EasyChair Preprint

№ 3263

Protocol-Independent Service Queue Isolation for Multi-Queue Data Centers

Gyuyeong Kim and Wonjun Lee

April 26, 2020

Protocol-Independent Service Queue Isolation for Multi-Queue Data Centers

Gyuyeong Kim and Wonjun Lee
Network and Security Research Lab.
School of Cybersecurity
Korea University, Seoul, Republic of Korea
Email: {gykim08,wlee}@korea.ac.kr

***Abstract*—To isolate service queues in switch ports, recent solutions leverage the power of Explicit Congestion Notification (ECN). However, this causes a fundamental dependency on ECN-based transport protocols, making it hard to use generic transport protocols. To this end, we design DynaQ, a protocol-independent multi-queue management solution that enables service queue isolation with generic transport protocols. DynaQ dynamically adjusts the packet dropping threshold of service queues. Our solution is inexpensive to implement on hardware. Through extensive testbed experiments and large-scale simulations, we show that compared to alternative schemes, DynaQ is the only solution that achieves work-conserving weighted fair sharing and low latency without protocol dependency.**

***Keywords*-service queue isolation; buffer management; data center networks**

I. Introduction

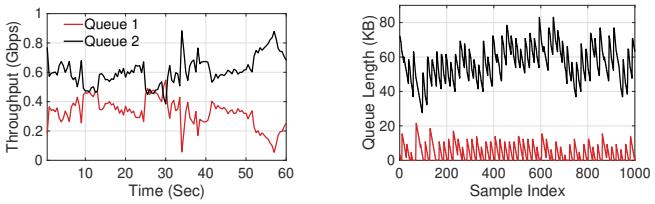
Data centers are shared by many services having diverse network performance requirements. To provide differentiated performance, the operator groups services into different traffic classes, and maps the classes into service queues in a switch port [1, 2]. The switch enforces network policy across the queues through packet schedulers like strict priority queueing (SPQ) and weighted round-robin (WRR). Meanwhile, the port buffer is shared among service queues in a best-effort manner. In this regime, it is difficult to isolate service queues because a queue with many flows can monopolize the buffer regardless of the allocated size. If a queue cannot occupy enough buffer space, it cannot achieve its fair share rate as well.

Recent solutions [1]–[3] leverage Explicit Congestion Notification (ECN), which maintains the maximum buffer occupancy around the ECN marking threshold K. Unfortunately, the existing solutions have a fundamental dependency on ECN-based transport protocols since ECN requires the cooperation between end-hosts and switches. This protocol dependency is undesirable because the end-hosts cannot use generic transport protocols. The recent advance in transport protocols shows that non-ECN protocols can outperform ECN-based protocols using different congestion signals (e.g. In-band network telemetry [4], credit [5], and network delay [6, 7]). These works are motivated by the drawbacks of ECN like coarse-grained congestion signaling and slow convergence time. Furthermore, in many multi-tenant environments, it is hard to enforce a specific transport protocol to virtual machines (VMs) and bare-metal (BM) servers because tenants own the network stack.

In this context, we ask the following question: *how to isolate service queues in switch ports without dependency on transport protocols?* We answer the question by presenting DynaQ, a protocol-independent multi-queue management scheme. DynaQ enables service queue isolation with generic transport protocols. The best-effort scheme and PQL provide us the following design guideline. First, to be work-conserving, a service queue should be able to occupy the buffer larger than or equal to the BDP. Second, to achieve weighted fair sharing, the switch should guarantee the buffer as much as the weighted BDP to a service queue. Third, to achieve the requirements simultaneously, the port buffer should be shared dynamically. Based on the guideline, DynaQ adjusts the packet dropping threshold of service queues dynamically so that a queue can occupy the buffer up to the port buffer size but does not take the buffer of unsatisfied active queues.

DynaQ is simple and can be implemented on hardware inexpensively with up to 7 clock cycles. The overhead is relatively small because switch ASICs require at least hundreds of clock cycles to process a packet. For example, Broadcom Trident 3 ASIC consumes at least 800 clock cycles to process a packet. We also discuss how DynaQ can be implemented on a programmable switching chip [8]. We implement a software prototype of DynaQ as a Linux qdisc module to compare DynaQ with various solutions.

We build a small-scale testbed with 5 servers connected to a server-emulated switch supporting 8×1GbE ports with two Intel I350-T4 NICs. We conduct extensive experiments to validate the efficiency of DynaQ. Our results show that DynaQ provides work-conserving weighted fair sharing regardless of the number of active queues, the number of competing flows, and different per-queue weights. Using multiple end-hosts with different transport protocols, we show that DynaQ works well with generic protocols as well. In addition, DynaQ outperforms compared schemes in the FCT for small and large flows. To complement the testbed experiments, we also perform large-scale simulations using ns-2.With link capacities of 10Gbps and 100Gbps, we demonstrate that DynaQ can preserve work-conserving weighted fair sharing in high-speed data center networks.



(a) Per-Queue Throughput (b) Queue Length

Figure 1. Violated fair sharing by unfair buffer occupancy.

II. Background and Motivation

A. Impact of Buffers on Bandwidth Sharing

Modern services have diverse network performance requirements. For example, web search requires low latency and data backup demands high throughput. To provide differentiated network performance based on network policy, the operator often leverages multiple service queues in a switch port. In multi-queue environment, the operator classifies services into different traffic classes. These classes are mapped into different service queues, and the queues are scheduled by weighted fair packet schedulers like WRR and deficit round robin (DRR) [1, 3, 9]. The operator also uses SPQ to prioritize latency-sensitive flows [2]. Shared SPQ queues have the higher priority and the other dedicated queues with fair schedulers have the lowest priority.

Unfortunately, network policy can be violated in spite of packet scheduling. To saturate the weighted bottleneck capacity, queue i requires the weighted Bandwidth Delay Product (BDP) that $WBDP_i = C \times RTT \times \frac{w_i}{\sum w}$ where RTT is the base RTT, C and w_i denote the link capacity and the weight of queue i, respectively. However, because the port buffer is shared among service queues in a best-effort manner, a service queue may not obtain enough buffer as much as the weighted BDP due to aggressive service queues. To show this, we conduct an experiment on a small-scale testbed consisting of 5 servers connected to a 1GbE server-emulated switch having a 85KB of buffer per port. In the switch, we configure DRR with equal weights. We have one receiver and 4 senders. Three senders are mapped into service queue 2 while the other sender belongs to service queue 1. We start 8 flows from each of the senders at the same time and measure per-queue throughput for 60 seconds every 0.5 second. We also record every queue length evolution and obtain 1K sequential samples at random time. Fig. 1 shows the results. Since the queues have the same weight, bandwidth should be shared equally. However, queue 1 cannot achieve its fair share rate because queue 1 cannot occupy the buffer larger than $WBDP_1$ due to a large arrival rate of queue 2.

