PRED: Performance-oriented Random Early Detection for Consistently Stable Performance in Datacenters

Xinle Du[†] Tong Li[‡,*] Guangmeng Zhou[§] Zhuotao Liu[§] Hanlin Huang[§]
Xiangyu Gao[§] Mowei Wang[†] Kun Tan[†] Ke Xu[§,*]
[†]*Huawei Technologies* [‡]*Renmin University of China* [§]*Tsinghua University*

Abstract

Random Early Detection (RED) has been integrated into datacenter switches as a fundamental Active Queue Management (AQM) for decades. Accurate configuration of RED parameters is crucial in achieving high throughput and low latency. However, due to the highly dynamic nature of workloads in datacenter networks, maintaining consistently high performance with statically configured RED thresholds poses a challenge. Prior art applies reinforcement learning to predict proper thresholds, but their real-world deployment has been hindered by poor tail performance caused by instability. In this paper, we propose PRED, a novel system that enables automatic and stable RED parameter adjustment in response to traffic dynamics. PRED uses two loosely coupled systems, Flow Concurrent Stabilizer (FCS) and Queue Length Adjuster (QLA), to overcome the challenges of dynamically setting RED parameters to adapt to the ever-changing traffic pattern. We perform extensive evaluations on our physical testbed and large-scale simulations. The results demonstrate that PRED can keep up with the real-time network dynamics generated by realistic workloads. For instance, compared with the static-threshold-based methods, PRED keeps 66% lower switch queue length and obtains up to 80% lower Flow Completion Time (FCT). Compared with the state-of-the-art learning-based method, PRED reduces the tail FCT by 34%.

1 Introduction

Datacenters host a variety of services with distinct networking preferences. For example, storage [1] and data mining [2] require high throughput, while web search [3] and machine learning [4] services require low latency. To meet these application requirements, a lot of innovative congestion control algorithms (e.g., DCTCP [3], DCQCN [5], TIMELY [6], HPCC [7], Swift [8], BFC [9]) have been proposed to reduce the delay caused by in-network queuing.

These new congestion controls demonstrate tremendous improvements; however, deployment is a challenging issue. For instance, it takes Google a year to release a new kernel to support new network functions [8, 10]. Many congestion controls need to be deployed on hardware Network Interface Cards (NICs) (e.g., HPCC [7]) or require new switches to support them (e.g., BFC [9]), which cannot be deployed in heterogeneous datacenters made up of legacy devices [11]. In some particular scenarios, such as multi-tenant datacenter networks, it is also difficult to modify the network protocol stack at the end host [12, 13].

To compensate for the host-based new congestion control protocols, the community also explores more accurate in-network congestion signals [14–17]. For instance, Explicit Congestion Notification (ECN) [18] is widely deployed in datacenter networks as the new congestion signal besides packet losses. Random Early Detection (RED) [19], the algorithm that controls how to discard packets or how to mark packets for ECN, has also been integrated into switches as a basic Active Queue Management (AQM) function. Many production datacenter transport protocols rely on ECN and RED [20–22], such as DCTCP and DCQCN.

Broadly speaking, the prior work on addressing RED/ECN's shortcomings can be classified as either new AQMs (e.g., TCD [14]), case-based RED/ECN (e.g., ECNsharp [15], BCC [16]), or learning-based RED/ECN (e.g., ACC [17]). The new AQM relies on a new switch design, thus requiring a long product release cycle [17]. The case-based arts (e.g., ECNsharp, BCC) realize that RED/ECN with a fixed threshold is ineffective in datacenters and therefore propose to consider multiple cases during threshold adjustment. However, they only recognize a limited number of cases and still use fixed thresholds for each case (see details in § 6.3.2). Prediction model-based ACC [17] leverages the deep reinforcement learning (DRL) [23] to adjust the RED settings dynamically in reaction to network congestion. However, it might make the adjustment of the RED unstable, resulting in sharp changes in queue length and poor tail performance (see details in § 6.3.3).

Thus from a philosophical standpoint, it is worth asking: Why can't static RED adapt to dynamic traffic? Can the prob-

*Tong Li and Ke Xu are the corresponding authors.

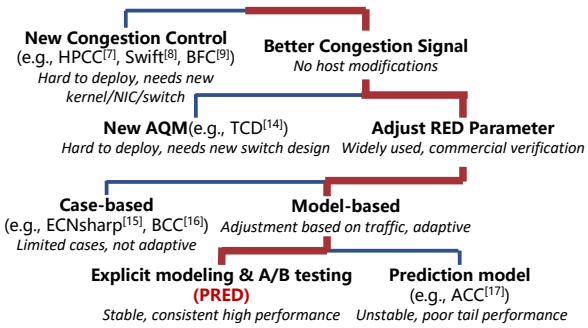


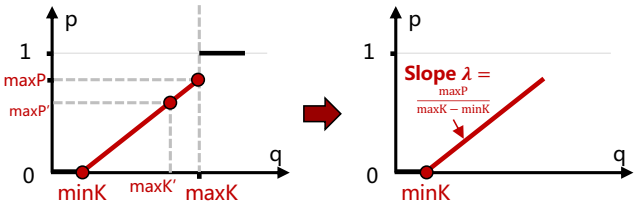
Figure 1: Design space in Datacenters.

lem be solved simply by dynamically adjusting the threshold of the already widely used RED without modifying the algorithm itself? Is there a stable way to adjust the RED threshold? In this paper, we seek to answer these questions with a new scheme PRED (Performance-oriented RED).

Figure 1 surveys the design space for congestion control and places PRED in context by following the thick red lines through the design tree. At the highest level, PRED is an AQM implemented on the switches to provide a better congestion signal. Next, instead of designing a new AQM that needs an entirely new switch design, PRED dynamically adjusts the parameters of the widely commercially proven RED in datacenters. As shown in the lowest branch of the design tree, PRED falls into the category of explicit modeling and A/B testing approach that is more adaptive and stable.

Through experiments, we have discovered that the performance of RED is mainly influenced by the concurrency and distribution of traffic. Prediction model-based ACC [17] initially employed a traffic-aware approach to proactively adjust the RED threshold; However, it failed to provide stable adjustment due to two reasons: **the impact of rapid changes in concurrent flow numbers** and **inaccurate predictions**. Different concurrency levels require different settings, but because there is a mismatch between the delay of DRL and the delay of estimating concurrency, ACC cannot make adjustments based on concurrency levels. Moreover, due to inherent error rates in machine learning predictions, the design principle of ACC fails to achieve consistently stable performance.

PRED employs two loosely coupled systems, namely the Flow Concurrent Stabilizer (FCS) and Queue Length Adjuster (QLA), to address the difficulties associated with unstable adjustments in dynamically adjusting RED parameters. In order to keep the queue length stable under different concurrency levels, FCS uses modeling to adjust RED parameters by counting the number of concurrent flows on a switch port. In order to achieve stable real-time adjustment of the queue length to accommodate varying traffic, QLA employs a step-by-step approach, utilizing testing and verification methods, to gradually modify the RED parameters. With FCS and QLA, PRED can achieve consistently high performance under dynamic traffic in a more stable way without modifying the RED logic.



(a) Two-point form (b) Point-slope form

Figure 2: Two Forms of RED parameter configuration.

We implement a prototype of PRED using the barefoot Tofino switch [24] as the programmable data plane and evaluate PRED extensively using various use cases. Our testbed experiments show that PRED can keep queues stable under various flow-concurrent conditions. Based on real network workloads, we demonstrate that PRED can properly adjust the RED parameters to a more workload-friendly state to keep up with the traffic dynamics.

We further perform larger-scale simulations to evaluate PRED. Our quantitative results confirm that PRED advances state-of-the-art in various aspects. For example, compared with the algorithms with static thresholds, PRED can achieve up to 80% lower Flow Completion Time (FCT) for short flows. From the microscopic view of switch queues, PRED effectively mitigates queue buildups by keeping the switch queue length 66% lower than that of the static threshold algorithms (from 25 to 15 packets). We further demonstrate that PRED reduces the 99th FCT by 34% compared to ACC, the state-of-the-art DRL-based RED configuration approach, whose uncertainty results in suboptimal parameter selections in extreme cases.

The rest of the paper is organized as follows. We introduce the background in § 2 and the motivation of the PRED in § 3. § 4 illustrates the design of our solution. In § 5, we address the implementation of PRED on P4 [25]. In § 6, we evaluate PRED in NS-3 [26] and a small-scale testbed. § 7 surveys the related work. Finally, § 8 concludes this paper.

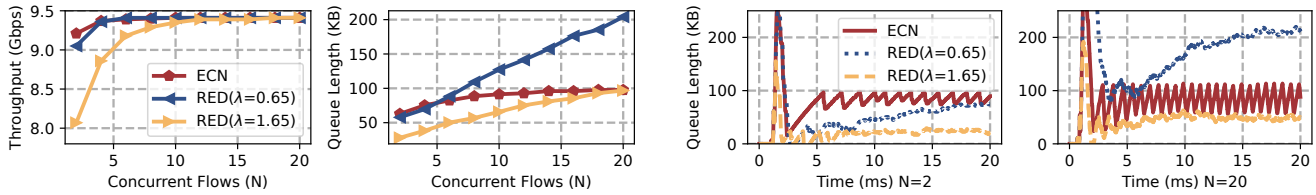
2 Background

2.1 The Form of RED Setting: Two-point Turns to Point-slope

RED [19] is the most widely applied AQM scheme and is commonly supported by commodity switches. [1] The state-of-the-art congestion control mechanisms have widely adopted RED in datacenter networks. For example, DCTCP [3] and DCQCN [5] adopt RED with the standard ECN on switches and use the marked-packet-aware rate control on end hosts. First of all, to facilitate the design of PRED in this paper, we revisit the RED parameter configuration by explaining why we replace the two-point form with the point-slope form.

Two-Point Form. RED works on switches and makes decisions about marking packets based on the output queue length

[1] We note that if not otherwise specified, RED adopts packet marking ECN instead of packet discarding in this paper.



(a) Testbed Result (b) Simulation Breakdown

Figure 3: RED performance with different flow concurrency levels (N).

(denoted by q). The decision-making is driven by three parameters, including the maximum marking probability (maxP), and two queue thresholds (minK and maxK). When the queue length is lower than minK, no action is taken. When the queue length exceeds maxK, every arrived packet is marked. When the queue length is between minK and maxK, RED calculates a probability p that the packet should be marked. In production networks, linear interpolation is the common practice to calculate p as $p = \text{maxP} \cdot \frac{q-\text{minK}}{\text{maxK}-\text{minK}}$. As plotted in Figure 2(a), the resulting curve of p is in the *two-point* form: point (minK, 0) and point (maxK, maxP).

Point-Slope Form. In this paper, we argue that applying the equivalent *point-slope* form makes the design of the RED parameter configuration more interpretable (see §2.2). As shown in Figure 2(b), we define λ ($\lambda \in (0, +\infty)$) as the slope of the line. λ is computed as $\lambda = \frac{\text{maxP}}{\text{maxK}-\text{minK}}$, and thus we have $p = \lambda \cdot (q - \text{minK})$. For example, given $\text{minK} = 0.005$ MB, $\text{maxK} = 0.2$ MB, $\text{maxP} = 0.1$, we have $\lambda \approx 0.5$.

2.2 The Essence of RED Setting: Controlling the Steady-state Queue Length

RED is a tool for congestion control convergence, so adjusting the RED threshold is essentially adjusting the steady-state queue length as it converges. To explain why we replace the two-point form with the point-slope form and the essence of the RED setup, we first take DCTCP as an example of the RED parameter breakdown based on a fluid model [3,5]. Considering N long-lived flows traversing a single bottleneck link with capacity C and propagation delay d (RTT without queueing), the relationship between the steady-state queue length (denoted by $\dot{q}$) and the RED setting is derived as follows[2] (see Appendix A for detailed derivations):

$$(\dot{q} - \text{minK})^2(\dot{q} + Cd) = \frac{2N}{(\frac{\text{maxP}}{\text{maxK}-\text{minK}})^2} = \frac{2N}{\lambda^2} \qquad (1)$$

λ can monotonically influence $\dot{q}$: The first thing we infer from Equation (1) is that λ can monotonically influence $\dot{q}$. This reveals that the point-slope form expresses a more clear physical meaning than the two-point one. If we adjust minK, maxK and maxP, different settings may lead to the same result, for example, (minK,maxK,maxP) and (minK,maxK',maxP') in Figure 2(a). It is easier for the operator to adjust the parameters if there is only one variable and the linear adjustment will result in a linear result. Therefore, the point-slope is better than the two-point because the monotone change in the λ in the two-point form results in a linear change in the steady-state $\dot{q}$. In the latter, we use (minK,λ) to replace (minK,maxK,maxP).

