstamp = `lower_32bits`(pkt.egress_global_tstamp)
2 time_low = `shift_right`(tmp_tstamp, 10)
3 **if** time_low ≤ register_low **then**
  /* Wrap around */
4   register_high = register_high + 1
5 **end**
6 register_low = time_low
7 pkt.current_time = register_high $\times 2^{22}$ + register_low

4.2 Update Switch States at Line Rate

ECN$^\sharp$ keeps several switch states on the switch registers and updates them for every packet arrival. To achieve line rate, Tofino imposes a register update limit: a Tofino program can only access a register once (in a pipeline). Note that in Tofino, reading a register, comparing the register value with another value, and then updating the register correspondingly are also treated as one access. For example, the line 4 to 5 of Figure 4a only has one access to `first_above_time[idx]`.

To implement the control flow of ECN$^\sharp$, we first directly use control flow function in P4 to apply different tables. A simple example is Figure 4b. However, directly interpreting the logic of pseudo code into control follow function implementation causes problems, e.g., both tables `read_first_above_time` and `add_now_to_first_above_time` access the same register `first_above_time[idx]`. This implementation cannot be compiled to the Tofino switch due to the hardware restrictions mentioned above. We notice that some reconfigurable hardware research work [29] also reports similar problem.

An alternative solution is to use `resubmit` primitive. The packet will be sent back to the beginning of the ingress pipeline with certain flags and regarded as a new packet. Then we can apply different tables based on the flags. This can guarantee the access to the

RIGHTSLINK

register is mutually exclusive. However, `resubmiting` packets occupies more pipeline resources and largely degrades the throughput.

To this end, we use the match action table to optimize the control flow of ECN$^\sharp$. Given a register, we first ensure that one register only has one table with actions accessing it. In this table, different actions reflect different control flow paths and they are naturally mutual exclusive. We further ensure the register is only accessed once within one action. Based on the two rules, we optimize the control flow by breaking it down into multiple match action tables. Figure 4c is one example. Before applying a match action table, we first compute the result of condition and attach the result to the packet as a metadata. Then we use the metadata to match the desired action, where the register will only be accessed once, and update switch states correspondingly. In our evaluation (§5.2), we confirm ECN$^\sharp$ can achieve line-rate performance.

5 EVALUATION

In this section, we conduct both testbed experiments and ns-3 simulations [3] to answer the following questions:

- **How does ECN$^\sharp$ perform in practice?** Our testbed experiment results show that compared with current practice, ECN$^\sharp$ achieves up to 23.4% smaller average FCT and 31.2% smaller 99th percentile FCT for small flows while providing the similar performance for large flows (§5.2). ECN$^\sharp$'s good burst control makes it largely outperform CoDel.
- **Does ECN$^\sharp$ scale to large datacenters?** We evaluate the performance of ECN$^\sharp$ using large scale ns-3 simulations (§5.3). Results show that ECN$^\sharp$ achieves up to 36.9% smaller FCT for small flows compared to current practice.
- **How does ECN$^\sharp$ control the queue length and react to bursty traffic?** We evaluate ECN$^\sharp$ using simulations mixing both the background traffic and bursty query traffic (§5.4). We find that compared to current practice, ECN$^\sharp$ keeps a much smaller average queue occupancy (182 packets to 8 packets). By changing the fanout of the incast traffic, we further show that compared to CoDel, ECN$^\sharp$ can support 1.75× more concurrent senders without TCP timeouts.
- **How robust is ECN$^\sharp$ to network settings?** Through targeted simulations, we show that ECN$^\sharp$ is robust to parameter choices (§5.4) and the packet scheduler (§5.4).

5.1 Methodology

Transport Protocol: We use DCTCP [12] at the end host by default. In testbed experiments, we use the DCTCP in Linux kernel 4.9.0 [2]. We also implement DCTCP in ns-3 simulator. The parameters are set as suggested in [12]. All endhosts in our evaluation are running DCTCP.

Schemes Compared: We compare ECN$^\sharp$ against the following three schemes:

- **DCTCP-RED:** [testbed, simulation] We use DCTCP-RED to refer to the modified RED in DCTCP paper [12], i.e., instantaneous ECN marking based on single threshold $K_{min} = K_{max} = K$. For testbed, we implement it on Barefoot Tofino switch. For simulation, we start from the ns-3's RED implementation and add instantaneous ECN marking. We set the threshold based on both the current practice, 90th percentile RTT (denoted as DCTCP-RED-Tail), and average RTT (denoted as DCTCP-RED-AVG). Please note, when only one queue is active, the performance of DCTCP-RED is identical to TCN.
- **CoDel:** [testbed, simulation] CoDel tracks minimal queueing over an interval to mark packets based on persistent congestion. For testbed, we implement CoDel on Barefoot Tofino to perform ECN marking.
- **TCN:** [simulation] TCN uses instantaneous sojourn time to adapt to packet schedulers. We implement TCN based on the software prototype provided by TCN paper [7].