B. Why Protocol Dependency Matters?

Existing solutions [1]–[3] leverage ECN to isolate service queues. The solutions have an underlying assumption: all end-hosts use ECN-based transport protocols. This is because ECN requires both ECN-enabled end-hosts and switches. The assumption implies that the existing solutions have a dependency on ECN-based transport protocols.

This is undesirable because the end-host network stack cannot adapt to the advance in transport protocols. We have witnessed the emergence of many non-ECN transport protocols, which have better performance than ECN-based transport protocols. For example, HPCC [4] leverages in-band network telemetry available in emerging switch ASICs because ECN does not know exactly how to adjust sending rates and causes a trade-off between latency and throughput with ECN marking threshold [10]. ExpressPass [5] shows that credit packets are better congestion signals than ECN in terms of fairness and convergence time. There also exist delay-based protocols like DX [7] and TIMELY [6], which are motivated by that ECN cannot inform the extent of congestion quickly. We believe that transport protocols will continue to evolve and their congestion signals will not be limited to ECN. The assumption also does not hold in many multi-tenant environments. VMs and BM servers are the major components in multi-tenant data centers, and their network stack is owned by tenants, not the network operator. Therefore, it is hard to enforce a specific transport protocol to all end-hosts.

C. Related Work

One may wonder whether modifying the schemes to drop packets instead of ECN marking can solve the problem. However, we find that simple changes are not enough for the following reasons.

MQ-ECN [1] determines a ECN marking threshold of queue i as following that $K_i = \min(\frac{quantum_i}{T_{round}}, C) \times RTT \times \lambda$ where $quantum_i$ is the weight/quantum of queue i and λ is a coefficient for transport protocols. T_{round} indicates the estimated total time to serve all queues once. The key drawback of MQ-ECN is that the solution relies on the concept of "round" with T_{round}. This means that MQ-ECN does not support packet schedulers like SPQ. Supporting SPQ is crucial to latency-sensitive services since it can accelerate small flows. Therefore, even we change the scheme to drop packets, the solution does not achieve our design goals since it fails to provide low latency.

TCN [2] uses the packet sojourn time as the threshold metric instead of queue length. Since the packet sojourn time can be calculated after passing through the queue, the solution performs dequeue marking. The standard ECN marking threshold is given by $T = RTT \times \lambda$. If we change TCN to drop the packet when the packet sojourn time exceeds T, we should drop the just dequeued packet. However, packet dropping at dequeue causes idle time on the link. This seriously degrades the effective throughput. In addition, dropping the buffered packet increases the FCT

as much as the packet sojourn time in addition to the retransmission timeout (RTO).

PMSB [3] provides both generic packet schedulers and early congestion notification, which MQ-ECN and TCN do not support, respectively. PMSB only marks packets when per-port ECN marking and per-queue ECN marking conditions are met at the same time where the port ECN marking threshold is given by $K = C \times RTT \times \lambda$. The per-queue ECN marking threshold for queue i is given by $K_i = \frac{w_i}{\sum w} C \times RTT \times \lambda$. Since $K_i \leq K$, the dropping version of PMSB is similar to PQL, which is supported in some production switches.

PQL assigns a static buffer size to a service queue. Therefore, each queue can enjoy its fair share regardless of other queues. However, PQL is not work-conserving because the amount of buffer that a single queue can occupy is limited to the assigned quota. Therefore, when few queues are active, the link capacity can be underutilized since the aggregate buffer occupancy can be less than the BDP. One might argue that assigning a buffer of the BDP to all service queues can solve the problem. Unfortunately, the on-chip SRAM buffer is a scarce resource in switches [11]. Therefore, we do not have enough buffers to reserve a buffer size as much as the BDP for all service queues.

We also discuss shared buffer management solutions like the dynamic threshold algorithm. Many switches allow a single port to occupy many buffers. However, even we allocate a large buffer size to a port, bandwidth cannot be shared fairly since aggressive queues eventually fill up the buffer. It also harms per-port fairness by taking excessive buffers that can be assigned to the other ports. Meanwhile, there exist deep buffered switches with an external large DRAM buffer. Unfortunately, the deep buffer switches have low switching throughput and insufficient per-packet processing delay to satisfy the requirements of modern user-facing applications.

BarberQ [12] addresses a similar problem by packet eviction. Packet eviction is an effective technique to absorb latency-sensitive microbursts. However, for service-queue isolation, it is enough to use packet dropping. pFabric [9] and QJump [13] also use multiple queues, but these works aim at minimizing the FCT of small flows, not isolating service queues.

III. DynaQ Design

A. Design goals

Our goal is *to design a multi-queue management solution that enables service queue isolation in switch ports over generic transport protocols.* We stipulate that a good solution should satisfy the following requirements *simultaneously*:

- **Protocol independence:** A solution should not be tied to a specific transport protocol.
- **Work conservation:** A solution should be able to utilize the whole link capacity regardless any time.
- **Weighted fair sharing:** A solution should strictly preserves weighted fair sharing among service queues any time regardless of traffic dynamics.
- **Low latency:** A solution should support arbitrary packet schedulers, especially SPQ, to minimize the FCT of small flows.
- **Practicality:** A solution should be inexpensive to implement on hardware.

B. Mechanisms

1) Basic Idea: DynaQ is the first protocol-independent multi-queue management scheme that satisfies the above requirements simultaneously. We observe that the best-effort scheme and PQL provide the following design guideline that: 1) to utilize the bottleneck link capacity fully, a service queue must be able to occupy the buffer larger than or equal to the BDP if there is free space in the port buffer; 2) to share bandwidth fairly while respecting different weights of service queues, a service queue must be able to occupy buffer space larger than or equal to the weighted BDP regardless of other service queues; 3) to guarantee the weighted fair share rate and sustain high link utilization at the same time, the switch should manage the port buffer dynamically among service queues. Without dynamic multi-queue management, we can meet only one of the two requirements at a time.