Key factors affecting the $\dot{q}$: From Equation (1), we can see that the steady-state $\dot{q}$ is only related to three key factors. **(a) The network-specific factor**, including the bottleneck link C and propagation delay d. Once the network topology is built, this factor will be fixed. **(b) The flow concurrency level** N. The larger N is, the larger $\dot{q}$ is. If the network experiences drastically changing N, the steady-state $\dot{q}$ will change accordingly. [3] **(c) The setting of RED parameters** (the only one that the network operator typically controls), i.e., λ and minK. By setting the RED parameters, we can properly adjust the steady-state queue length.

3 Motivation

3.1 RED Requires Traffic Awareness

We define the *flow concurrency level* on a switch as the number of flows whose packets are buffered in the switch's output queue at a certain time, define the *flow distribution* as the distribution of flow sizes and define *steady-state queue length* (denoted by $\dot{q}$) as the average queue length of the switch output queue at the stable state. Based on the measurement studies on datacenter network workloads with varying flow concurrency levels and flow distributions, we observe two interesting observations in RED parameter configurations.

Observation 1 (Flow Concurrency): Steady-state queue length grows with the number of flows. From Equation (1) in § 2, we conclude that the larger N is, the larger $\dot{q}$ is. In this section, we also verify this observation through experiments. In these experiments, we run many-to-one (incast) tests with different flow concurrency levels through both testbed evaluation and NS-3 simulation (see setup details in § 6). *ECN* acts as the baseline, which represents the legacy way of marking packets using a single threshold [3], i.e., minK = maxK. *RED* ($\lambda = 1.65$) represents a more aggressive RED parameter setting with higher ECN marking probabilities, and *RED*

[2] Similar conclusions are reached by other ECN-based congestion controls, as demonstrated in studies [27]. Additional sources, including DCQCN and TCP [27, 28], also uphold consistent conclusions. For brevity, we skip the detailed discussion. The assumption of N long-lived flows is for simplicity in understanding the congestion model, but it is not a strict requirement.

[3] The RED capacity of queue adjustment is limited. When N becomes too large, Equation (1) becomes ineffective. The detailed discussion is in § 4.3.

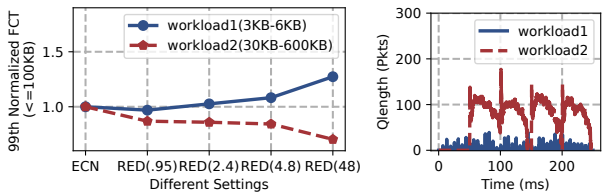


Figure 4: RED performance with different workloads.

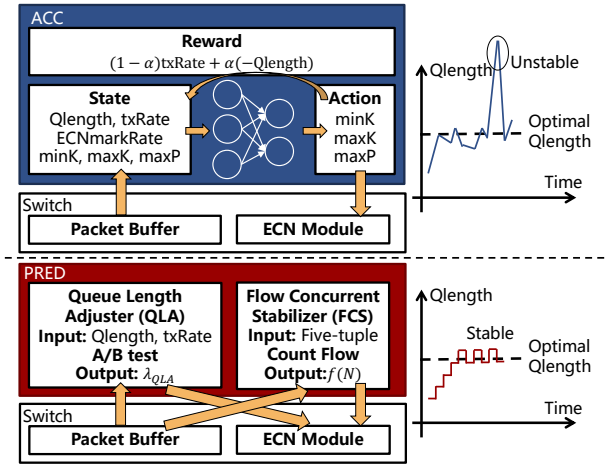


Figure 5: The design comparison of ACC and PRED.

($\lambda = 0.65$) represents a more conservative RED.

Figure 3(a) shows the testbed results of total throughput and queue length on the switch. Through experiments, we again verify that the queue length of both RED increases with the increase of N as shown in Figure 3(a). This conclusion holds even in the case of multiple bottlenecks, where several hops before the last hop are also congested (see § 6). Datacenter network traffic can be very bursty, but the static RED algorithm increases $\dot{q}$ as the flow concurrency level increases. The tail FCT of short flows will increase with the increase of concurrency, which seriously affects the application performance. A large number of concurrent flows can also suffer from the TCP incast throughput collapse because the queue length is not bounded at the bottleneck switch. A high queue length may trigger back pressure mechanisms (e.g., ECN in DCTCP and priority-based flow control (PFC) in DCQCN) that restrain pushing more packets into the network.

We can also find that different concurrency levels require different λ. *RED* ($\lambda = 1.65$) suffers from under-bandwidth utilization when the flow concurrency level is low, and *RED* ($\lambda = 0.65$) suffers from high queuing delay (queue length) when the flow concurrency level is high. We further zoom into the run-time queue length via simulations for the case of $N = 2$ and $N = 20$. As shown in Figure 3(b), a larger λ (e.g., $\lambda = 1.65$) achieves better performance (i.e., high bandwidth and low latency) when the concurrency level is large (e.g., $N = 20$), and vice versa. Although *ECN* achieves acceptable bandwidth utilization and overall low latency, its amplitude of the steady-state queue length increases with the flow concurrency level. This indicates that applying the single threshold is insufficient to achieve consistent low latency. **To sum up, static RED is not suitable for dynamic flow concurrency.**

Observation 2 (Flow Distribution): Small flows require a large steady-state queue length, while large flows require a small steady-state queue length. In this experiment, we examine RED performance given different flow distributions. We created two different workloads to represent pure small flows and large flows: workload 1 has a flow distribution of size from 3 KB to 6 KB, and workload 2 has a flow distribution of size from 30 KB to 600 KB. In the many-to-one scenario, the number of concurrent senders is fixed at 18 (see setup details in § 6). Figure 4(a) plots the normalized 99th-percentile FCT of flows. The x-axis denotes different settings of λ. It is clear that under workload 1, FCT increases with λ, while under workload 2, FCT decreases with λ. This shows that the proper RED parameter configuration highly depends on the flow distributions in the workload.

For flow distribution such as workload 1, where small flows are dominant, the bottleneck queue is only occupied for a short time. The changes in queue length are spikes one by one in Figure 4(b). Link utilization is not full all the time. Therefore, in this case, increasing the ECN marking probability (increasing λ) will further reduce the link utilization, resulting in a larger FCT of the small flows. To achieve low FCT, the steady-state queue length is expected to be large enough (decrease λ) to avoid triggering packet marking at the bottleneck switch. On the other hand, for flow distribution such as workload 2, where larger flows are dominant, the high link utilization and high bottleneck queue occupancy will last for a long period. This queue occupancy is unnecessary and significantly impacts the latency of small flows. Thus, the steady-state queue length is expected to be small enough to achieve low FCT performance. **In conclusion, static RED is also unsuitable for dynamic flow distribution.**

3.2 Our Goals: Traffic Awareness and Stability

Since the influence of flow concurrency and flow distribution makes the static RED unsuitable for dynamic flows, a natural question is: Can we make the RED parameters configuration traffic-aware? ACC [17] and PRED share a common high-level approach of dynamically adjusting RED parameters. In the following sections, we will highlight the limitations of ACC and explain how PRED overcomes these limitations by achieving stable queue length adjustments.

Limitations of ACC. ACC was the first to use deep reinforcement learning (DRL) to adjust RED parameters to make RED traffic-aware. As shown in Figure 5, ACC takes the following four features as the input state: The current queuing length, the output data rate for each link, the output rate of ECN marked packets for each link, and the current ECN setting. The output action is the next ECN setting. Then, high throughput and short queues are used as reward functions to continuously train the neural network to learn historical data, so as to make

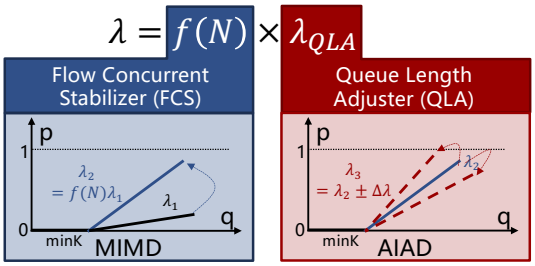


Figure 6: The design rationale of PRED.

predictions for future scenarios. DRL is a valuable approach for adapting to dynamic environments. However, as discussed in § 6.3.3, ACC tends to induce high instability in the queue, resulting in prolonged tail FCT and substantial degradation of the overall user experience.

There are two reasons for the instability of ACC adjustment: (a) **The impact of rapid changes in concurrent flow numbers:** ACC struggles to promptly adjust to varying concurrency levels because of the swift fluctuations in N, resulting in instability. Given that the delay in DRL falls within the millisecond range, while the calculation of concurrency level N operates at the microsecond scale, achieving real-time matching of N is infeasible. Even if ACC can match N in real-time, it also has the following problem. (b) **Inaccurate predictions:** ACC operates on a prediction model derived from observed and learned historical data to make decisions. However, incorporating additional network conditions does not ensure accurate predictions matching 100% real-world scenarios. Predictions inherently carry an error rate, and when an inaccurate prediction is made, the network's performance becomes unstable, ultimately leading to a degradation in tail FCT. Next, we will explain how PRED can be stabilized through the following two problems.

How to bound queue according to flow concurrency? Our answer is to count N directly in the switch and adjust the RED λ according to the flow concurrency. In Equation (1), we analyzed the relationship between $\dot{q}$ and N and λ. The left-hand term $(\dot{q} - \mathsf{minK})^2(\dot{q} + Cd)$ of Equation (1) contains the steady-state queue $\dot{q}$, and if we want $\dot{q}$ to remain constant, then the right-hand term $\frac{2N}{\lambda^2}$ of Equation (1) needs to remain constant. N changes dynamically, but if the switch can calculate the current value of N and then increase/decrease the corresponding λ, then the steady-state queue length $\dot{q}$ will not change with N. We design the Flow Concurrent Stabilizer (FCS) to count the N and change the λ to the concurrency level. As shown in Figure 5, the input to the FCS is a five-tuple of each packet. It outputs the multiplier factor of the change in the RED parameter by calculating the number of flows over a period of time. In Figure 6, FCS works in the form of Multiplicative-Increase Multiplicative-Decrease (MIMD) because it can calculate the corresponding λ directly from N.

How can queue length adjustment be traffic-aware in a stable manner? Our answer to this question is that we test and then verify in small steps. The traffic is uneven and time-

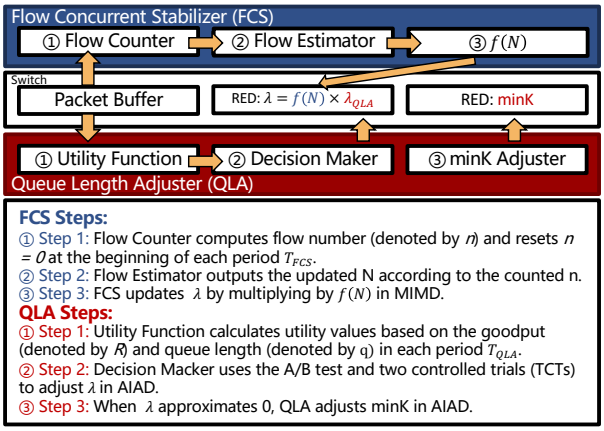
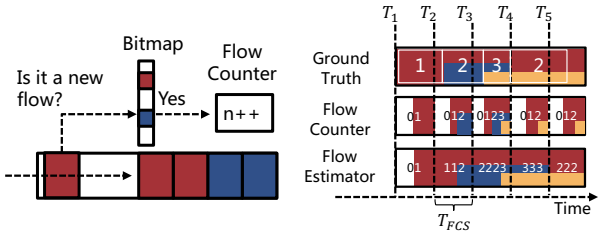


Figure 7: The design of PRED.

varying in datacenters. In order to achieve stable high performance, some workloads demand a smaller steady-state queue length to operate at the optimized point [29], while others demand a larger steady-state queue length to avoid inefficient link utilization. It is difficult for the switch to know the needed steady-state queue length. But the switch knows the queue length and throughput. If the switch can always maintain a high throughput and a low queue length, this queue length must be close to the needed steady-state queue length. We design the Queue Length Adjuster (QLA) to achieve traffic awareness at run time. In Figure 5, QLA takes throughput and queue length as input and outputs a corresponding linear factor λ_{QLA} through the A/B test. First, it sets a utility function in terms of goodput and queue length to judge the performance of the current RED setting. And then, as shown in Figure 6, it applies a direct small modification to λ_{QLA} to adapt λ to the traffic. To reduce the impact of noise, it is tested twice and adjusted only when the results are consistent. Note that any measurement behavior will affect the network performance itself, so the detection step of FCS should be small. Using any of the more aggressive detection algorithms here leads to instability in the network, such as Additive-Increase Multiplicative-Decrease (AIMD), binary search, and prediction-based machine learning algorithms. The QLA works in the form of Additive-Increase Additive-Decrease (AIAD) because it needs stable but fast convergence. AIAD is a heuristic process that relies on A/B testing. Unlike other methods, it demonstrates a clear convergence direction with incremental testing steps. However, it tends to operate at a slower pace compared to other algorithms, which may prioritize speed over stability.