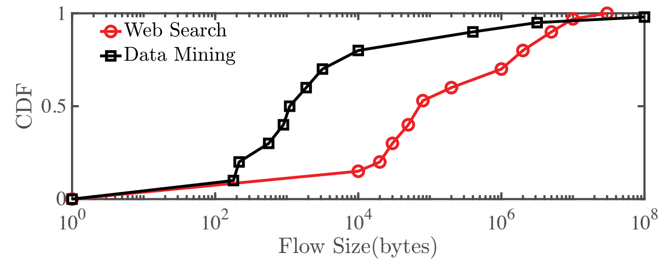


Figure 5: Flow size distributions.

Workloads: We generate traffic based on two realistic workloads in production: web search [12] and data mining [22]. The flow size distributions follow Figure 5. Both workloads are heavy-tailed. In testbed experiments, we use an open source traffic generator [8, 18] to generate the benchmark traffic. Similar to previous work [14, 17], flows arrive according to a Poisson process to achieve the desired network utilization. We also use the same approach to generate traffic in simulations.

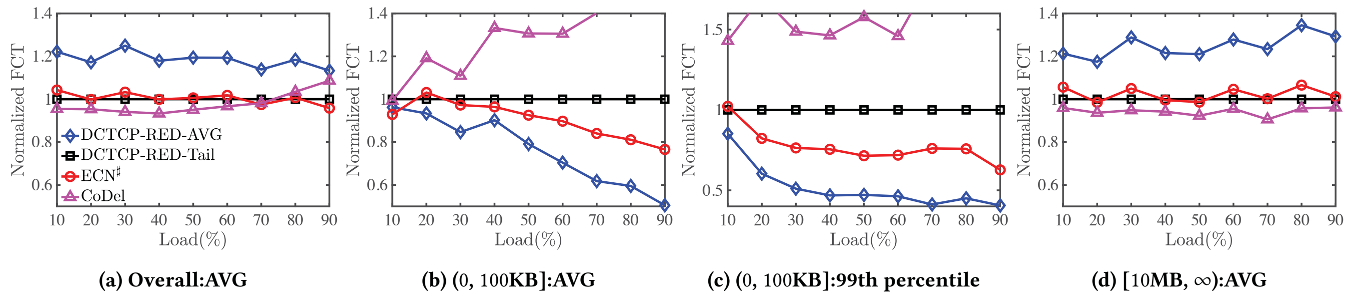
Metrics: We use the Flow Completion Time (FCT) as the primary metric. Besides the overall average FCT, we also breakdown FCT results across short flows (< 100KB) and large flows (> 10MB). We run experiments and simulations three times and report the average value.

5.2 Testbed Experiments

Testbed setup: We use the testbed in §2.3 with 8 servers connected to a Barefoot Tofino switch with ECN$^\sharp$'s implementation (§ 4). There are 7 senders and 1 receiver.

We use `netem` to emulate a 3× RTT variations (from 70μs to 210μs). The RTTs generated are based on the distribution in Figure 1, which is long-tail distribution. For DCTCP-RED, we set the marking threshold of DCTCP-RED-AVG to 80KB and DCTCP-RED-Tail to 250KB. For CoDel, we set *interval* to be 200μs and *target* to be 85μs. For ECN$^\sharp$, by following the rule-of-thumb(§3.4), we set the *ins_target* to be 200μs, *pst_interval* to be 200μs and *pst_target* to be 85μs.