Following the above design guideline, DynaQ allows a single service queue to occupy free buffer space in the port, but prevents the service queue from taking the buffer of unsatisfied active queues[1]. Table I summarizes mathematical notations used to describe our work. To realize the idea, DynaQ assigns a packet dropping threshold T_i for each service queue i, which means the total size of packets that can be buffered. The switch dynamically adjusts T_i every packet arrivals. Since a service queue should be able to occupy the buffer up to the port buffer size, the switch does not simply drop the arriving packet P when the dropping threshold is exceeded. Instead, if the buffer occupancy of the service queue of packet P exceeds T with packet P, the switch increases the dropping threshold of the queue and decreases that of the victim queue, which is defined as the queue has the largest extra buffer size. However, to guarantee the weighted fair share rate, the switch drops packet P without threshold adjustment when the victim queue is an unsatisfied active queue.

2) Detailed Design: DynaQ operates before enqueueing decisions. As shown in Algorithm 1, the switch first compares dropping threshold T_p with the sum of queue length of queue p and the size of packet P. If enqueueing packet P does not make the queue length exceed T_p, nothing will be done. Otherwise, the switch begins to adjust packet dropping thresholds or to drop the packet.

[1]We express that queue i is unsatisfied if the packet dropping threshold T_i of queue i is less than satisfaction threshold S_i defined in Eq. (3). Otherwise, queue i is satisfied.

Table I
USED NOTATIONS.

Notation	Description
M	Number of queues
P	Arriving packet
p	Queue index of packet P
B	Port buffer size
w_i	Weight of queue i
T_i	Packet dropping threshold of queue i
q_i	Queue length of queue i
$WBDP_i$	Weighted BDP of queue i
S_i	Satisfaction threshold of queue i
T_i^{ex}	Extra buffer size of queue i

Algorithm 1 Pseudocode of DynaQ

```
1: if q_p + size(P) > T_p then                          ▷ Exceeds threshold?
2:     v ← arg max_{i<M, i≠P} T_i^ex                    ▷ Find victim queue
3:     if T_v < size(P) ‖ (q_v > 0 & T_v − size(P) < S_v) then
4:         Return Drop(P)                               ▷ Protect unsatisfied queues
5:     else                          ▷ It is okay to adjust dropping thresholds
6:         T_v ← T_v − size(P)                          ▷ Decrease T of victim v
7:         T_p ← T_p + size(P)                          ▷ Increase T of queue p
8:     end if
9: end if
```

Packet Dropping Threshold: DynaQ isolates service queues using dynamic packet dropping thresholds. Each queue i has its own dropping threshold T_i. The aggregate dropping thresholds of all service queues is equal to the port buffer size. If the aggregate threshold exceeds the port buffer size, the buffer will be shared like the best-effort scheme. In contrast, if the aggregate threshold is less than the port buffer size, the buffer sharing policy becomes close to PQL. Therefore, when the switch is on, DynaQ initializes the packet dropping threshold of queue i that

$$T_i^{init} = B \times \frac{w_i}{\sum w} \tag{1}$$

In addition, to ensure $\sum_{i=1}^{M} T = B$, DynaQ always decreases the dropping threshold of victim queue before increasing the dropping threshold of the queue of packet P.

Victim Queue Selection: Before adjusting dropping thresholds, the switch should find the victim queue v. One intuition is that the victim should be the queue who is expected to experience the minimal impact after decreasing its dropping threshold. Therefore, a natural way to select the victim is finding the queue with the largest threshold. However, this may not work when service queues are with different queue weights. For example, consider 3 service queues with weights of 1:2:3. The switch can choose queue 3 as the victim queue although queue 3 has only a minimum buffer size of required to enjoy the weighted fair share rate.

To respect different queue weights, DynaQ selects a service queue with the largest extra buffer size as the victim queue. The extra buffer size of queue i is defined as

$$T_i^{ex} = T_i - S_i \tag{2}$$

The satisfaction threshold S_i specifies the minimum buffer size of queue i to achieve its weighted fair share rate regardless of other queues. Theoretically, $WBDP_i$ is enough to saturate the weighted bottleneck link capacity $C\frac{w_i}{\sum w}$. However, we find that the switch does not preserve weighted fair sharing when $S_i = WBDP_i$. This is because T_i fluctuates over time, preventing queue i from enjoying its fair share rate stably. Therefore, we need to satisfy the inequality $S_i > WBDP_i$ to make headroom to reduce the impact of the change of T_i. Thus, we simply use that

$$S_i = B \times \frac{w_i}{\sum w} \tag{3}$$

Modern line-rate switches have enough buffer size to allocate a buffer size per port larger than BDP. Since $B > BDP$, it is obvious that $S_i > WBDP_i$.

Victim Queue Search without Loops: Finding the victim queue can be done through linear search using loops. However, modern switching ASICs prevent loop operations to guarantee a deterministic packet processing delay.

To deal with this constraint, DynaQ uses binary search to find the victim queue. We make `MaxIdx` function that returns the index of the larger queue after comparing the extra buffer size between the two service queues. For example, when the switch supports 4 service queues, we can find the index of victim queue by referring to the return value of $\texttt{MaxIdx}\left(\texttt{MaxIdx}\left(1,2\right),\texttt{MaxIdx}\left(3,4\right)\right)$. This requires $O(\log n)$ complexity bounded to the number of service queues that the switch supports. Modern switches typically support 4 or 8 service queues per port. Therefore, the complexity is fixed to $O(2)$ or $O(3)$ depending on the target switch architecture.

Packet Dropping and Threshold Adjustment: After finding the victim queue, the switch decides whether to drop the packet or adjust dropping thresholds. The switch drops packet P if the dropping threshold of queue v is less than the size of packet P or queue v is an unsatisfied active queue. The former condition is to ensure $T_i \geq 0, \forall i$. The latter condition is to protect an unsatisfied active queue. If we allow a queue to take the buffer of unsatisfied active queues, aggressive queues with many flows can occupy the port buffer excessively. Meanwhile, the switch does not protect inactive queues from the aggressive queues to utilize free buffer space for high link utilization. When the above conditions are not met, DynaQ exchanges the dropping thresholds of queue v and queue p as much as the size of packet P. This finishes the operation of DynaQ. After this, the switch performs packet enqueueing decisions based on the port buffer occupancy.