4 Design

In this paper, we propose PRED to answer the question of how to set RED settings dynamically. Figure 7 illustrates the design of PRED, primarily focused on adjusting λ to maintain an optimized $\dot{q}$. The FCS determines N using the flow counter and flow estimator. Subsequently, it applies $f(N)$, a monotonically increasing function of N, to dynamically



(a) Flow Counter (b) Flow Estimator

Figure 8: Examples of how the modules work in FCS.

adjust λ according to the concurrency level. The QLA directly modifies λ_{QLA} to achieve traffic-awareness at runtime. In the following sections, we will delve into each module, covering topics such as N estimation, utility function design, noise reduction through two controlled trials, and dynamic adjustments of minK.

4.1 Flow Concurrent Stabilizer

As shown in Figure 7, the FCS comprises three modules: Flow Counter, Flow Estimator and $f(N)$. (1) Within each T_{FCS} period, the Flow Counter computes the number of flows (denoted by n) traversing a port and resets $n = 0$ at the period's onset. (2) Simultaneously, the Flow Estimator calculates the updated value of N based on the accumulated n. (3) Subsequently, the FCS module adjusts λ by multiplying it with $f(N)$. In the following, we explain how the Flow Counter and Flow Estimator work and discuss the selections of $f(N)$.

Flow Counter (to count new flows). A flow can be identified by the hash of the five-tuple: <source_ip, destination_ip, source_port, destination_port, protocol>. We define the start of the flow as the first time a packet with a particular five-tuple appears at a switch and count the number of flows that send a packet within each timeout-sized interval T_{FCS}. This new-arrival count way helps to avoid biases in cases when the start of flows or the end of flows are missed [30]. The logic is straightforward, as in Figure 8(a), the Flow Counter checks if it is a new flow arrival by looking up the bitmap maintained for each port. If yes, the flow counter increases n by 1. For every period of T_{FCS}, bitmap and n are reset and recalculated.

Flow Estimator (to estimates N). Flow Estimator estimates N by taking into account both the last-period n (denoted by n_{last}) and this-period n output by the Flow Counter, which is $N = \max\{n_{last}, n\}$. Since Flow Counter always starts at 0, we can't treat n as the current N. Figure 8(b) shows an example. 'Ground Truth' represents the actual value of flow concurrency at the present moment. At T_2, the Flow Counter outputs $n = 0$, while the last-period estimate $n_{last} = 1$. If FCS uses n instead of N, FCS will get a bad flow estimate of 0. In this paper, we estimate N at T_2 as $N = \max\{n, n_{last}\} = 1$.

Selection of $f(N)$. In order to bound the steady-state queue length regardless of flow concurrency, according to Equation (1), $f(N)$ should meet $f(N) \geq \sqrt{N}$. Through experiments, we find that using $f(N) = \sqrt{N}$ cannot completely bound the length of the steady-state queue because Equation (1) assumes N synchronous traffic, which is difficult to meet when N becomes large. We test with the choice of $f(N)$ through experiments and choose $f(N) = N$. We have further given the reason why N is feasible in § 6.

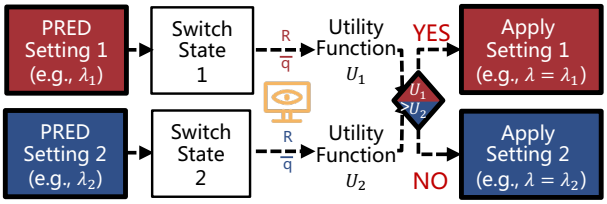


Figure 9: The closed-loop decision-making of QLA.

4.2 Queue Length Adjuster

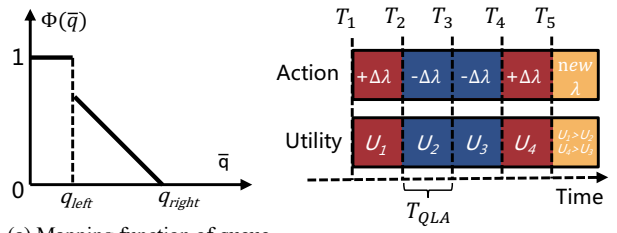
As shown in Figure 7, the QLA comprises three modules: Utility Function, Decision Maker and minK Adjuster. (1) The Utility Function calculates utility values based on the goodput (denoted by R) and the average queue length (denoted by $\bar{q}$) within each period T_{QLA}. (2) The Decision Macker employs the A/B testing and two controlled trials (TCTs) to adjust PRED in AIAD. (3) If deemed necessary (e.g., when λ approaches 0), it further adjusts minK using a similar approach.

The QLA module applies a closed-loop decision-making structure to set λ according to the demand of the traffic pattern. The efficiency of this performance-oriented learning approach is also validated in prior works (e.g., PCC [31–33]). As illustrated in Figure 9, the control action of QLA is its choice of PRED setting. The decision-making is based on the A/B testing of two actions, e.g., $\lambda = \lambda_1$ and $\lambda = \lambda_2$.These performance metrics are combined with the numerical utility values, say U_1 and U_2, respectively, via the utility function. The decision-making of λ is then based on comparing U_1 and U_2. However, the dynamic nature of the network can produce a lot of noise, which leads to inaccurate results in the A/B testing. To reduce the impact of noise, TCTs and adjusts only when the results are consistent.

Utility Function (to decide trade-offs). From the network operator's point of view, we usually select the critical network performance metrics of latency and throughput as the reward function [17]. We define the utility function as a trade-off between latency and throughput (i.e., the trade-off between high utilization and low queue):

$$U(\lambda_t) = \beta \times \frac{R_t}{C} + (1-\beta) \times \Phi(\bar{q}_t)$$

R represents the average throughput of one egress queue, i.e., the amount of data delivered to the link during the time interval T_{QLA}. We normalize the R by the link bandwidth C to represent the link utilization. The latency is represented by the average queue length $\bar{q}$ to indicate the impact of queuing delay. β is a weight factor. We select the average value instead of the instantaneous queue length because the instant queue length varies in a large range, which can make utility unstable. Φ() is a mapping function as shown in Figure 10(a). Generally, the lower the queue length, the better. Note that when the



(a) Mapping function of queue length

(b) Two controlled trials

Figure 10: Design details of QLA.

queue goes short (e.g., $\bar{q} \leq q_{left}$), the marginal benefit of further reducing the queue length goes to zero. This avoids the situation where the utility function becomes unstable when the queue is too short.

Decision Maker (to reduce noise impact). A simple A/B test will be disturbed by network noise. To decide which direction and amount to change its value, QLA uses TCTs to reduce the noise impact. Assume QLA is currently at setting λ_{QLA}. As shown in Figure 10(b), QLA takes four consecutive T_{QLA} and divides them into two pairs. For each pair, QLA attempts a slightly higher $\lambda_{QLA}+\Delta\lambda$ and slightly lower $\lambda_{QLA}-\Delta\lambda$, each for one T_{QLA}. After the four consecutive trials, QLA needs to judge the next adjustment based on the four utility values ($U_i, i = 1,2,3,4$) obtained four times. If the higher λ_{QLA} consistently has higher utility ($U_1 > U_2$ *and* $U_4 > U_3$), then FCS adjusts its λ_{QLA} to $new\lambda_{QLA} = \lambda_{QLA}+\Delta\lambda$; and if the lower λ_{QLA} consistently has higher utility then QLA picks $new\lambda_{QLA} = \lambda_{QLA}-\Delta\lambda$; But, if the results are inconclusive, e.g., $U_1 > U_2$ *but* $U_4 < U_3$, QLA stays at its current λ_{QLA}.

Adjustment of minK. Besides λ, the other important value in the RED parameter is minK. The initial value of minK can be set very small so that λ can make the steady-state queue adjusted over a wide range ($[\text{minK}, +\infty)$) theoretically. However, in the design, we cannot use $\lambda = 0$ to get a large steady-state queue $\dot{q}$ because the FCS module fails. So when $\lambda < \lambda_{\min}$, QLA no longer adjusts λ and adjusts minK instead. Let minK change ΔminK each time and λ always stays λ_{min} for FCS. For example, in § 3.1, workload 1's burst flow does not occupy the buffer. In this case, minK should be bigger, and the steady-state queue $\dot{q}$ is further increased to avoid the burst being marked with ECN.

4.3 Discussion

Some notable design details of PRED are essential for reasoning about the differences from legacy RED. We briefly discuss these considerations below.

Range of PRED adjustment capabilities. The RED parameter adjustment has limitations, particularly when N surpasses a certain value. Through experiments, we test the adjustment range of PRED and observe that when the number of concurrent long flow N exceeds 32, PRED is unable to maintain queue stability (see Appendix D.6). Fortunately, the occurrence of concurrent long flows is rare [3], indicating that the range of PRED is generally adequate in practice. Moreover, to address high-concurrency short-flow bursts, FCS can rapidly notify the end nodes by adjusting λ to ensure fast convergence of the queue.

Computing and storage overhead. The computing and storage requirements are minimal. FCS is designed to be pipelined on the data plane, allowing all steps to execute in parallel, regardless of the size of the topology or the number and types of present flows. QLA involves a simple comparison of several values, imposing small computational overhead. In terms of storage requirements, each port requires only a few registers to cache the traffic count N and a corresponding bitmap. For example, each port uses 128 bits, N uses 8 bits, and the remaining 120 bits are used as bitmap.

About the maxK. In PRED, we utilize the point-slope form (minK,λ) instead of the the two-point form (minK,maxK,maxP). This means that in the point-slope form, the maxK is implicitly incorporated within the parameter λ. However, we acknowledge that maxK is still necessary for final protection in practical implementation because the point-slope is still implemented on the two-point basis. To accommodate extreme scenarios, we set a larger fixed value for maxK. In our measurements, maxK is set to 500 KB.

Selection of T_{FCS}. The T_{FCS} should be carefully set as its accuracy significantly affects the performance. First of all, T_{FCS} must be greater than RTT because the estimate will be small because at least one RTT is required for all flows to pass through the switch. T_{FCS} should not be too big because when the flow number decreases, the update of N will be delayed. As shown in Figure 4.1(b), N remains 3 from time T4 to time T5, while the ground truth is only 2. Considering the queuing delay during congestion, we find that setting $T_{FCS} = 1.25RTT$ is applicable. We have also verified the parameter sensitivity analysis in Appendix D.5.

Selection of T_{QLA}. Note that T_{FCS} is independent of T_{QLA} because the two modules are separate modules. The setting of T_{QLA} is a tradeoff. The smaller T_{QLA}, the faster the convergence, but the more unstable. Vice versa. In addition, it is also limited by the location where the QLA module is deployed. If the QLA module is deployed on the control plane, the interaction delay between the control plane and the data plane should be considered. In our implementation, reading the register in the control plane takes 5 to 12 ms, and updating the RED table takes 80 to 110 ms. Therefore T_{QLA} is at least 200 ms due to hardware limitations. We discuss the details in Appendix C.2. In this paper, it is recommended $T_{QLA} = 5RTT$ in simulation (on the data plane) and $T_{QLA} = 400$ ms in the testbed (on the control plane due to hardware limitation). We have also verified the parameter sensitivity analysis in Appendix D.5.