RIGHTSLINK



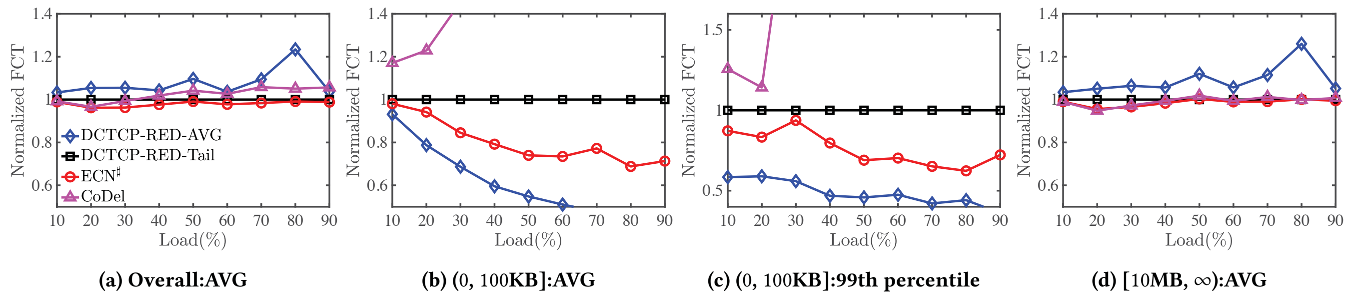
(a) Overall:AVG

(b) (0, 100KB]:AVG

(c) (0, 100KB]:99th percentile

(d) [10MB, ∞):AVG

Figure 6: [Testbed] FCT statistics with web search workload. All are normalized to the results achieved by DCTCP-RED with threshold derived from 90 percentile RTT.



(a) Overall:AVG

(b) (0, 100KB]:AVG

(c) (0, 100KB]:99th percentile

(d) [10MB, ∞):AVG

Figure 7: [Testbed] FCT statistics with data mining workload. All are normalized to the results achieved by DCTCP-RED with threshold derived from 90 percentile RTT.

Realistic Workloads

We evaluate ECN♯ using both web search and data mining workloads. The results are shown in Figure 6 and Figure 7. All results have been normalized to the result of current practice, DCTCP-RED-Tail. They are breakdown in terms of overall FCT (a), FCT of short flows (b, c) and FCT of large flows (d). All Figures show normalized FCT.

Short flows: Compared to current practice, i.e., DCTCP-RED-Tail, ECN♯ can achieve up to 23.4% (964μs to 738μs at 90% load) and 31.2% (548μs to 377μs at 80% load) lower average FCT in web search and data mining workloads, respectively (Figure 6b and Figure 7b). At the 99th percentile, ECN♯ achieves up to 37.2% (5242μs to 3287μs at 90% load) and 37.6% (2161μs to 1347μs at 80% load) lower FCT with both workloads (Figure 6c and Figure 7c). These results show that, by well controlling the persistent queue buildups caused by RTT variations, ECN♯ can eliminate unnecessary queueing delay and effectively improve the performance of delay-sensitive short flows.

We observe that DCTCP-RED-AVG achieves the best performance for short flows, but it adversely hurts the FCT of large flows, e.g., over 20% for the web search workload (Figure 6d). This is because DCTCP-RED-AVG sets a lower ECN marking threshold, which strictly limits the queueing to a very low level, thus benefiting FCT of the short flows. In the meanwhile, such a low marking threshold throttles the throughput of large flows, affecting their FCT.

We also note that CoDel achieves very bad performance for short flows. The reason is that CoDel suffers from frequent TCP timeouts (1 TCP timeout adds > 1 milliseconds to FCT) because it does not react to the instantaneous queueing state, and thus experiences frequent packet loss under traffic bursts.

Large flows: From Figure 6d and Figure 7d, we see ECN♯ achieves similar results as DCTCP-RED-Tail and outperforms DCTCP-RED-AVG with both workloads. For web search workload, ECN♯ achieves up to 25.6% (94411μs to 70192μs at 80% load) lower FCT compared to DCTCP-RED-AVG. For data mining workload, ECN♯ achieves up to 20.5% (1049035μ to 833396μs at 80% load) lower FCT compared to DCTCP-RED-AVG. The reason is that by conservatively marking packets when observing persistent queue buildups, ECN♯ can bring down the buffer occupancy without hurting throughput for large flows.

Overall: In general, ECN♯ achieves good overall performance among the four schemes. In data mining workload, ECN♯ performs the best at all loads, whereas in web search workload its overall FCT degradation is within 4.2% (2240μs to 2148μs) compared to DCTCP-RED-Tail at 10% load (Figure 6a). In contrast, DCTCP-RED-AVG achieves very bad performance. This further confirms that ECN♯ can maintain the throughput of all flows.