3) Discussion: We now discuss several design issues.

ECN Support: DynaQ should support ECN-based transport protocols since they are also generic transport protocols. Since there exist ECN-based solutions, we employ PMSB [3] rather than designing our own ECN-based mech-

anism. Specifically, when ECN is enabled in the switch, DynaQ does not adjust dropping thresholds but marks the packet when the port buffer occupancy exceeds the port ECN marking threshold $K = C \times RTT \times \lambda$ and the queue length of arriving packet queue exceeds the per-queue ECN marking threshold $K_i = \frac{w_i}{\sum w} C \times RTT \times \lambda$ simultaneously.

Port Buffer Size: We have assumed that the port buffer size is constant so that the sum of dropping thresholds $\sum T$ can be equal to the port buffer size B. However, the operator can change the port buffer size, breaking the equality between $\sum T$ and B. This can be resolved by performing the initialization of the dropping thresholds via Equation (2) after adjusting the port buffer size.

IV. Implementation

A. Hardware Implementation

DynaQ can be implemented on hardware inexpensively. Since we cannot program most switching chips, we first analyze the overhead in ASIC implementation. Next, we discuss the implementation on programmable switches.

Processing Overhead in ASIC Implementation. The processing overhead of DynaQ in ASIC implementation is relatively small since the switching ASIC consumes hundreds of clock cycles to process a packet. Consider a typical hardware running at a clock frequency of 1 Ghz where 1 clock cycle is 1 ns. We also presume that the switch supports 8 service queues per port. Note that commodity switch ASICs support 4-8 service queues. With Algorithm 1, we can know that DynaQ requires up to 7 clock cycles. Broadcom Trident 3 ASIC offers a minimum per-packet processing delay of 800 ns. For this case, the overhead of DynaQ is only 0.88%.

Let us show the detailed analysis. In the worst case, Lines 1-3 and Lines 6-7 in Alg. 1 are performed. Line 1 consumes 1 clock cycle. Line 2 requires $\log 8 = 3$ clock cycles. Line 3 consumes 2 clock cycles because $(q_v > 0$ & $T_v - size(P) < S_v)$ must be performed before $||$ operation with $T_v < size(P)$. Note that comparison operations like $q_v > 0$ can be pipelined. Lines 6-7 require 1 clock cycle with pipelining because they have no read/write dependency.

Implementation on Programmable Switches. We analyze how DynaQ can be implemented on an programmable switch built with Barefoot Tofino [8]. With Tofino ASIC, we can program processing pipelines. Our implementation is based on Tofino Native Architecture (TNA).

TNA consists of 9 blocks whose 6 blocks are programmable and the other blocks only provide a fixed set of operations. DynaQ should be implemented in the Packet buffer and Replication Engine (PRE) where packet enqueueing and dequeueing decisions occur. However, the PRE is a fixed-function block in TNA. Therefore, we cannot implement any packet buffering mechanism directly due to the limited programmability. We should implement DynaQ in either ingress or egress pipeline indirectly. Since DynaQ operates before buffering, we implement our solution in the ingress pipeline.

Most variables like packet dropping thresholds, queue weights, and satisfaction thresholds can be defined as user-defined metadata. In addition, we can manipulate them at runtime. `MaxIdx` function to find the victim queue also can be defined in the ingress pipeline. One challenge to implement DynaQ in TNA is to obtain the queue length information. Since the PRE is not programmable, it is hard to obtain the queue length of the arriving packet queue in the ingress pipeline. Note that this is not an issue in ASIC implementation. In TNA, the queue length metadata is included in egress intrinsic metadata, but it is prohibited to share metadata between the pipelines. Although TNA supports metadata bridging, we can only deliver ingress pipeline metadata to the egress pipeline.

To deal with the issue, we may utilize an extern register to inform the queue length to the ingress pipeline. We deliver the value of `deq_qdepth` metadata, the queue length at the packet dequeue time, to the ingress pipeline. The value is stored in user-defined queue length metadata, and DynaQ reads it when the operation is triggered. This may result in DynaQ operations based on inaccurate queue length information. However, with round-robin based schedulers, we believe that some inaccuracy is tolerable to isolate service queues. The correct implementation on programmable switches is our future work.

B. Software Implementation

To compare DynaQ with various existing works in a flexible environment, we implement a software prototype of DynaQ as a Linux qdisc module on a server-emulated switch. Our module consists of two stages as follows.

Enqueueing Stage: When the packet from TCP/IP stack arrives to the qdisc layer, the module returns the index of the corresponding service queue by referring to the DSCP field in the IP header. Next, the switch checks whether the buffer is available to enqueue the arriving packet. Basically, the switch performs packet enqueueing decisions based on the port buffer occupancy or per-queue buffer occupancy relying on switch configuration. When the switch uses DynaQ, the switch performs additional operations to adjust packet dropping thresholds before enqueueing. If the packet can be buffered, the switch enqueues the packet into the corresponding queue. When ECN is enabled, the switch performs ECN marking at the end of enqueueing stage.

Dequeueing Stage: The switch dequeues packets through work-conserving packet schedulers, which includes SPQ, DRR, and WRR. The packet schedulers follow the data structure and mechanism of the current Linux qdisc implementation. The dequeued packets go to NIC drivers and NIC hardware before it is transmitted to the wire. Our module uses a token-bucket rate limiter to shape outgoing traffic at 99.5% of the NIC capacity. This is to avoid excessive

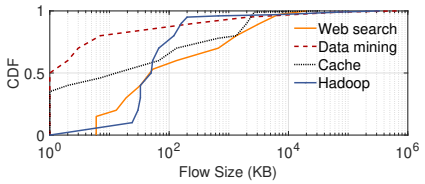


Figure 2. Used workloads in dynamic flow experiments.

buffering in NIC drivers and NIC hardware, which can lead to inaccurate buffer occupancy in the qdisc.

V. Performance Evaluation

In this section, we evaluate the performance of DynaQ.We answer the following key questions:

- How does DynaQ perform in practice?
- Does DynaQ perform well in large-scale data centers?
- How robust is DynaQ to network settings?