Selection of other parameters. PRED also has some parameter settings that are trade-offs. In QLA, β is the weight that represents the bandwidth-delay tradeoff. The network operator can easily set the parameters based on the requirement of running applications. For example, our measurements are

based on $\beta = 0.4$. The initial minK needs a relatively small value in order to ensure that the adjustment range of the λ can be larger. The initial minK is 10 packets. q_{left} is the parameter that QLA uses to ensure that the value of the utility function remains stable when the queue is small. The larger the q_{left}, the more stable the utility value, but the longer the queue length. q_{left} is set to 15 packets in this paper. λ_{min} is the dividing point between the adjusting λ and the adjusting minK in QLA, we use $\lambda_{min} = 0.05$, For the trial step parameter setting, the bigger the trial step, the faster the adjustment speed, but the worse the stability, we use $\Delta\lambda = 0.025$, and $\Delta\text{minK} = 5$ packets in this paper. PRED tunes the RED parameters because they are sensitive to traffic dynamics. The parameters introduced by PRED are traffic-insensitive. However, fine-tuning these parameters remains essential, and enhancing the parameter selection process is a key focus of our future work.

5 Implementation

Ideally, PRED's FCS and QLA would be implemented in switch ASICs. We implement a prototype of PRED on Barefoot Tofino 1 programmable switch, including 12 MAU stages, 120 MB SRAM and 6.2 MB TCAM per pipeline. The development effort on the switch includes about 350 lines of *P*4 code [34] for the data plane and 300 lines of Python for the control plane. The FCS module needs to quickly recognize *N*, so it is deployed on the data plane. Theoretically, the QLA can be deployed on either the data plane or the control plane. Deployment on the control plane has low resource overhead but high latency, and vice versa. We chose to deploy the QLA to the control plane due to limited MAU stages. We have implemented PRED where both QLA and FCS are deployed on the data plane, but 17 stages are needed (see details in Appendix C). Tofino 1 [24] has only 12 stages, so we deploy the QLA module on the control plane. However, the new Tofino 2 and Tofino 3 switches support 20 stages [24], so the QLA and FCS modules can be deployed in the data plane at the same time. All implementation details are deferred to Appendix C.

6 Evaluation

In this section, we conduct both testbed experiments and NS-3 simulations[4] to answer the following questions:

- How does PRED perform in practice and scale to large datacenters?
- Why does PRED achieve high performance?
- What are the advantages of PRED compared to Deep Reinforce Learning methods?
- How do we decide the key parameters in PRED itself?

6.1 Methodology

Transport Protocol. We are using DCTCP [3] as the default congestion control at the end host. We also have experiments to verify the capabilities of PRED with DCQCN (details in Appendix D.1). Since the results are similar, we only use DCTCP later. In our testbed experiments, we used DCTCP from Linux kernel 4.15.0 [35]. The NS-3 simulation [26] implementation of DCTCP refers to the official implementation [36]. The parameters are set as suggested in [3].

Schemes Compared. We compare PRED against the following 9 schemes:

(a) RED [testbed, simulation]: RED's implementation is based on instantaneous ECN marking rather than weighted average queues, as in the DCTCP [3] and DCQCN [5] papers. For the testbed, we implement it on the Barefoot Tofino switch. For simulation, we start from the NS-3's RED implementation and add instantaneous ECN marking.

(b) ECN [testbed, simulation]: Here we refer to the single-valued RED algorithm as ECN, i.e., instantaneous ECN marking based on a single threshold $minK = maxK = K$.

(c) ECNsharp [simulation]: ECNsharp [15] marks packets based on both instantaneous and persistent congestion.

(d) CoDel [simulation]: CoDel [37] tracks minimal queueing over an interval to mark packets based on persistent queueing.

(e) ACC [simulation]: ACC [17] is a practical approach that allows automatic adjustment of ECN parameters at switches by utilizing deep reinforcement learning (DRL). We refer to the original design in paper [17] and re-implement ACC in the NS-3 simulation.

(f) Only FCS [testbed, simulation]: The particular PRED which contains only the Flow Concurrent Stabilizer.

(g) Only QLA [testbed, simulation]: The particular PRED which contains only the Queue Length Adjuster.

(h) TCN [simulation]: TCN uses instantaneous sojourn time to adapt to packet schedulers. We implement TCN based on the software prototype provided by TCN paper [38].

(i) HPCC [simulation]: HPCC [7] is a new congestion control scheme, which needs to modify the NIC and the switch that supports INT. HPCC can adjust the rate MIMD through the INT information.

Metrics. We use the Flow Completion Time (FCT) as the primary metric. Besides the overall average FCT, we also break down FCT results across small flows (< 100 KB) and large flows (> 1 MB). All are normalized to the results achieved by PRED. We also show a queue length timing diagram for a fine-grained comparison of the various algorithms.

6.2 Testbed Experiments

In this section, we analyze the performance of PRED. Our assessment aims to confirm the stability of traffic concurrency (FCS), assess the effectiveness of dynamically adjusting queues in response to traffic dynamics (QLA), and determine the importance of integrating both modules.

Flow Concurrent Stabilizer. We first tested PRED's stability of traffic concurrency, as well as different design options. We use the testbed in § 3.1 with 3 servers connected to a Barefoot Tofino switch with PRED 's implementation. There are 2 senders and 1 receiver. We used iperf3 [39] for sending traffic

[4]The implementation in NS3 is identical to that of the programmable switch, yet the languages differ; NS3 is C++, and the testbed is P4.

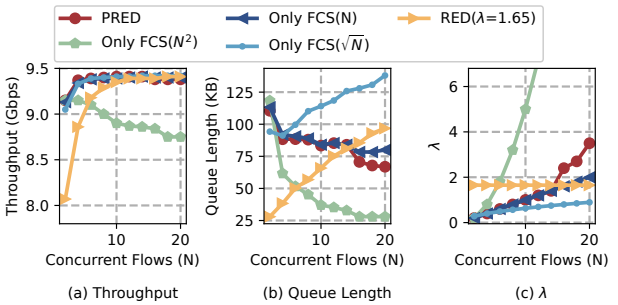


Figure 11: [Testbed] PRED performance with different flow concurrency levels (N).

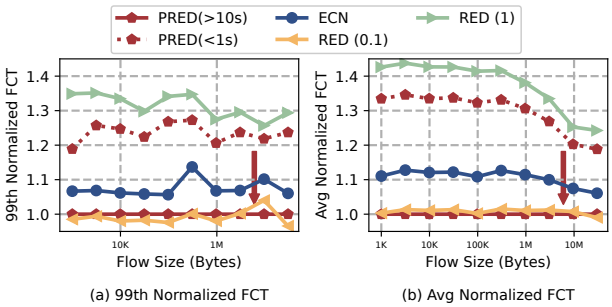


Figure 12: [Testbed] FCT statistics ($T_{QLA} = 0.4$ s).

and measured the throughput and queue length by varying the concurrent flows sent by each sender. Using only two senders is to verify that concurrency problems still exist in the case of multiple bottlenecks (one at the sender and one at the switch). We use RED as a comparison. We pick initial λ whose effect is approximately equal when N=2. λ in FCS(N) and PRED is 0.1 ($2 \times 0.1 = 0.2$), λ in FCS(N^2) is 0.05 ($2^2 \times 0.05 = 0.2$), and λ in FCS($\sqrt{N}$) is 0.2 ($\sqrt{2} \times 0.2 = 0.28$).

Figure 11 shows the performance of PRED and only FCS with different design choices. We can see from the figure 11(a) that all algorithms except FCS(N^2) and RED can maintain a relatively high throughput of about 9.4 Gbps. FCS(N^2) sacrificed about 0.4 Gbps throughput due to low queue length. From figure 11(b), the queue length of PRED and FCS(N) remained stable with the increase of N. The queue length of FCS($\sqrt{N}$) was still growing, but the queue length of FCS(N^2) was decreasing with the increase of N. The reason for the noise when $N = 2$ is that there is not enough concurrency. Figure 11(c) shows the λ as N changes.

The above analysis shows that it is most appropriate for PRED to select the N for the FCS module. In § 4.1, we analyze that $f(N) = \sqrt{N}$ can eliminate the influence of N on queue length. However, the modeling assumes concurrent traffic, so when N<5, the FCS module can still maintain a stable queue, but when N>5, the concurrent traffic assumption fails so that the queue will grow. We also tested $f(N) = N$ and $f(N) = N^2$, N^2 turns out to be too aggressive, while N is more feasible.

Finally, we compared PRED and only FCS (N) separately. As can be seen from Figure 11(b), when N>15, PRED will make the queue length lower without losing throughput. Compared with only FCS, PRED can further adjust the queue length by adjusting λ according to the network condition.

Queue Length Adjuster. We next tested PRED's ability to tune parameters dynamically. We use 7 servers connected to a Barefoot Tofino switch. There are 6 senders and 1 receiver. Each sender uses an open-source traffic generator [40, 41] to generate the benchmark traffic to the receiver. The network load is 60%. K in ECN is 70 packets. We generate traffic based on WebSearch [3] realistic workloads.

Figure 12 shows FCT on different flow sizes. From the results, we can see that the 99th FCT and average FCT of RED (λ =0.1) is the smallest, and the FCT of RED (λ =1) is the largest. With the same workload, PRED does not initially have a suitable λ, so the FCT of PRED (<1s) is close to RED (1). When run for about 10 s, PRED converge to the appropriate λ, so the FCT of PRED (>10s) converges to RED (0.1). Note that the implementation of PRED on Tofino 1 places QLA on the control plane, so $T_{QLA} = 0.4$ s. As discussed in § 5, the bigger T_{QLA} is, the slower convergence rate is, but the deterministic direction of adjustment is stable. If PRED can be implemented on Tofino 2 or a better switch, T_{QLA} can become smaller and PRED can converge faster.

Based on the above analysis, we know that PRED can bound steady-state queue length with dynamic flow concurrency and can make queue length adjustment traffic-aware in a stable manner to maximize the network effect.

6.3 Simulation Experiments

In this section, we will use simulation to analyze the performance of PRED. At the same time, we will fine-grain compare the differences between different algorithms and answer why PRED can have a good performance. First, the performance of PRED in real datacenters is tested by large-scale NS-3 simulation. Then we analyze why FCS and QLA should cooperate with each other through the sequence diagram of queue length and analyze why the performance of different algorithms is not optimal. Then we compared the differences between DRL-enable ACC and PRED. Finally, we analyze the PRED related parameter settings and packet scheduler in detail.

6.3.1 Large Scale Simulations

To complement our testbed experiments, we evaluate PRED on a larger-scale spine-leaf topology with realistic workloads.

Setup: We simulate a 128-host leaf-spine topology with 8 spine and 8 leaf switches. Each leaf is connected to 16 servers via 10 Gbps links. The spine and leaf switches are also connected via 10 Gbps links. The latency of the link is 10 μs. We use ECMP for load balancing. We generate traffic based on two realistic workloads in production: WebSearch [3] and DataMining [42]. Each sender sends messages in a Poisson flow, and the target loads for the fixed receiver range from 10% to 90%. K in ECN is 70 packets. The instantaneous marking threshold for ECNSharp is 80 μs, the persistent target threshold is 10 μs and the persistent interval is 150 μs. For CoDel, we set the interval to be 150 μs and the target to be 10 μs. The results are shown in Figure 13 and 14. Due to the space

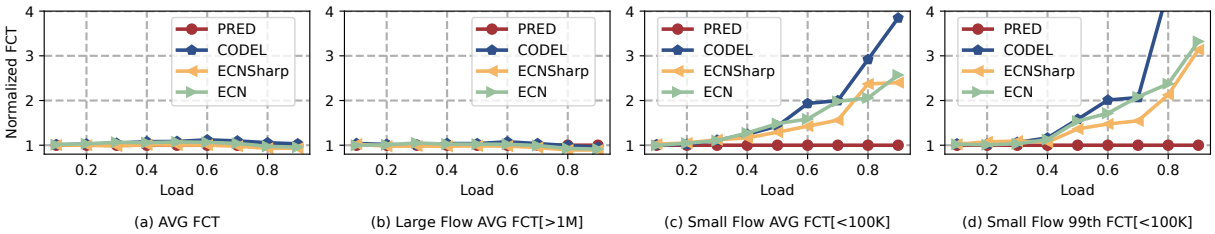


Figure 13: [Simulation] FCT statistics with different ECN algorithm.