Towards Larger RTT Variations

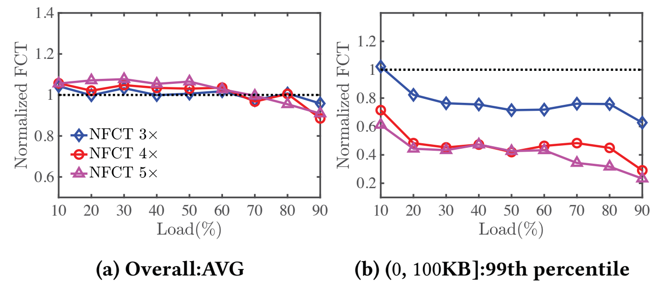
We increase the RTT variation from 3× (70μs to 210μs) to 5× (70μs to 350μs). We evaluate ECN♯ using the more challenging

RIGHTSLINK

web search workload, where the traffic is more bursty. We compare ECN$^\sharp$ with current practice, DCTCP-RED-Tail. The results are shown in Figure 8. NFCT $n\times$ denotes the normalized FCT of ECN$^\sharp$ to DCTCP-RED-Tail when RTT has $n\times$ variation.

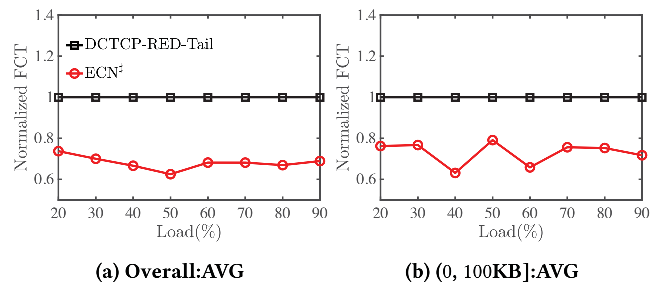
As shown in Figure 8a, ECN$^\sharp$ generally achieves similar overall FCT as DCTCP-RED-Tail. When the RTT variation reaches 5×, the FCT degradation is within 7.6%(2854μs to 2650μs at 30% load). When the load is higher, ECN$^\sharp$ can achieve slightly better results than DCTCP-RED-Tail. The reason is that due to the conservative marking for persistent congestion, ECN$^\sharp$ can consistently achieve comparable overall FCT no matter how RTT variation enlarges.

Furthermore, ECN$^\sharp$ greatly outperforms DCTCP-RED-Tail for short flows. As show in Figure 8b, ECN$^\sharp$ can achieve up to 37.3% (5242μs to 3287μs at 90% load) lower FCT when the RTT variation is 3×. The advantage is enlarged to 71.2% and 73.4% when the variation is 4× and 5× respectively. This shows ECN$^\sharp$ can effectively reduce the FCT for short flows by controlling the persistent congestion. When the RTT variation is larger, ECN$^\sharp$ can bring more benefit to delay-sensitive short flows.



(a) Overall:AVG **(b) (0, 100KB]:99th percentile**

Figure 8: [Testbed] FCT statistics with web search workload when RTT variation enlarges. NFCT $n\times$ denotes the normalized FCT of ECN$^\sharp$ to DCTCP-RED-Tail when RTT has $n\times$ variation. ECN$^\sharp$ achieves lower FCT for short flows and comparable throughput consistently.



(a) Overall:AVG **(b) (0, 100KB]:AVG**

Figure 9: [Simulations] FCT statistics of large-scale simulations with web search workload. All results have been normalized to DCTCP-RED-Tail.

5.3 Large Scale Simulations

To complement the small-scale testbed, we further evaluate ECN$^\sharp$ on a larger-scale spine-leaf topology with production workloads (§5.3).

Setup: We simulate a 128-host leaf-spine topology with 8 spine and 8 leaf switches. Each leaf is connected to 16 servers via 10Gbps links. The spine and leaf switches are also connected via 10Gbps links. We use ECMP for load balancing. The RTT has 3× variations and varies from 80μs to 240μs. The average RTT here is $\sim$ 137μs and 90th percentile is $\sim$ 220μs. The distributions are similar to Figure 1, which is a long-tail distribution.

In our simulation, the calculation of packets' sojourn time is implemented with packet tags in ns-3 [4]. When a packet enqueues, we add a packet tag with the enqueue timestamp to the packet. When the packet dequeues, we calculate the sojourn time by deducting the enqueue timestamp from the current timestamp.

We evaluate ECN$^\sharp$ using both web search and data mining workloads. The results are shown in Figure 9. Due to the space limitation, we omit the results with the data mining workload. All results have been normalized to FCT achieved by DCTCP-RED-Tail.

Short flows: As shown in Figure 9b, ECN$^\sharp$ outperforms DCTCP-RED-Tail by at least 18.5% (7195μs to 5867μs) at 50% load and the performance gap reaches 36.9% (7297μs to 4607μs) at 40% load. This further confirms that, by controlling the persistent queue buildups, ECN$^\sharp$ can effectively reduce the FCT for short flows.