Traffic Workloads: We use four realistic workloads derived from production data centers, which are a web search workload [14], a data mining workload [15], a cache workload [16], and a hadoop workload [16]. As shown in Figure 2, these workloads generate flows whose size distributions are heavy-tailed. For example, in the data mining workload, roughly 50% of flows are 1KB while 90% of bytes are from flows larger than 100MB. Like MQ-ECN [1] and TCN [2], we also use the web search workload for all service queues in testbed experiments while using all the four workloads in simulations. This is because the web search workload is the most challenging workload due to its less skewed flow size distribution that generates multiple concurrent flows on the bottleneck. In addition, it is hard to make all workloads active during experiments because every workloads result in different finish times.

Compared Schemes: We mainly compare DynaQ with the following schemes: BestEffort and PQL. BestEffort denotes the best-effort scheme that manages the buffer among service queues in a FIFO manner. PQL isolates service queues by reserving a static per-queue buffer size. When we consider low latency, we also compare DynaQ against the ECN-based solutions, TCN [2], and PMSB [3].

Performance Metric: Our primary performance metrics are throughput and FCTs. We also use throughput share ratio and Jain's fairness index for better understanding of throughput results. For the average FCT, we breakdown the FCT across different flow sizes to analyze the impact on small ($\leq$ 100KB) and large flows ($>$ 10MB). We also consider the 99th percentile FCT of small flows to evaluate tail latency. Due to space limitation, we omit the result of medium flows whose results are similar to overall flows. For clear comparison, the FCT results are normalized by the values of DynaQ.

A. Testbed Experiments

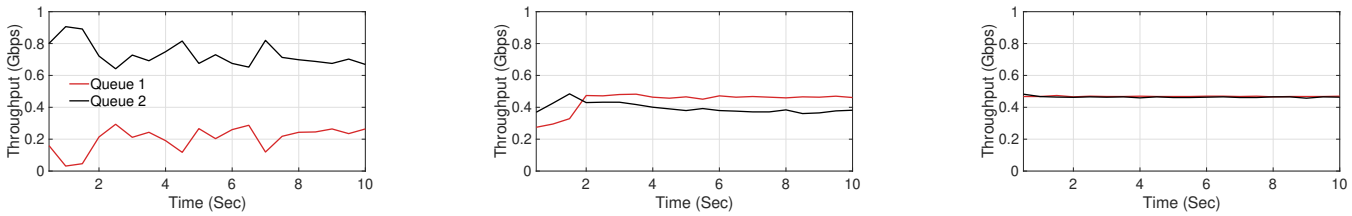
Testbed Setup: We have built a small-scale testbed, which consists of 5 servers connected to a server-emulated switch. The switch is equipped with two Intel I350-T4 v2 NICs where each NIC supports 4$\times$1GbE ports. Each server is also equipped with a Intel gigabit NIC. All servers use Linux kernel 3.18.11. We use TCP for the non-ECN schemes and DCTCP [14] for the ECN-based schemes. We set TCP RTO_{min} to 10ms as suggested in many existing works [1, 2, 14, 17, 18]. The initial congestion window size is set to 10 packets as suggested in RFC6928. In end-hosts and the switch, we disable large send offload (LSO) and large receive offload (LRO) to reduce traffic burstiness and emulate switch hardware behaviors more correctly. To emulate a switch with Broadcom 56538 ASIC, the switch has a 85KB of port buffer, which is completely shared by all service queues. PQL is the only exceptional scheme since it limits per-queue buffer size. The base RTT is roughly 500 μs and the corresponding BDP is 62.5KB. We set ECN marking thresholds for DCTCP and TCN to 30KB and 240μs, respectively. These are the best values experimentally found. Note that there is a theory-practice gap in determining ECN marking thresholds [2].

1) Static Flow Experiments: In static flow experiments, we focus on weighted fair sharing and work conservation.

Convergence and Queue Evolution: In this experiment, we show the basic results with two active queues between 4 DRR queues having the equal quantum of 1.5KB. We use three servers where two servers are the senders for each service queue and the other one is the receiver. Using `iperf`, each sender starts flows to the receiver for 10 seconds. The sender of queue 2 generates 16 flows while the sender of queue 1 has only 2 flows. Ideally, the active queues should share bandwidth equally regardless of the number of competing flows and the inactive queues.

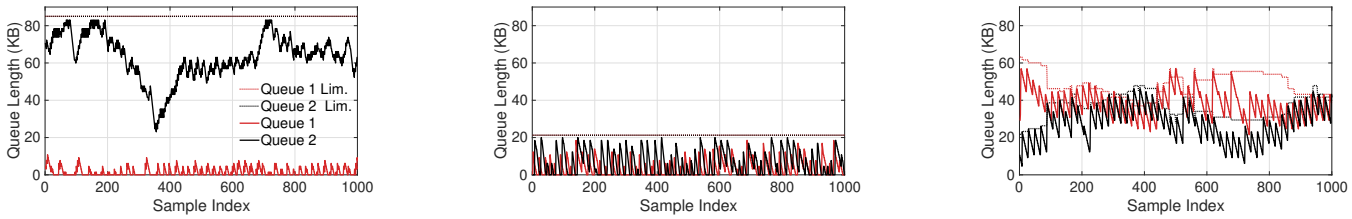
Figure 3 shows throughput of the active queues over time. It is easy to find that DynaQ is the only solution that shares bandwidth fairly. Throughput of the active queues in BestEffort does not converge, resulting in significant unfairness. With PQL, the active queues share the bandwidth fairer than BestEffort, but still results in considerable unfairness.

Figure 4 reports the queue length evolution for the active queues. We measure per-queue buffer occupancy every enqueueing and dequeueing operations and obtain 1K sequential samples. The dotted lines indicate per-queue buffer size. The queue evolution samples explain the throughput in Figure 3. In BestEffort, since queue 2 has more flows, queue 2 dominates the port buffer while queue 1 with smaller flows occupies a small buffer. In PQL, queue 2 can occupy buffer space more than that in BestEffort, but it is limited to a reserved size. Unlike the other schemes, DynaQ shares the buffer dynamically that dropping thresholds change over time. Thanks to this, each queue can occupy enough buffer



(a) BestEffort (b) PQL (c) DynaQ

Figure 3. Throughput convergence of two active DRR queues with equal weights.



(a) BestEffort (b) PQL (c) DynaQ

Figure 4. Queue length evolution of two active DRR queues with equal weights.

regardless of the number of flows and active queues.