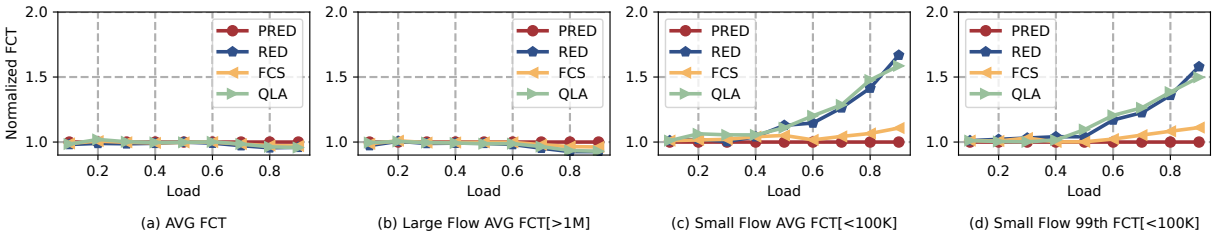
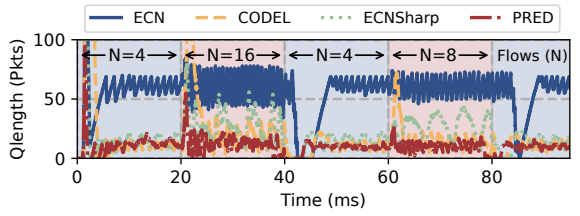
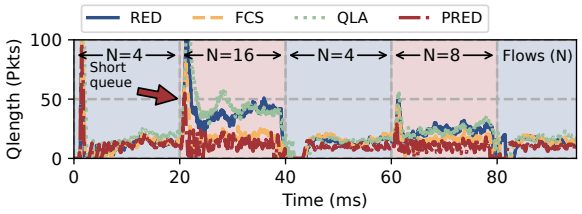


Figure 14: [Simulation] FCT statistics with different RED algorithm.



(a) Compare with ECN algorithm



(b) Compare with FCS and QLA

Figure 15: [Simulation] Queue Length.

limitation, We listed the DataMining results in Appendix D.2. All results have been normalized to FCT achieved by PRED.

Results: As shown in Figure 13, PRED' small flow 99th FCT (1.4 ms) has a reduction ranging from 68% (ECNSharp, 4.34 ms) to 80% (CODEL, 6.68 ms) compared to ECN algorithms at 90% load. For Large flows, there was a negligible increase in the FCT of PRED (96.8 ms), ranging from 4.6% (CODEL, 92.5 ms) to 12.5% (ECNSharp, 86 ms) compared to ECN algorithms at 90% load. This proves that PRED can outperform other ECN algorithms in real-workload scenarios. In Figure 14, PRED and FCS perform better than the static threshold RED. QLA alone does not improve performance. FCS alone is not optimal at high concurrency, and QLA should be combined to further reduce the FCT of small flows.

6.3.2 PRED Microscopic View

We want to know why PRED is better, so we conduct a comparative analysis through time sequence diagrams.

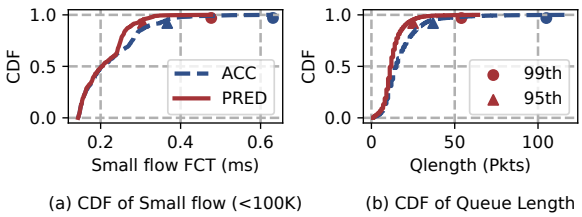
Setup: We use a simple 16-to-1 topology with 10 Gbps links, 16 servers are senders and 1 receiver. Other settings are similar to § 6.3.1. To provide a clearer illustration of how different schemes handle their queues, we sample the queue length of the bottleneck link every 10 μs. The number of concurrent flows is constantly changing.

Results: Figure 15 shows the queue changes of different algorithms at the bottleneck port. PRED effectively eliminates queue buildups by keeping the switch queue length 66% lower than that of the static threshold algorithms (from 25 packets to 15 packets). As can be seen in Figure 15(a), PRED gradually adjusts the queue length over time, making the queue lower than other algorithms. And since it is essentially a RED algorithm, probabilistic marking makes the queue more stable, and the vibration amplitude will be smaller. ECNsharp uses two ECN thresholds, but fixed adjustment makes the queue unstable as shown in Figure 15(a). In addition, PRED can converge faster when burst is encountered, resulting in less queue fluctuation.

Figure 15(b) compares FCS and QLA. From figure 15(b), FCS can converge rapidly but will keep the queue length stable at about 23 packets, while PRED will further reduce the queue length. Because QLA cannot adjust λ MIMD according to N, the convergence process is turbulent. Therefore, we can conclude that FCS and QLA need to cooperate with each other so that PRED can achieve the best performance.

6.3.3 Compared with ACC

ACC uses DRL to automatically adjust the RED parameter. DRL requires a lot of data and training to use. We compare PRED and ACC with a simple many-to-one incast scenario. The RED part of ACC uses NS3 simulation and the Learning part of ACC uses NS3-GYM [43]. We use only one agent to learn the traffic conditions in the congested port, reducing the amount of data to learn. We use Double DQN [23] to implement ACC. The **State** is 6 normalized values $(\bar{q}, R, R^m, \mathsf{minK}, \mathsf{maxK}, \mathsf{maxP})$. They are the queue length $(\bar{q})$, the port speed (R), the speed of marking ECN



(a) CDF of Small flow (<100K) (b) CDF of Queue Length

Figure 16: [Simulation] CDF of FCT and Queue length.

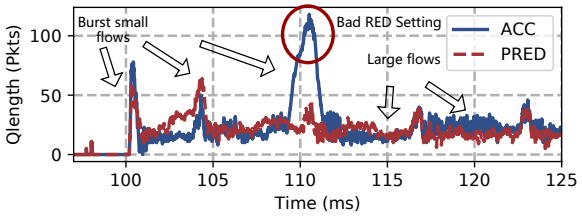


Figure 17: [Simulation] Qlength of ACC and PRED.

(R^m), and the current RED setting (minK, maxK, maxP). **Action** space dimension is $4 \times 3 \times 21$, i.e., $\text{minK}(2,4,8,16) \times \text{maxK}(20,50,100) \times \text{maxP}(\{1\%, j \times 5\%\}, \forall j \in [1,20])$. The **Reward** function is consistent with PRED's utility function. We train ACC for 200,000 epochs using the same trace for both training and testing. It's important to note that in our current testing, ACC encounters the same trace as during training. Even if we aim for deep reinforcement learning to over-fit the results, the stability of the outcomes cannot be guaranteed.

Setup: We use a simple 18-to-1 topology with 10 Gbps links, 18 servers are senders, and 1 receiver. Other settings are similar to § 6.3.1. We send flows from 18 senders to the receiver. The size of both large and small flows are generated based on two artificial workloads mentioned in § 3.1 (workload 1 (3-6 KB) for small flows and workload 2 (30-600 KB) for large flows). Both kinds of traffic are sent synchronously, and we measure the FCT of small flows (< 100 KB).

Results: Figure 16 shows the FCT and queue length of ACC and PRED in this particular scenario. As can be seen from Figure 16(a), FCT is similar at 50th, but PRED reduced 99th FCT by 34% compared to ACC (from 0.63 ms to 0.47 ms). From Figure 16(b), the tail of the queue length in ACC is also larger. The 95th percentile in the ACC is 37 packets, and the PRED is 25 packets; The 99th queue length ACC is 105 packets, while the PRED is 54 packets.

According to the above analysis, PRED was better than ACC in tail FCT and tail queue length. To further understand why PRED is better than ACC, we selected a time slice of the ACC runtime, shown in Figure 17. From the figure, the queue length convergence of PRED and ACC is similar because they share the same objective function. However, at around 110 ms, the ACC queue length significantly increases. The reason is that even after ACC is deployed, it still needs to have a random exploration probability from learning new network condition changes. Some bad RED Settings will be selected (around 110 ms in Figure 17), resulting in large queue lengths and high tail latency. In conclusion, the use of machine learning methods

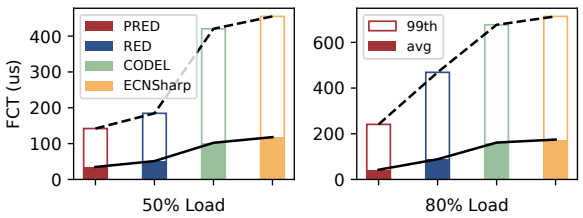


Figure 18: [Simulation] Small Flow FCT statistics with DataMining workload.

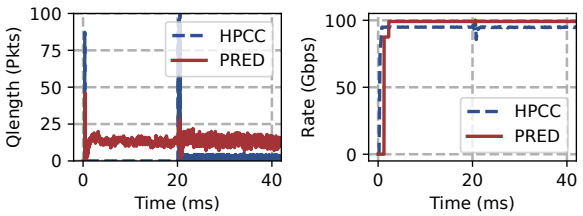


Figure 19: [Simulation] Comparison of PRED and HPCC.

needs to be carefully set up in the exploration scheme, The unreadable and untunable model will affect the tail latency. A white-box PRED can ensure that the queue converges to the appropriate position.

6.3.4 100 Gbps with 1000s Concurrent Flows

We also evaluate PRED on 100 Gbps networks and with thousands of concurrent flows.

Setup: The topology is the same as the one used in § 6.3.1 with a 128-host leaf-spine topology and 100 Gbps links. The latency of the link is 1 μs. On 128 hosts, randomly select 1000 senders, with an average of 7 senders started per host, and choose a fixed receiver. Each sender sends messages in a Poisson flow, and the target loads for the fixed receiver are 50% and 80% respectively. The DataMining [42] results are shown in Figure 18. Due to the space limitation, We listed the WebSearch results in Appendix D.3.

Results: As shown in Figure 18, PRED' small flow 99th FCT (240 μs) has a reduction ranging from 48% (RED, 468 μs) to 66% (CODEL, 713 μs) at 80% load. PRED' small flow average FCT (42 μs) has a reduction ranging from 52% (RED, 88 μs) to 75% (CODEL, 174 μs) at 80% load. This proves that PRED can outperform other ECN algorithms in 100 Gbps with 1000s concurrent flows.

6.3.5 Comparison of PRED and HPCC

We also compare PRED with HPCC [7] to show how PRED compares to the newer congestion control schemes. HPCC's convergence target is 95% bandwidth, so it can always keep the short queue length and is very friendly for short flows. HPCC needs to modify NICs and switches but PRED only needs to modify switches incrementally. Here we want to see how far PRED is from HPCC.

Setup: We use a simple 20-to-1 topology with 100 Gbps links, 20 servers are senders and 1 receiver. The latency of the link is 1 μs. The parameters are set as suggested in [7]. The NS-3 simulation implementation of HPCC refers to the

implementation in [7]. Starting at 0 ms, start two long flows. At 20 ms, 6 long flows are started concurrently.

Results: Figure 19 shows the queue length and throughput of PRED and HPCC. As shown in Figure 19, HPCC has consistently maintained a low queue length of almost zero and 93% throughput. The PRED queue length gradually converges from 20 to 15, and it is always full throughout. At 20 ms burst, the PRED queue fluctuates less, mainly because HPCC does not have a slow start process, so the queue changes greatly.

6.3.6 Packet Scheduler and Parameter Design Choice

We also compare PRED with TCN [38] and ECNsharp [15] to show how PRED works with arbitrary packet scheduler and analyze parameter design choice. Due to space limitations, we present the results in Appendix D.4.

7 Related Work

The most relevant work for PRED is ACC [17], which has been discussed in detail in this paper. Next, we will introduce some other related work.