Overall: As shown in Figure 9a, compared to DCTCP-RED-Tail, ECN$^\sharp$ achieves 26.3% (37479μs to 27644μs at 20% load) to 37.4% (36931μs to 23095μs at 50% load) lower overall average FCT. This is because ECN$^\sharp$ maintains high throughput for large flows while effectively improving latency for short ones.

5.4 ECN$^\sharp$ Deep Dive

In this section, we first show a microscopic view of queues to explain the superior results achieved in both §5.2 and §5.3. Then, we analyze ECN$^\sharp$'s parameter sensitivity. Finally, we compare ECN$^\sharp$ with the latest datacenter AQM solution, TCN [16].

Microscopic View

Simulation setup: We use a simple 16 to 1 topology with 10Gbps links, 16 servers are senders and 1 receiver. Other settings are similar to §5.3. For CoDel, the *interval* is set to 240μs while *target* is set to 10μs.

We send flows from 16 senders to the receiver. The size of both large and short flows are generated based on data mining workload. We also generate some query flows to emulate bursty incast traffic. The size of query flows follows a uniform distribution from 3KB to 60KB. We start N (100 in default) concurrent query flows at 4s. We further change the value of N to show how these bursty query flows affect the performance of all schemes.

Queue occupancy: To better illustrate how different schemes manage their queues, we sample the queue length of the bottleneck link for 0.005 seconds and the results are shown in Figure 10. Compared to other schemes, ECN$^\sharp$ achieves two advantages:

RIGHTSLINK

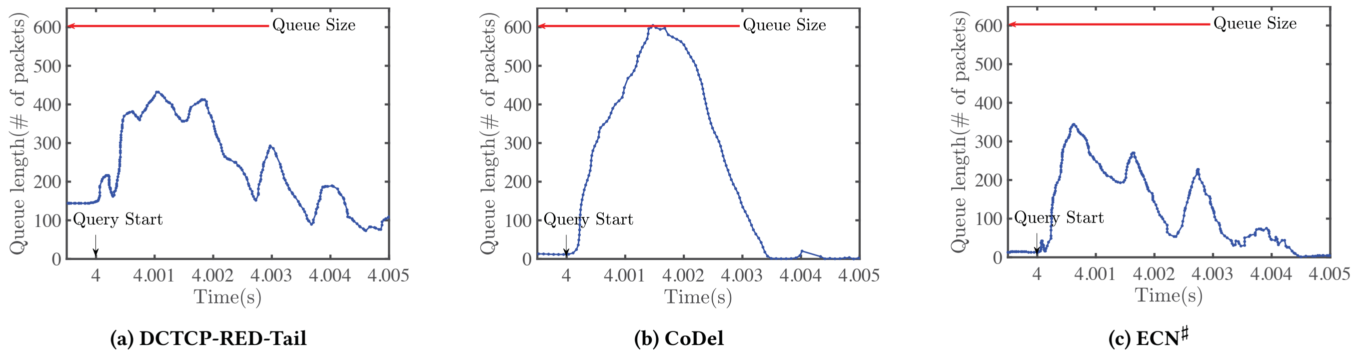


Figure 10: [Simulations] Queue occupancy (with 100 concurrent query flows). ECN♯ achieves much lower queue occupancy (8 packets) compared to current practice, DCTCP-RED-Tail (182 packets). Meanwhile, ECN♯ achieves comparable performance in handling incast (no packets drop), large outperforms CoDel (drops 125 packets).

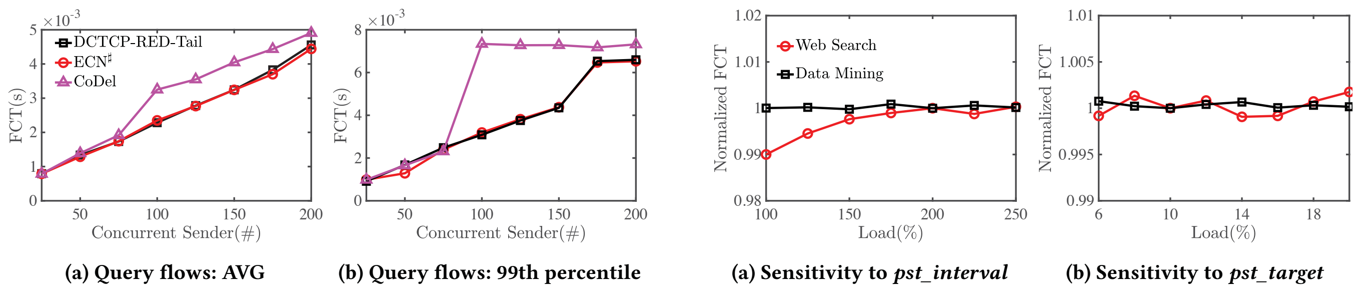


Figure 11: [Simulations] Statistics of query flow completion time. ECN♯ can achieve comparable performance as DCTCP-RED-Tail and largely outperforms CoDel.