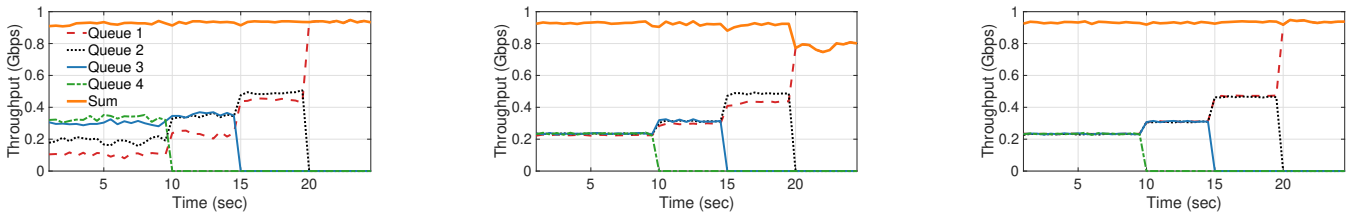
Weighted Fair Sharing and Work Conservation: In this experiment, we consider 4 DRR queues with the equal quantum as same as the previous experiment. However, we now vary the number of active queues over time. Each queue has a different number of flows that the sender of queue i starts 2^i flows to the receiver simultaneously. From 10 seconds, we change the number of active queues by stopping flows. At 10 seconds, the sender of queue 4 stops traffic. After 5 seconds, queue 3 becomes inactive. At 20 seconds, the sender of queue 2 no longer sends flows. The sender of queue 1 finishes at 25 seconds. We measure per-queue throughput every 0.5 seconds and obtain the aggregate throughput as well. Ideally, service queues should share bandwidth fairly regardless of the number of flows and the aggregate throughput always should be high regardless of the number of active queues.

Figure 5 shows the results. We observe that BestEffort fails to achieve fair sharing. When all the four queues are active, queue 4 with 16 flows occupies the largest bandwidth share because the port buffer is dominated by the packets of queue 4. Due to this, queue 1 only obtains 0.14Gbps of average throughput for the first 10 seconds. Even when only queue 1 and 2 are active at 15 seconds, the two queues do not share bandwidth fairly in spite of the scheduler because queue 2 has twice as many flows as queue 1.

PQL shows the better results than BestEffort. When all queues are active, PQL preserves fair sharing. However, when the number of active queues decreases, fair sharing is violated. We can see the unfair bandwidth sharing between the two service queues at $15 \sim 20$ seconds. More importantly, we can see that the aggregate throughput decreases as the queues becomes inactive. It is notable that the average aggregate throughput during $20 \sim 25$ seconds is only 0.78Gbps. This is because each service queue cannot utilize the remaining free buffer space. Unlike the compared schemes, DynaQ makes the service queues share bandwidth almost perfectly regardless of the number of flows. In addition, since DynaQ allows a queue to utilize free buffer space, the aggregate throughput does not decrease even when few queues are active.

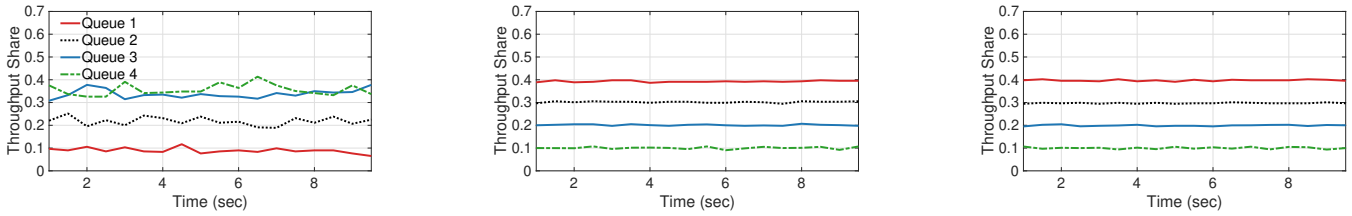
Impact of Queue Weights: In this experiment, we use the same scenario in the previous experiment. The difference is that we configure different quantums for the DRR queues. Our default quantum is 1.5KB of MTU. We set the weights of the queues to $\{4, 3, 2, 1\}$, which result in $\{6, 4.5, 3, 1.5\}$KB of quantums. We measure the per-queue throughput every 0.5 seconds for 10 seconds. Ideally, the queues should share the bandwidth by respecting their assigned weights.

Figure 6 shows the throughput share of each service queue. The throughput share is defined as $R_i(t)/\sum R(t)$ where $R_i(t)$ denotes the throughput of queue i at time t. The result with BestEffort shows that BestEffort does not preserve weighted fair sharing. In spite of different quantums, BestEffort allows a queue with many flows to occupy more throughput share. For example, the average throughput share of queue 4 for 10 seconds is 0.35 although its desirable share is 0.1. PQL achieves weighted fair sharing in this experiment. This is not surprising because PQL assigns a static buffer size to each service queue. However, as we have shown in the previous experiment, PQL loses throughput as the number of active queues decreases although it is omitted in this experiment. The result demonstrates that DynaQ



(a) BestEffort (b) PQL (c) DynaQ

Figure 5. Bandwidth sharing between 4 DRR queues with equal weights.



(a) BestEffort (b) PQL (c) DynaQ

Figure 6. Bandwidth sharing between 4 DRR queues with different queue weights of 4:3:2:1.

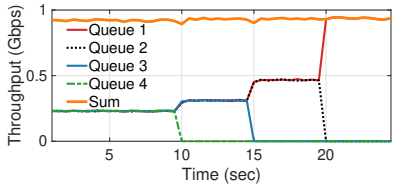


Figure 7. Throughput with 2 TCP and 2 CUBIC senders.

achieves weighted fair sharing by respecting the assigned weights regardless of the number of competing flows.