Dynamically Adjust the RED Threshold. Since the RED algorithm was initially proposed, many proposals studied how to adjust the RED parameters dynamically [44–48]. Feng's ARED [44, 45] adjusts the marking/dropping probability, maxP, in RED to keep the average queue size stable between minK and maxK, in the form of MIMD. Floyd's ARED [46] adjusts maxP in AIMD way to make the queue length fluctuate around a certain value. STAQM [48] adjusts AQM parameters by modeling and parameterizing the RED algorithm and PI algorithm [49] to estimate network state according to network measurement and estimation. These art demonstrate the benefits of dynamic RED parameters. However, they only assure convergence of queue length, while the exact converged queue length must be manually set. As a result, their applicability in datacenter networks is limited.

Recently, there also exist some new algorithms [14–16] for ECN proposed in datacenters. BCC [16] dynamically adjusts the RED algorithm based on the global shared buffer usage. Port-level RED is used when utilization is low, and the shared buffer RED is used when the utilization is high. TCD [14] finds that in lossless networks, the queue has an undetermined state between congestion and non-congestion. It is critical to identify the undetermined state and notify the end host. These art found that ECN with a fixed threshold would be ineffective in datacenters and therefore set more than one state during threshold adjustment. However, they still use fixed thresholds that cannot automatically adjust based on traffic dynamics.

Buffer Sizing and AQM in Internet. The most well-known rule of buffer sizing showed that the minimum buffer size should be $C \times RTT/\sqrt{N}$ when there are a large number of N long-lived TCP Reno flows [50]. Existing new congestion controls such as BBR [29] also require new theories to set buffer sizes [51,52]. Bruce et al. [30] also discussed how to estimate the number of N on a router. The above studies inspired PRED's motivation to dynamically adjust RED Settings by counting traffic numbers in datacenters.

Most ECN-based datacenter congestion control algorithms [3, 53, 54] set the two thresholds of RED to the same value, $\mathsf{minK} = \mathsf{maxK} = K$. The ideal ECN marking threshold K is given as $K = \lambda \times C \times RTT$ [3, 38, 41, 54, 55]. Different congestion controls have different λ, TCP is 1 [54], DCTCP is 0.17 [55]. Through experiments in §3.2, we found that the relationship between single-value RED and N is that the amplitude of the queue increases with the increase of N. Therefore, PRED does not regulate based on single values.

MQ-ECN [41] first pointed out the drawbacks of existing ECN/RED implementations in the packet scheduling context. To adapt to the varying queue capacity caused by packet schedulers, TCN [38] proposed to use instantaneous sojourn time to mark packets. ECNsharp [15] inherits the merit of TCN but further tracks the persistent congestion state to reduce long-term queue buildups. PRED can make the steady-state queue consistent for different priorities/queues so that it can be well adapted to different packet schedulers.

Congestion Control for Datacenters. Most of the network congestion problems are solved by proposing new congestion control solutions. HULL [56] sacrifice throughput in exchange for low latency. It marks the ECN by calculating link utilization rather than queue length. PRED can use a scheme similar to the HULL, but the difficulty is that it requires the endpoint to use the packet pacing function to ensure throughput utilization. pFabric [57], BFC [9], and HPCC [7] rely on precise in-network state information of switches and update transmission rate for each flow. pFabric achieves near-optimal FCT by using (infinite) priority queues on switches. The BFC enables the switch queue to reach one-hop BDP through the backpressure rate control of per-flow and per-hop. HPCC adjusts the speed MIMD through the INT information of the switch. PCN [58], Homa [59] rely on the receiver to send credit packets to determine the sending rate of each flow. TIMELY [6] and Swift [8] are RTT-based schemes to adjust the flow rate at end host. These approaches achieve remarkable performance. However, they require changes to the network stack that are not easy to implement on older hardware devices.

8 Conclusion

This paper revisits the RED parameter configuration and proposes the design of PRED by capturing the key features of datacenter traffic dynamics in terms of flow concurrency and flow distributions. Adaptively and effectively, PRED achieves automatic adjustment of RED parameters stably. Through evaluations, we demonstrate that PRED is more stable than ACC [17], the state-of-the-art learning-based approach. Without any modifications at the end host, and with a more stable RED parameter configuration that adapts well to traffic dynamics, as the future work, we believe PRED has good potential to be deployed in modern production datacenters.

Acknowledgements

We are thankful to the anonymous NSDI reviewers and our shepherd, Vijay Chidambaram, for their constructive feedback. This work is supported in part by the China National Funds for Distinguished Young Scientists (No.62425201), the NSFC Projects (No.61932016, No.62132011, No.62221003 and No.62202473), and the CCF-Huawei Populus Euphratica Innovation Research Funding.

References

[1] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *IEEE MSST*, pages 1–10, 2010.

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[3] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *ACM SIGCOMM*, pages 63–74. ACM, 2010.

[4] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[5] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. *ACM SIGCOMM Computer Communication Review*, 45(4):523–536, 2015.

[6] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. *ACM SIGCOMM Computer Communication Review*, 45(4):537–550, 2015.

[7] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: high precision congestion control. In Jianping Wu and Wendy Hall, editors, *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019*, pages 44–58. ACM, 2019.

[8] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. Swift: Delay is simple and effective for congestion control in the datacenter. In *ACM SIGCOMM*, pages 514–528. ACM, 2020.

[9] Prateesh Goyal, Preey Shah, Kevin Zhao, Georgios Nikolaidis, Mohammad Alizadeh, and Thomas E. Anderson. Backpressure flow control. In *USENIX NSDI*, pages 779–805. USENIX Association, 2022.

[10] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve D. Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena E. Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 399–413. ACM, 2019.

[11] Abhishek Dhamija, Balasubramanian Madhavan, Hechao Li, Jie Meng, Shrikrishna Khare, Madhavi Rao, Lawrence Brakmo, Neil Spring, Prashanth Kannan, Srikanth Sundaresan, and Soudeh Ghorbani. A large-scale deployment of DCTCP. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 239–252, Santa Clara, CA, April 2024. USENIX Association.

[12] Bryce Cronkite-Ratcliff, Aran Bergman, Shay Vargaftik, Madhusudhan Ravi, Nick McKeown, Ittai Abraham, and Isaac Keslassy. Virtualized congestion control. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 230–243. ACM, 2016.

[13] Keqiang He, Eric Rozner, Kanak Agarwal, Yu Gu, Wes Felter, John Carter, and Aditya Akella. Ac/dc tcp: Virtual congestion control enforcement for datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 244–257. ACM, 2016.

[14] Yiran Zhang, Yifan Liu, Qingkai Meng, and Fengyuan Ren. Congestion detection in lossless networks. In *ACM SIGCOMM*, pages 370–383. ACM, 2021.

[15] Junxue Zhang, Wei Bai, and Kai Chen. Enabling ecn for datacenter networks with rtt variations. In *ACM CoNext*, pages 233–245. ACM, 2019.

[16] Wei Bai, Shuihai Hu, Kai Chen, Kun Tan, and Yongqiang Xiong. One more config is enough: Saving (dc) tcp for high-speed extremely shallow-buffered datacenters. *IEEE/ACM Transactions on Networking*, 29(2):489–502, 2020.

[17] Siyu Yan, Xiaoliang Wang, Xiaolong Zheng, Yinben Xia, Derui Liu, and Weishan Deng. Acc: Automatic ecn tuning for high-speed datacenter networks. In *ACM SIGCOMM*, pages 384–397. ACM, 2021.

[18] K Ramakrishnan, Sally Floyd, and D Black. Rfc3168: The addition of explicit congestion notification (ecn) to ip, 2001.

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

[20] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *ACM SIGCOMM computer communication review*, 45(4):183–197, 2015.

[21] Glenn Judd. Attaining the promise and avoiding the pitfalls of tcp in the datacenter. In *USENIX NSDI*, pages 145–157. USENIX Association, 2015.

[22] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, et al. Jupiter evolving: transforming google's datacenter network via optical circuit switches and software-defined networking. In *ACM SIGCOMM*, pages 66–85. ACM, 2022.

[23] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, volume 30, 2016.

[24] Tofino product family brochure. https://www.intel.com/content/dam/www/central-libraries/us/en/documents/tofino-product-family-brochure.pdf. 2022.

[25] The P4 Language Consortium. $p4_{16}$ language specification version 1.0.0. 2016.

[26] Network simulator 3. (2019). https://www.nsnam.org/. 2019.

[27] Yibo Zhu, Monia Ghobadi, Vishal Misra, and Jitendra Padhye. Ecn or delay: Lessons learnt from analysis of dcqcn and timely. In *ACM CoNext*, pages 313–327. ACM, 2016.

[28] Sally Floyd. Tcp and explicit congestion notification. *ACM SIGCOMM Computer Communication Review*, 24(5):8–23, 1994.

[29] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time. *ACM Queue*, 14(5):20–53, 2016.

[30] Bruce Spang and Nick McKeown. On estimating the number of flows. In *Stanford Workshop on Buffer Sizing*, 2019.

[31] Mo Dong, Qingxi Li, Doron Zarchy, P Brighten Godfrey, and Michael Schapira. Pcc: Re-architecting congestion control for consistent high performance. In *USENIX NSDI*, pages 395–408. USENIX Association, 2015.

[32] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. Pcc vivace: Online-learning congestion control. In *USENIX NSDI*, pages 343–356. USENIX Association, 2018.

[33] Tong Meng, Neta Rozen Schiff, P Brighten Godfrey, and Michael Schapira. Pcc proteus: Scavenger transport and beyond. In *ACM SIGCOMM*, pages 615–631. ACM, 2020.

[34] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[35] Dctcp in linux kernel. https://www.kernel.org/doc/html/latest/networking/dctcp.html. 2022.

[36] Dctcp in ns3 simulation. https://www.nsnam.org/docs/release/3.36/models/html/tcp.html?highlight=dctcp. 2022.

[37] Kathleen Nichols and Van Jacobson. Controlling queue delay. *Communications of the ACM*, 55(7):42–50, 2012.

[38] Wei Bai, Kai Chen, Li Chen, Changhoon Kim, and Haitao Wu. Enabling ecn over generic packet scheduling. In *ACM CoNext*, pages 191–204. ACM, 2016.

[39] iperf - the ultimate speed test tool for tcp, udp and sctp. https://iperf.fr/. 2022.

[40] Traffic generator. https://github.com/HKUST-SING/TrafficGenerator. 2022.

[41] Wei Bai, Li Chen, Kai Chen, and Haitao Wu. Enabling ecn in multi-service multi-queue data centers. In *USENIX NSDI*, pages 537–549. USENIX Association, 2016.

[42] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta.

Vl2: A scalable and flexible data center network. In *ACM SIGCOMM*, pages 51–62. ACM, 2009.

[43] Piotr Gawłowicz and Anatolij Zubow. ns-3 meets OpenAI Gym: The Playground for Machine Learning in Networking Research. In *ACM MSWiM*. ACM, 2019.

[44] Wuchang Feng, Dilip Kandlur, Debanjan Saha, and Kang Shin. Techniques for eliminating packet loss in congested tcp/ip networks. Technical report, Citeseer, 1997.

[45] W-C Feng, Dilip D Kandlur, Debanjan Saha, and Kang G Shin. A self-configuring red gateway. In *IEEE INFOCOM*, volume 3, pages 1320–1328, 1999.

[46] Sally Floyd, Ramakrishna Gummadi, Scott Shenker, et al. Adaptive red: An algorithm for increasing the robustness of red's active queue management, 2001.

[47] Srisankar Kunniyur and Rayadurgam Srikant. Analysis and design of an adaptive virtual queue (avq) algorithm for active queue management. *ACM SIGCOMM Computer Communication Review*, 31(4):123–134, 2001.

[48] Honggang Zhang, Don Towsley, CV Hollot, and Vishal Misra. A self-tuning structure for adaptation in tcp/aqm networks. In *ACM SIGMETRICS*, pages 302–303. ACM, 2003.

[49] Chris V Hollot, Vishal Misra, Don Towsley, and Wei-Bo Gong. On designing improved controllers for aqm routers supporting tcp flows. In *IEEE INFOCOM*, volume 3, pages 1726–1734, 2001.

[50] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. Sizing router buffers. *ACM SIGCOMM Computer Communication Review*, 34(4):281–292, 2004.

[51] Bruce Spang. Updating the theory of buffer sizing. *ACM SIGMETRICS Performance Evaluation Review*, 49(3):55–56, 2022.