Figure 12: [Simulations] Parameter Sensitivity. ECN♯ is robust to parameter choices with only <1% variations on overall FCT with different parameter settings.

- **Low persistent queueing:** ECN♯ keeps much lower queue occupancy (8 packets) compared to DCTCP-RED-Tail (182 packets). The low queue occupancy confirms ECN♯'s effectiveness in handling persistent queue buildups caused by the mismatch between RTTs and marking threshold.
- **No packet drops with incast flows:** ECN♯ achieves comparable results in handling bursty query traffic (no packet drops) with DCTCP-RED-Tail, outperforming CoDel (drop 125 packets). The reason is that by leveraging the merit of instantaneous marking, ECN♯ can achieve good burst tolerance.

Impact of bursty query flows: To have a deeper view on how those schemes handle bursty query flows, we change the concurrent query senders from 25 to 200 and trace the query completion time. The results are shown in Figure 11. When there are 100 concurrent senders, CoDel begins to suffer from packet loss, seriously degrading the performance. On the contrary, ECN♯ achieves comparable performance with DCTCP-RED-Tail, and begins to suffer from packet loss when the concurrent senders are increased to 175. In summary, due to its good burst tolerance, ECN♯ can support 1.75× more concurrent senders than CoDel with bursty incast traffic. The reason is that ECN♯ leverages the instantaneous marking and has a good control over bursty traffic.

Parameter Sensitivity

We next show how robust ECN♯ is to different parameter settings. We set both *pst_target* and *pst_interval* based on the rule-of-thumb proposed in §3.4. We conduct analysis on both workloads. The results are shown in Figure 12. We have the following two observations from the results:

- Different parameter settings do not have a large impact on the performance of ECN♯ considering the overall FCT. The difference is within 1% for web search workload and 0.2% for data mining workload.
- After reducing the value of *pst_interval*, ECN♯ behaves differently on the two workloads. For data mining workloads, ECN♯ performs slightly worse (0.2% longer FCT). However, for web search workload, ECN♯ performs 1% better. The reason is that web search workload is more bursty, and a more aggressive marking mechanism is better. We can reduce the *pst_interval* to increase the marking frequency.

Packet scheduler

We finally compare ECN♯ with TCN [16], the latest AQM solution to show how ECN♯ works with arbitrary packet scheduler. We mainly focus on the following questions.

RIGHTSLINK

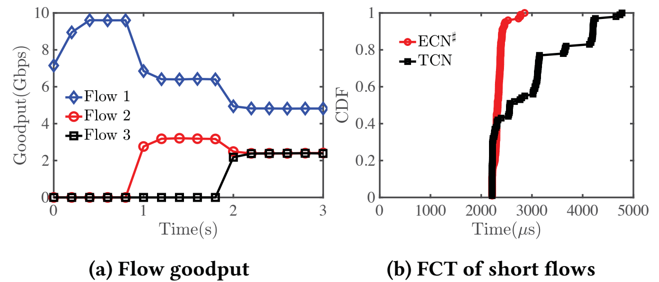
- Can ECN♯ preserve the packet scheduling policy?
- Can ECN♯ mitigate the persistent queueing exacerbated by RTT variations even with packet scheduler?

To evaluate this, we configure the switch with Deficit Weighted Round Robin (DWRR) with 3 queues/services. The weights among the 3 queues/services are 2 : 1 : 1. We first start a long-lived TCP flow from sender 1 and classify this flow into queue/service 1, then from sender 2 and classify into queue/service 2, finally from sender 3 into queue/service 3. We also randomly start short flows (the size is from 3KB to 60KB) from the rest of senders to probe the queue occupancies. We set the TCN marking threshold to 150μs to avoid throughput loss. The other settings are identical to §5.4.

Figure 13a shows the goodput of flows achieved by ECN♯. We observe at the beginning, only queue 1 is active, flow 1 achieves around ∼ 9.6Gbps goodput (Goodput is slightly smaller than throughput due to packet header overhead). After flow 2 starts, queue 2 becomes active and flow 1 achieves ∼ 6.42Gbps goodput while flow 2 achieves ∼ 3.18Gbps goodput. Finally, when three queues all become active, flow 1, flow 2 and flow 3 achieve around 4.82Gbps, 2.40Gbps and 2.40Gbps goodput respectively, which strictly preserves the packet scheduling policy.