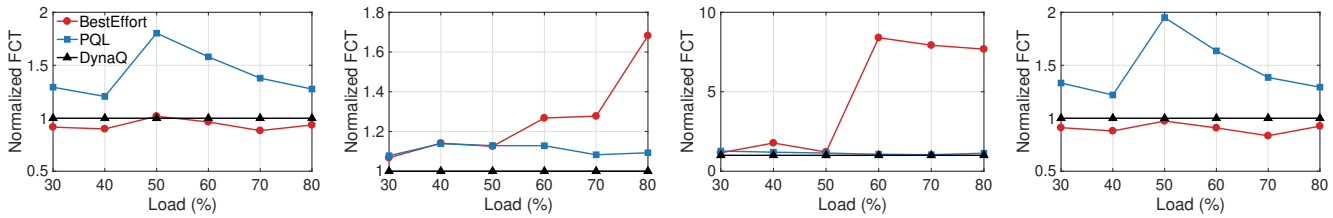
Impact of Transport Protocols: We consider a scenario where senders use different transport protocols. We have tried to use emerging protocols, but it is hard to obtain their codes. Instead, we use TCP and CUBIC. Unlike the previous experiments where all senders use TCP, we now make the senders of queue 3 and queue 4 use CUBIC. Except transport protocols, we use the same scenarios in the equal sharing experiment. Figure 7 shows the results. We can see that DynaQ achieves fair sharing regardless of employed transport protocols. One notable point is that, at 10 seconds and 15 seconds, we can see that the aggregate throughput decreases slightly for a moment when queue 4 and queue 3 become inactive. This is because of the time for ramp-up of the other queues, not due to our buffer sharing policy.

2) Dynamic Flow Experiments: In dynamic flow experiments, we show that DynaQ can achieve low latency.

Methodology: At the switch, we configure SPQ/DRR where one queue has higher priority than the other four DRR queues. Packets in the DRR queues can be dequeued only when the SPQ queue is empty. This is a common switch configuration to accelerate latency-sensitive small flows [2]. We use 1.5KB of the equal quantum for all DRR queues.

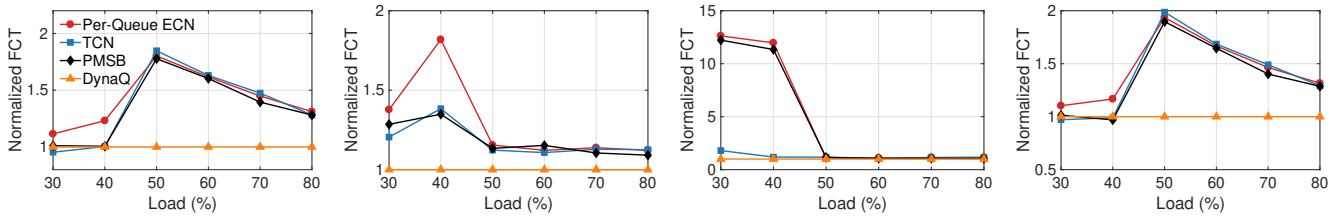
We use a client/server application [1] to generate traffic with the web search workload. We have 4 servers and 1 client. The client initially opens 5 persistent TCP connections to each server. The client generates requests to the servers through available connections. When there is no available connection, the client creates a new connection. The inter-arrival time of generated requests follows a Poisson process. When a request arrives, each of servers responds with requested data size. The server application sets DSCP values for outgoing packets using `setsockopt` and a flow is mapped to one of the service queues randomly. We also employ a two-level PIAS [19] to classify small flows with 100KB of a priority demotion threshold. Therefore, the first 100K bytes are buffered into the SPQ queue and the remaining bytes are enqueued into the DRR queues in the low priority. Overall, we generate 10K flows by varying the traffic load from 30% to 80%.

Comparison with Non-ECN Schemes: Figure 8 reports the FCT results that compare DynaQ with non-ECN schemes. The average FCTs of overall and large flows are similar since most of bytes in the benchmark traffic come from large flows. Compared to PQL, DynaQ achieves the better average FCT for large flows by up to $1.95\times$. Similarly, DynaQ outperforms PQL in the average FCT for overall flows by up to $1.80\times$. This is because a service queue in PQL can occupy a limited buffer space. The results against BestEffort are mixed whose gaps are within $0.90\times \sim 1.02\times$ and $0.83\times \sim 0.97\times$ for the average FCT of overall flows and large flows, respectively. The reason why BestEffort outperforms DynaQ is that large flows blocks small flows in the port buffer. Not surprisingly, we can see that DynaQ



(a) Overall: Avg (b) (0, 100KB]: Avg (c) (0, 100KB]: 99th Percentile (d) (10MB, ∞]: Avg

Figure 8. FCT comparison against non-ECN schemes with SPQ (1 queue) / DRR (4 queues).



(a) Overall: Avg (b) (0, 100KB]: Avg (c) (0, 100KB]: 99th Percentile (d) (10MB, ∞]: Avg

Figure 9. FCT comparison against ECN-based schemes with SPQ (1 queue) / DRR (4 queues).

significantly outperforms BestEffort in the FCT of small flows. Although BestEffort outperforms DynaQ in the FCT of large flows, BestEffort cannot preserve weighted fair sharing as we have shown.

For the average FCT of small flows, DynaQ achieves the best performance between the schemes. DynaQ beats BestEffort by $1.26\times$ on average across the traffic loads. The performance gap increases as traffic load grows. Compared to PQL, DynaQ has the better performance within $1.08\times \sim 1.14\times$. From the 99th percentile FCT result, we find that BestEffort shows the significantly worse performance than DynaQ when traffic load increases. For example, the FCT gap in 60% of load is $8.40\times$. Unlike BestEffort, PQL shows relatively stable performance. However, DynaQ still outperforms PQL by $1.14\times$ on average across the loads.

Comparison with ECN-based Schemes: We now discuss the FCT results of DynaQ compared to the ECN-based schemes. Figure 9 shows the results. Like the results in Figure 8, the results for overall and large flows are similar. DynaQ shows the mixed performance against the ECN-based solutions, but generally outperforms the comparisons. TCN shows the similar average FCTs for overall and large flows when traffic loads are within $30\% \sim 40\%$. However, the maximum gap is only $0.95\times$ for overall flows when load is 30%. DynaQ outperforms TCN for the rest traffic loads whose the ranges of performance gaps are within $1.28\times \sim 1.85\times$ and $1.29\times \sim 1.99\times$ for overall and large flows, respectively. PMSB also has a similar performance to TCN. Per-Queue ECN shows the worst performance between the schemes. When we consider the results for small flows, DynaQ beats the other ECN-based schemes in both average and 99th percentile FCTs. For example, DynaQ is better than PMSB by up to $1.29\times$ in the average FCT. The ECN-based schemes show the worse performance at lower traffic loads than high traffic loads. DynaQ outperforms PMSB and Per-Queue ECN by $12.23\times$ and $12.63\times$ at 30% of traffic load, respectively.