[52] Tong Li, Kai Zheng, Ke Xu, Rahul Arvind Jadhav, Tao Xiong, Keith Winstein, and Kun Tan. TACK: improving wireless transport performance by taming acknowledgments. In Henning Schulzrinne and Vishal Misra, editors, *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, pages 15–30. ACM, 2020.

[53] Balajee Vamanan, Jahangir Hasan, and TN Vijaykumar. Deadline-aware datacenter tcp (d2tcp). *ACM SIGCOMM Computer Communication Review*, 42(4):115–126, 2012.

[54] Haitao Wu, Jiabo Ju, Guohan Lu, Chuanxiong Guo, Yongqiang Xiong, and Yongguang Zhang. Tuning ecn for data center networks. In *ACM CoNext*, pages 25–36. ACM, 2012.

[55] Mohammad Alizadeh, Adel Javanmard, and Balaji Prabhakar. Analysis of dctcp: stability, convergence, and fairness. *ACM SIGMETRICS Performance Evaluation Review*, 39(1):73–84, 2011.

[56] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *USENIX NSDI*, pages 253–266. USENIX Association, 2012.

[57] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 435–446. ACM, 2013.

[58] Wenxue Cheng, Kun Qian, Wanchun Jiang, Tong Zhang, and Fengyuan Ren. Re-architecting congestion management in lossless ethernet. In *USENIX NSDI*, pages 19–36. USENIX Association, 2020.

[59] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *ACM SIGCOMM*, pages 221–235. ACM, 2018.

A Problem Formulation

We take DCTCP as an example of the RED parameter breakdown based on a fluid model, as in paper [3, 5]. Table 1 summarizes the main notations in this paper. Note that we can also analyze other sources (e.g., DCQCN and TCP) in a similar way [27, 28], since the conclusions remain similar, we omit them here for conciseness.

In a fluid model, we consider N long-lived flows traversing a single bottleneck link with capacity C. The following nonlinear, delay-differential equations describe the dynamics of the window size $w(t)$, the DCTCP's estimated congestion degree $\alpha(t)$, and the queue size $q(t)$ at the switch:

$$\frac{dw}{dt} = \frac{1}{RTT(t)} - \frac{w(t)\alpha(t)}{2RTT(t)} p(t - RTT^*) \tag{2}$$

$$\frac{d\alpha}{dt} = \frac{g}{RTT(t)} (p(t - RTT^*) - \alpha(t)) \tag{3}$$

$$\frac{dq}{dt} = N\frac{w(t)}{RTT(t)} - C \tag{4}$$

Variable	Description
w	Window size
N	Flow concurrency level
α	Estimated congestion degree of DCTCP
g	DCTCP's parameter
t	Time
q	Bottleneck queue length
RTT	Round-trip time (RTT)
C	Bottleneck link capacity
d	Propagation delay

Table 1: Variables of fluid model.

Here $p(t)$ indicates the RED (ECN) marking process at the switch and is given by:

$$p(t) = \begin{cases} 1, & q(t) > \mathsf{maxK} \\ \frac{q(t)-\mathsf{minK}}{\mathsf{maxK}-\mathsf{minK}}\mathsf{maxP}, & \mathsf{minK} < q(t) <= \mathsf{maxK} \\ 0, & q(t) <= \mathsf{minK} \end{cases} \quad (5)$$

and $RTT(t) = d + q(t)/C$ is the RTT, where d is the propagation delay (assumed to be equal for all flows), and $q(t)/C$ is computed as the queueing delay.

Equations (2) and (3) describe the congestion window adjusting scheme of DCTCP at the source, and Equation (4) describes the queuing process at the switch. The source and the switch are coupled through the packet marking process $p(t)$. This feedback delay is approximately a fixed value $RTT^* = d + q_{avg}/C$.

Equation (2) models the window evolution and consists of the standard additive increase term, $1/RTT(t)$, and a multiplicative decrease term, $-w(t)\alpha(t)/2RTT(t)$. The latter term models the source's reduction of window size by a factor $\alpha(t)/2$ when packets are marked. Equation (3) is a continuous approximation of DCTCP's estimated congestion degree. Equation (4) models the queue evolution: $N\frac{w(t)}{RTT(t)}$ is the network input rate and C is the service rate.

By setting the LHS (left hand side) of Equations (2)-(4) to zero, we see that any fixed points of the DCTCP (if they exist) must satisfy:

$$1 - \frac{\dot{W}\dot{\alpha}}{2}\dot{p} = 0, \dot{p} - \dot{\alpha} = 0, N\frac{\dot{W}}{d + \frac{\dot{q}}{C}} - C = 0 \quad (6)$$

At any of the fixed points, we assume the value of p is $\dot{p}$, which is shared by all flows, and we can get:

$$\dot{p} = \frac{\dot{q} - \mathsf{minK}}{\mathsf{maxK} - \mathsf{minK}}\mathsf{maxP} = \lambda(\dot{q} - \mathsf{minK}) \quad (7)$$

Combining Equations (6) and (7), we eliminate the variable $\dot{p}$, $\dot{\alpha}$ and $\dot{W}$. After simplification, we see that the value of $\dot{q}$ is determined by:

$$(\dot{q} - \mathsf{minK})^2(\dot{q} + Cd) = \frac{2N}{(\frac{\mathsf{maxP}}{\mathsf{maxK}-\mathsf{minK}})^2} = \frac{2N}{\lambda^2} \quad (8)$$

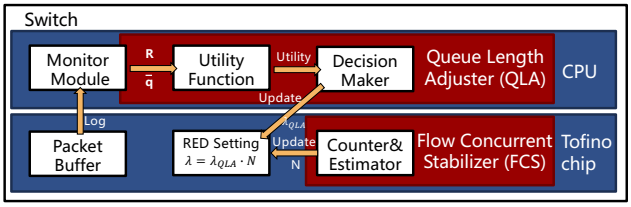


Figure 20: Hardware architecture of PRED on Tofino.

From Equation (8), we can see that the steady-state $\dot{q}$ is only related to three factors. The first factor is related to bottleneck link C and feedback delay d. Once the network topology is built, this factor will be fixed. The second factor is the flow concurrency level N, and the larger N is, the larger $\dot{q}$ is. Due to the dramatic change of the flow concurrency N, the steady-state $\dot{q}$ will fluctuate sharply, which will affect the stability of the network. The last but most important factor is the settings of the RED parameter, i.e., λ and minK. We thus infer that the essence of configuring RED parameters is to adjust the steady-state queue length.

B Additional Discussion

Highly dynamic traffic pattern. If the network is highly dynamic, PRED will tend to be stable, rather than a faster convergence or faster traffic prediction. In a highly dynamic network, the QLA will filter out the noise, and PRED will adjust the parameters only when there is a positive reward in the statistics within T_{QLA} time. § D.5 shows that PRED requires at least 200 ms to fully adapt to a purely short-flow load from a purely high-flow load.

Overestimating N or underestimating N. If the true N exceeds the capacity of the estimator, then N will be underestimated; If a lot of bursts come, then N will be overestimated. In this case, the FCS flow estimate will be biased, but the noisy N will still be better than the static threshold. As can be seen in § 6.3.2, PRED can quickly respond to changes in N, resulting in faster congestion adjustment and feedback even if inaccurate. In addition, QLA can also compensate for long-term estimation inaccuracies, so that PRED still keeps the switch statistically high performance.

C Additional Implementation

Figure 20 shows the hardware architecture of PRED on the Tofino switch.

C.1 Data Plane Modification

Counter & Estimator Module. PRED needs to know how many flows are passing through the port of the switch at a time interval. As described in § 4.1, switches only need a simple bitmap and counters to implement this module. However,

each stage of the switch can operate on only one register, not an array of bitmap registers. Thus, the switch cannot bulk reset the bitmap registers periodically on the data plane. The control plane can reset the registers of the data plane, but it cannot be used because of the high API delay (>1 ms).

To solve this problem, PRED uses a time interval sequence counter and a write-after hashing operation to calculate the number of concurrent flows. Algorithm 1 shows the FCS data plane processing logic. When a packet $\mathcal{P}$ arrives, the switch first checks whether the current time *now* has been more than an interval T_{FCS} since the last update T_{start}(line 1). If the time is over this interval, some registers need to be updated and reset (lines 2-5). If not, the packet $\mathcal{P}$ needs to be checked to see whether it belongs to a new flow (lines 7-11).

We first analyze the reset process after the timeout (lines 2-5). T_{start}, n_{last}, n_{now}, $Interval_{seq}$ and $Bitmap$ are all Registers. T_{start} is the beginning of the interval (line 2) and needs to be set to the current time *now* after the timeout occurs. n_{last} is the flow number collected in the last interval (line 3), which should be set to n_{now} after the timeout. n_{now} is the flow number collected in the current interval (line 4), which needs to be set to 0 after the timeout. $Interval_{seq}$ is the serial number of the time interval (line 5), which is used to distinguish different time intervals. $Bitmap$ is a hash table of length $2^{IndexSize}$.

Next, when there's no timeout (lines 7-11), we analyze the most critical process of counting flow numbers. The packet $\mathcal{P}$'s flow hash H_{new} is computed by hashing over the packet's 5-tuple and interval sequence number $Interval_{seq}$ (line 7). Then the last *IndexSize* bits of the hash value is used as the storage index S_{id} (line 8). The most crucial step is to read the stored flow hash H_{old} in the S_{id} position in the hash table *bitmap* and overwrite the H_{new} in the corresponding position (line 9). Then PRED checks whether the values of H_{new} and H_{old} are equal and if they are not, then the new flow is detected (lines 10-11). The reason for using interval sequence number $Interval_{seq}$ for packet hashing is that we want flows across different time intervals to be recounted at different time intervals. To this end, the same flow must have different hash values at different time intervals. With the above steps, PRED does not need to bulk reset the value in the bitmap register periodically on the data plane because the new hash value is overwritten each time. Two conflicts may affect the flow count, i.e., hash conflict (different hash parameters but same hash value) and position conflict (different hash value but same S_{id}). The hash conflict causes conflicting flows to be viewed as one flow, and the position conflict causes conflicting flows to alternately overwrite the same register, bringing unusually high flow counts. However, the chances of the two conflicts are low, and thus they have little effect on the performance.

RED Setting Module. Because the P4 switch has no direct division to use for the RED Setting Module, here PRED programs the RED algorithm into the PRED table (line 13). The

Algorithm 1 Data Plane Processing Logic.

1: **if** $now - T_{start} > T_{FCS}$ **then**
2: $\quad \mathsf{UpdateReg}_{T_{start}}(now)$ ▷ Next time interval
3: $\quad \mathsf{UpdateReg}_{n_{last}}(n_{now})$ ▷ Record the last interval flow number
4: $\quad \mathsf{UpdateReg}_{n_{now}}(0)$ ▷ Reset the current interval flow number
5: $\quad \mathsf{UpdateReg}_{Interval_{seq}}(+1)$ ▷ Update the interval sequence
6: **else**
7: $\quad H_{new} \leftarrow \mathsf{Hash}(Packet5Tuple, Interval_{seq})$ ▷ Derive flow hash from packet 5-tuple and interval sequence
8: $\quad \mathsf{S_{id}} \leftarrow H_{new}[IndexSize : 0]$ ▷ Derive storage index
9: $\quad H_{old} \leftarrow \mathsf{Read\&UpdateReg}_{Bitmap}(\mathsf{S_{id}}, H_{new})$ ▷ Get the old hash value from the bitmap and set the new value
10: $\quad$ **if** $H_{new} != H_{old}$ **then** ▷ New flow is detected
11: $\quad\quad \mathsf{UpdateReg}_{n_{now}}(+1)$
12: $F_{Num} \leftarrow MAX(n_{last}, n_{now})$
13: $Prob \leftarrow ReadTable_{PRED}(q, F_{Num})$
14: **if** $Prob > Random()$ **then**
15: $\quad MarkECN(\mathcal{P})$

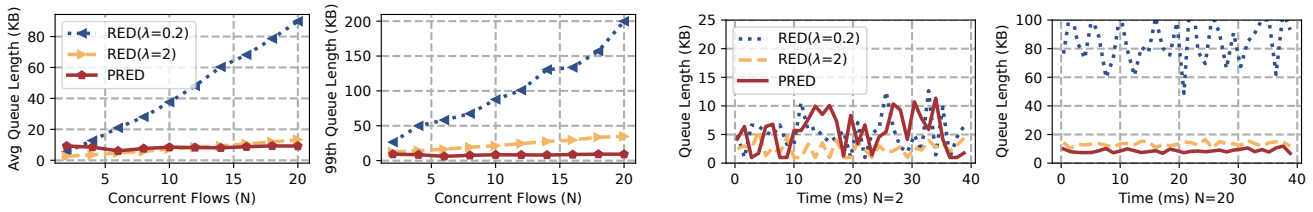
table's key is the queue length and the flow number, and the table's value is the marking probability. After obtaining the marking probability, PRED compares it with the random value. If the marking probability is greater than the random value, ECN labeling will be tagged (lines 14-15).