We also measure the FCT of short flows among all queues and the results are shown in Figure 13b. Compared to TCN, ECN♯ achieves 19.6% better average FCT (2913μs to 2341μs) for short flows.

In summary, ECN♯ can strictly preserve the packet scheduling policy with multiple queues/classes. Furthermore, compared to TCN, ECN♯ achieves better performance for short flows because ECN♯ eliminates persistent queue buildups exacerbated by the RTT variations.



(a) Flow goodput **(b) FCT of short flows**

Figure 13: [Simulations] ECN♯ with packet schedulers. ECN♯ can preserve the network scheduling policy while achieve better performance for short flows.

6 RELATED WORK

ECN-based Transports in Datacenters: The ECN-related literature on datacenter networks is vast. Alizadeh et al. [12] identified the transport requirements in production datacenters and proposed DCTCP to address the challenges. To achieve good burst tolerance, DCTCP uses a simplified version of ECN/RED [20] to mark packets at the switch.

ECN⋆ [37] uses the standard TCP congestion control algorithm at the end host. D^2TCP [36] and L^2DCT [30] modified the DCTCP congestion control algorithm to meet flow deadlines . Shan et al. proposed CEDM [34] to accurately mark ECN to reduce throughput loss by reducing queue oscillations. These solutions apply the same AQM marking as DCTCP . Thus they all suffer from increased queueing delay with the current practice under RTT variations (§2.3). ECN♯ can benefit all of them. DCQCN [39] is a rate-based congestion control to enable RDMA in datacenters. At the switch, it requires RED-like probabilistic marking to ensure convergence. ECN♯ can be extended to work with it.

MQ-ECN [18] first pointed out the drawbacks of existing ECN/RED implementations in packet scheduling context. To adapt to the varying queue capacity caused by packet schedulers, TCN [16] proposed to use instantaneous sojourn time to mark packets. ECN♯ inherits the merit of TCN, but further tracks the persistent congestion state to reduce long-term queue buildups.

Buffer Sizing and AQM in Internet: Appenzeller et al. [15] showed that the minimum buffer size should be $C \times RTT/\sqrt{n}$ when there are a large number of n concurrent large flows. However, datacenter networks typically have a small number of concurrent large flows [12]. Therefore, people still applied the old rule-of-thumb $C \times RTT$ for buffer sizing for regular TCP.

Many AQM solutions have been proposed to handle bufferbloat in Internet. CoDel [31] tracks the minimum sojourn time over a varying time interval to detect bad queues. PI [26] and PIE [32] use a proportional-integral controller to keep the queueing to a constant target. Relative to them, ECN♯ further applies the aggressive instantaneous ECN marking to handle transient bursts.

New Design over Programmable Switches: With the emergence of programmable switches, such as Barefoot Tofino. Many clean designs, such as NDP [25] and PERC [27] have been proposed to deliver ultra low latency communication. However, they are not compatible with state-of-the-art ECN-based solutions in datacenters. On the contrary, ECN♯ is fully compatible with ECN-based solutions and utilizes the networking programmability to mitigate performance degradation caused by RTT variations.

7 CONCLUSION

In this paper, we have presented ECN♯ a simple yet effective solution that enables ECN in datacenters with high RTT variations. ECN♯ marks packets based on both instantaneous and persistent congestion states. It inherits the merit of current instantaneous ECN marking based on a high-percentile RTT to achieve high throughput and burst tolerance, and further marks packets when observing long-term switch queue buildups to eliminate the persistent queueing caused by flows with smaller RTTs. We have implemented ECN♯ on a Barefoot Tofino switch and evaluated it through both testbed experiments and ns-3 simulations. Our evaluation shows that ECN♯ is a viable solution that achieves all our design goals.

ACKNOWLEDGMENTS

We thank the anonymous CoNEXT reviewers and our shepherd Prof. Eric Rozner for their constructive feedback and suggestions. This work is supported in part by Hong Kong RGC GRF-16215119, CRF-C703615G, and a Huawei research grant.

REFERENCES

[1] 2012. DCTCP in Windows Server 2012. https://technet.microsoft.com/en-us/library/hh997028(v=ws.11).aspx. (2012).

[2] 2014. DCTCP in Linux kernel 3.18. https://kernelnewbies.org/Linux_3.18. (2014).

[3] 2015. The Network Simulator ns-3. https://www.nsnam.org. (2015).