Packet Buffer Module. The packet buffer module provides some queue information to the control plane, including $\bar{q}$ and R. R can be obtained directly from switch statistics. The instantaneous queue length cannot match the performance of the switch. Therefore, PRED samples the queue length multiple times, sum it and saves it in the register. The control plane can read the register to get the sum value and the sampling times to infer the average queue length.

C.2 Control Plane Modification

The QLA module needs at least 6 additional stages in the Tofino switch if it is to be deployed in the data plane. We have implemented PRED where both QLA and FCS are deployed on the data plane, but a total of 17 stages are needed. In § 4.2, we introduced that the QLA module needs to save four utility function values to judge the adjustment of λ_{QLA}. These operations require at least 3 stages, two for reading/writing the register and one for comparison operation. Since the RED table requires additional Key entries to identify the λ_{QLA}, the additional stages need at least 3 stages. For pure FCS modules, 11 stages have been used, so the existing Tofino switches do not support deploying both modules to the data plane at the same time. However, the new Tofino 2 and Tofino 3 switches support 20 stages [24], so the QLA and FCS modules can be deployed in the data plane at the same time.

In this paper, we deploy the QLA module on the control plane. The Monitor Module subscribes raw data from the



(a) Different flow concurrency levels (b) Breakdown

Figure 21: [Simulation] PRED performance with DCQCN.

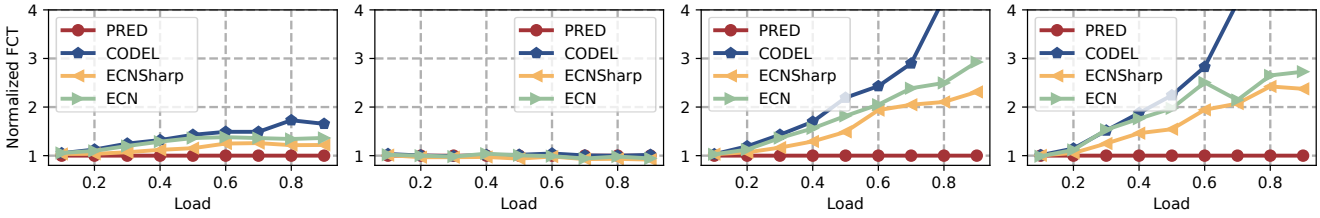


Figure 22: [Simulation] FCT statistics with different ECN algorithm. All are normalized to the results achieved by PRED (DataMining Workload).

Register for features analysis, including the total bytes sent and egress queue depth. In detail, at each time interval T_{QLA}, the Monitor Module achieves the subscribed data from forwarding chips's registers. Then, the utility function calculates the current utility value and passes it to the Decision Maker. The control module makes a judgment every 4 intervals T_{QLA} and sets the new RED setting to the data plane.

Discussion. There is a problem with implementing the QLA module in the control plane. Reading the register in the control plane takes 5 to 12 ms, and updating the Red table takes 80 to 110 ms. Therefore T_{QLA} is at least 200 ms due to hardware limitations. We discuss the setting of T_{QLA} in § D.5. The larger T_{QLA} is, the slower convergence will be. Therefore, if the implementation is upon Tofino 1, PRED has a slower convergence rate. However, we believe that if implemented upon Tofino 2 or Tofino 3, this problem would not occur.

D Additional Simulation Results

D.1 PRED with DCQCN

Setup: We use a simple 20-to-1 topology with 40 Gbps links, 20 servers are senders, and 1 receiver. The latency of the link is 2 μs. The parameters are set as suggested in [5, 27]. The NS-3 simulation implementation of DCQCN refers to the implementation in [27].

Results: Figure 21 shows the PRED performance with DCQCN. Figure 21(a) shows the average queue length and 99th queue length as N increases. The queue length for both RED (0.2) and RED (2) increases with N, and PRED avoids this problem. Figure 21(b) shows the time sequence diagram when N=2 and N=20. PRED at low concurrency (N=2), queue length changes close to RED (0.2) to ensure throughput. At high concurrency (N=20), PRED is close to RED (2) to ensure low latency. To sum up, PRED can also cooperate well with DCQCN.

D.2 Large Sacle Simulation in DataMing workload

Figure 22 is the result under the DataMining workload. For details, refer to § 6.3.1. As shown in Figure 22, PRED' small flow 99th FCT (0.76 ms) has a reduction ranging from 58% (ECNSharp, 1.83 ms) to 86% (CODEL, 5.28 ms) compared to ECN algorithms at 90% load. For Large flows, there was a negligible increase in the FCT of PRED (44 ms) ranging from 6.3% (ECN, 41.36 ms) to 8% (ECNsharp, 40 ms) compared to ECN algorithms at 90% load. This proves that PRED can outperform other ECN algorithms under the DataMining workload.

D.3 100 Gbps Large Sacle Simulation in WebSearch workload

Figure 23 is the result under the WebSearch workload. For details, refer to § 6.3.4. As shown in Figure 23, PRED' small flow 99th FCT (142 μs) has a reduction ranging from 27% (RED, 171 μs) to 59% (CODEL, 350 μs) at 80% load. PRED' small flow average FCT (29 μs) has a reduction ranging from 52% (RED, 61 μs) to 75% (CODEL, 117 μs) at 80% load.

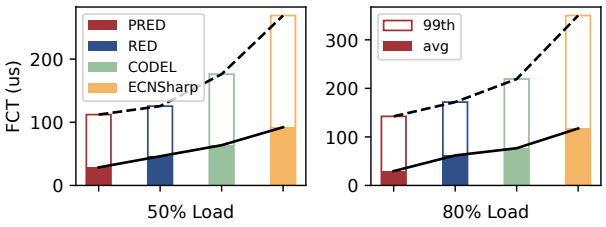
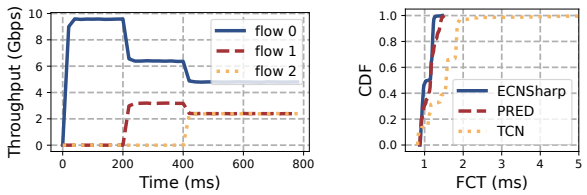


Figure 23: [Simulation] Small Flow FCT statistics with WebSearch workload.



(a) Flow throughput (b) FCT of small flows

Figure 24: [Simulations] PRED with packet schedulers.

This proves that PRED can outperform other ECN algorithms in 100 Gbps with 1000s concurrent flows.

D.4 Packet Scheduler

Setup: 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 it into queue/service 2, and finally from sender 3 into queue/service 3. We also randomly start small flows (the size is from 3 KB to 60 KB) from the rest of the senders to probe the queue occupancies. We set the TCN marking threshold to 150 μs to avoid throughput loss. The other settings are identical to §6.3.1.

Results: Figure 24(a) shows the throughput of flows achieved by PRED. We observe at the beginning that only queue 1 is active, flow 1 achieves around 9.6 Gbps. After flow 2 starts, queue 2 becomes active, and flow 1 achieves 6.42 Gbps while flow 2 achieves 3.18 Gbps. Finally, when three queues all become active, flow 1, flow 2 and flow 3 achieve around 4.82 Gbps, 2.40 Gbps and 2.40 Gbps goodput, respectively, which strictly preserves the packet scheduling policy.

We also measure the FCT of small flows among all queues, and the results are shown in Figure 24(b). PRED is similar to ECNsharp and superior to TCN in small flow. In summary, PRED can strictly preserve the packet scheduling policy with multiple queues/classes.

D.5 Parameter Design Choice

We focus here on the two most important time parameters T_{FCS} and T_{QLA} in PRED, and we will discuss the other parameters in future work. We analyzed how to set these in § 4, and

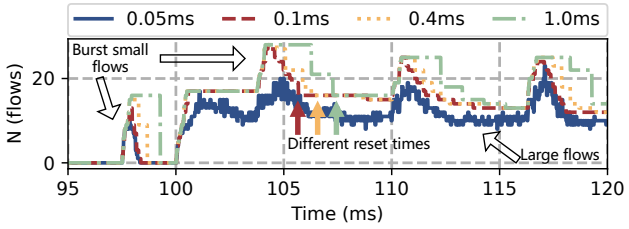


Figure 25: [Simulation] N Estimator of Different T_{FCS}. (RTT is 0.08 ms)

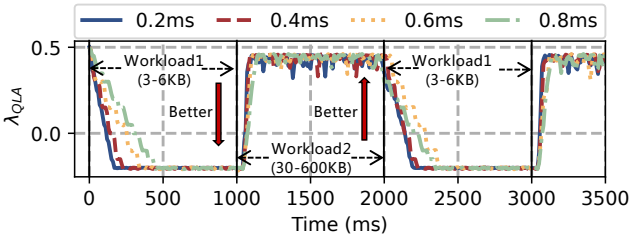


Figure 26: [Simulation] λ_{QLA} of Different T_{QLA}.

we verified these by experiment in this section.

Setup: The topology, parameter settings, and traffic patterns are all similar to the previous subsection.

Results: Figure 25 shows the estimation of flow number under different T_{FCS}. The burst state and the stationary state of the large flows both have 18 flows. When burst flows and large flows are concurrent, there should be 36 flows. From figure 25, when $T_{FCS} < RTT$, PRED cannot accurately estimate the correct N. Because all the traffic does not pass through the port completely. When $T_{FCS} > RTT$, PRED can estimate the correct N, but with the increase of T_{FCS}, the reset time of N is also increasing. So T_{FCS} should be slightly larger than RTT, and our rule of thumb is 1.25RTT.

Figure 26 shows the convergence of λ_{QLA} for different T_{QLA}. The workload 1 (3-6 KB) ranges from 0 to 1,000 ms and from 2,000 to 3,000 ms. Workload 2 (30-600 KB) is 1,000-2,000 ms. In workload 1, all flows are small, so the smaller the λ, the smaller the FCT. Therefore, it can be seen that PRED can rapidly reduce λ when 0-500 ms, while it can rapidly increase λ when 1,000 ms. And the smaller T_{QLA} is, the faster it converges, but the more unstable. Therefore, T_{QLA} is recommended to be 5RTT.

D.6 Range of PRED Adjustment Capabilities

We tested the range of PRED adjustment capabilities using Testbed. As shown in Figure 27, the throughput and queue length change with the number of concurrent N. The blue line indicates the parameter setting with the minimum λ and the lowest ECN probability, and we can see that its queue length is large and its throughput is high.

The red line indicates the parameter setting with the maximum λ and the maximum ECN marking probability. It can be seen that when $N < 32$, there is a loss in throughput and

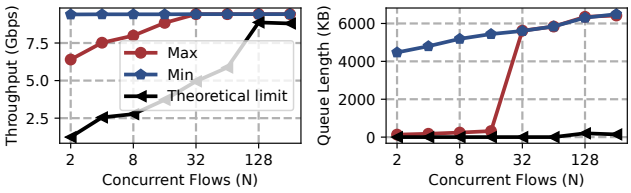


Figure 27: [Testbed] Range of PRED adjustment capabilities.

the queue length is small. When $N > 32$, the parameter Settings in the adjustment range cannot limit the stability of the queue. This means that PRED limits the number of concurrent requests to no more than 32 within the range of optional parameters. However, in practice, there are not that many concurrent long flows in the datacenter [3], and the range of PRED is sufficient.

The black line represents per-packet ECN, that is, every packet that enters the switch is marked with an ECN. It can be seen that the adjustment range of the black line is very large. However, the adjustment granularity is too large, and the change of very small parameters will lead to drastic changes in the queue and throughput, so the adjustment range of PRED is only between the red line and the blue line.