[4] 2015. The Network Simulator ns-3. Packets. https://www.nsnam.org/docs/release/3.10/manual/html/packets.html#tags-implementation. (2015).

[5] 2016. Barefoot Tofino. https://www.barefootnetworks.com/technology/. (2016).

[6] 2016. netem in Linux. https://wiki.linuxfoundation.org/networking/netem. (2016).

[7] 2016. TCN Prototype. https://github.com/HKUST-SING/TCN-Software. (2016).

[8] 2016. Traffic Generator. https://github.com/HKUST-SING/TrafficGenerator. (2016).

[9] 2018. Barefoot Capilano SDE. https://www.barefootnetworks.com/products/brief-capilano/. (2018).

[10] 2018. P4 Language. https://p4.org/. (2018).

[11] 2019. Linux Virtual Server. http://www.linuxvirtualserver.org. (2019).

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

[13] Mohammad Alizadeh, Adel Javanmard, and Balaji Prabhakar. 2011. Analysis of DCTCP: Stability, Convergence, and Fairness. In *SIGMETRICS 2011*.

[14] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pFabric: Minimal Near-optimal Datacenter Transport. In *SIGCOMM 2013*.

[15] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. 2004. Sizing Router Buffers. In *SIGCOMM 2004*.

[16] Wei Bai, Kai Chen, Li Chen, Changhoon Kim, and Haitao Wu. 2016. Enabling ECN over Generic Packet Scheduling. In *CoNEXT 2016*.

[17] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. 2015. Information-agnostic Flow Scheduling for Commodity Data Centers. In *NSDI 2015*.

[18] Wei Bai, Li Chen, Kai Chen, and Haitao Wu. 2016. Enabling ECN in Multi-service Multi-queue Data Centers. In *NSDI 2016*.

[19] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A Fast and Reliable Software Network Load Balancer. In *NSDI 2016*.

[20] Sally Floyd and Van Jacobson. 1993. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking (ToN)* 1, 4 (1993), 397–413.

[21] Rohan Gandhi, Hongqiang Harry Liu, Y. Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. 2014. Duet: Cloud Scale Load Balancing with Hardware and Software. In *SIGCOMM 2014*.

[22] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM 2009*.

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

[24] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. 2015. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *SIGCOMM 2015*.

[25] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *SIGCOMM 2017*.

[26] CV Hollot, Vishal Misra, Don Towsley, and Wei-Bo Gong. 2001. A control theoretic analysis of RED. In *INFOCOM 2001*.

[27] Lavanya Jose, Lisa Yan, Mohammad Alizadeh, George Varghese, Nick McKeown, and Sachin Katti. 2015. High Speed Networks Need Proactive Congestion Control. In *HotNets 2015*.

[28] Glenn Judd. 2015. Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter. In *NSDI 2015*.

[29] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2016. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *SIGCOMM 2016*.

[30] Ali Munir, Ihsan A Qazi, Zartash A Uzmi, Aisha Mushtaq, Saad N Ismail, M Safdar Iqbal, and Basma Khan. 2013. Minimizing flow completion times in data centers. In *INFOCOM 2013*.

[31] Kathleen Nichols and Van Jacobson. 2012. Controlling Queue Delay. *ACM Queue* 10, 5 (2012), 20:20–20:34.

[32] R. Pan, P. Natarajan, C. Piglione, M. S. Prabhu, V. Subramanian, F. Baker, and B. VerSteeg. 2013. PIE: A Lightweight Control Scheme to Address the Bufferbloat Problem. In *HPSR 2013*.

[33] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. 2013. Ananta: Cloud Scale Load Balancing. In *SIGCOMM 2013*.

[34] Danfeng Shan and Fengyuan Ren. 2017. Improving ECN marking scheme with micro-burst traffic in data center networks. In *INFOCOM 2017*.

[35] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *SIGCOMM 2015*.

[36] Balajee Vamanan, Jahangir Hasan, and T.N. Vijaykumar. 2012. Deadline-aware Datacenter TCP (D2TCP). In *SIGCOMM 2012*.

[37] Haitao Wu, Jiabo Ju, Guohan Lu, Chuanxiong Guo, Yongqiang Xiong, and Yongguang Zhang. 2012. Tuning ECN for Data Center Networks. In *CoNEXT 2012*.

[38] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. 2017. High-resolution Measurement of Data Center Microbursts. In *IMC 2017*.

[39] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion Control for Large-Scale RDMA Deployments. In *SIGCOMM 2015*.

RIGHTSLINK