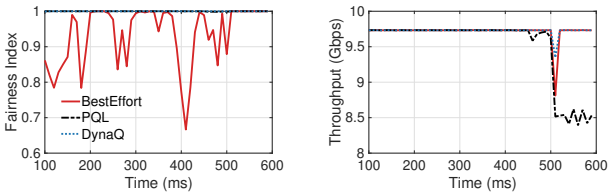
B. Large-Scale Simulations

We conduct simulations to evaluate the performance of DynaQ in large-scale environments using ns-2.

1) Static Flow Simulations: In this simulations, we evaluate DynaQ's robustness.

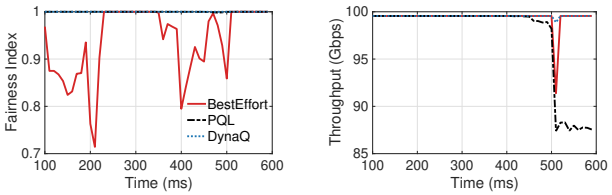
Methodology: Like our testbed, we build a star topology to emulate a compute rack. We consider two high-speed links: 10Gbps and 100Gbps. The base RTTs are $84\mu s$ and $40\mu s$ for each of links. In the switch, we configure WRR with equal weights for packet scheduling. We have enabled Jumbo frame for 100Gbps links. We consider Broadcom Trident+ and Trident 3 ASICs for 10Gbps and 100Gbps links, respectively. Therefore, each port has 192KB and 1MB of buffers, which are completely shared by service queues, except in PQL. We use TCP for the transport protocol and set RTO_{min} to 5ms, the lowest stable value in `jiffy` timer.

Impact of Link Capacity: We first perform a simulation with 10Gbps links. We have 8 services, which are mapped to each of 8 service queues. There exist $2 \times i$ senders for queue i. Every senders of service queues start a flow at the beginning simultaneously. From 200ms, the senders of queues $2 \sim 8$ stop their transmissions every 50ms in order. For example, the senders of queue 3 finishes its flows at 250ms and queue 1 is the only active queue after 500ms. We measure per-queue throughput every 10ms. Using the measured data, we calculate Jain's fairness index between



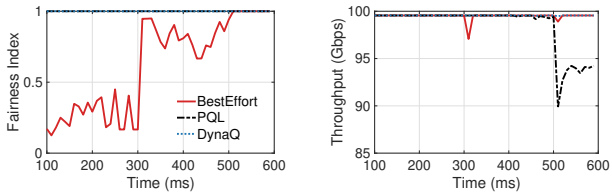
(a) Fairness Index (b) Aggregate Throughput

Figure 10. Bandwidth sharing on 10Gbps links.



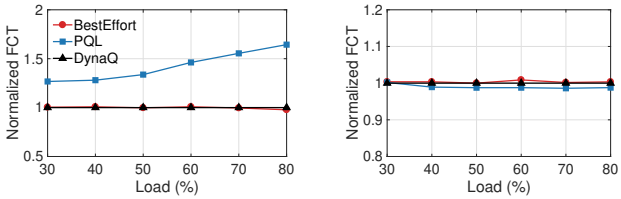
(a) Fairness Index (b) Aggregate Throughput

Figure 11. Bandwidth sharing on 100Gbps links.



(a) Fairness Index (b) Aggregate Throughput

Figure 12. Bandwidth sharing on 100Gbps links with many flows.



(a) Overall: Avg (b) (0, 100KB]: 99th Percentile

Figure 13. FCT comparison with SPQ (1 queue) / DRR (7 queues).

active queues and the aggregate throughput. The aggregate throughput is to inspect whether the link is fully utilized at any time. If a solution preserves weighted fair sharing and work conservation, the two metrics always should be high.

Figure 10 shows the bandwidth sharing results with 10Gbps links. Not surprisingly, DynaQ and PQL achieve the near-optimal fairness index while BestEffort causes fluctuations. For example, the fairness index plunges to 0.67 at 410ms. This is because BestEffort cannot handle service queues with many flows. Figure 10 (b) shows that DynaQ is the only solution that maintains high link utilization between the schemes. When queue 8 finishes at 500ms, PQL causes a huge throughput collapse. PQL maintains the aggregate throughput around 8.5Gbps after 500ms. This is because queue 1, the only active service queue, cannot occupy the buffer as much as the BDP.

We perform the same simulation with 100Gbps and Figure 11 reports the results. We find that DynaQ can achieve work conservation and preserve weighted fair sharing with a high link capacity. The overall tendency is very similar to the 10Gbps results. BestEffort cannot provide per-queue fairness and PQL loses a significant amount of throughput when service queue 1 is the only active queue. In addition, DynaQ does not lose throughput much at 500ms. Unlike DynaQ, BestEffort causes 9.2Gbps of throughput loss.

Impact of Traffic Dynamics: We now inspect the how DynaQ is robust to traffic dynamics. We conduct a simulation with the almost same scenario as the previous simulation with 100Gbps. The only different setting is the number of senders per service queue. We consider a very extreme scenario where service queue i has $2^{(3+i)}$ senders generating a single flow. For example, queue 8 has 2048 flows. Figure 12 plots the results. We observe that DynaQ is robust to the extreme traffic scenario. BestEffort shows the worst performance in the fairness index that it achieves only 0.24 of fairness index for the first 200ms. The scheme also loses throughput at 300ms for a moment. PQL still fails to achieve work conservation. The aggregate throughput stays below 94.5Gbps from 500ms.

2) Dynamic Flow Simulations: We now evaluate DynaQ's performance in large-scale data center networks.

Methodology: We build a non-blocking leaf-spine topology, a widely used data center network topology design. Our topology has 12 leaf (ToR) switches and 12 spine (Core) switches. Each leaf switch has 12×10Gbps downlinks and 12×10Gbps uplinks. The base RTT across spine switches is 85.2μs. We use ECMP as the load balancing scheme. We use TCP and RTO_{min} is 5ms. Each switch port has a 192KB buffer shared by service queues except in PQL.

We configure SPQ/DRR with 8 service queues where one queue is the shared high priority queue and the other 7 queues are dedicated DRR queues with equal quantums. Like the testbed experiments, we also employ a two-level PIAS [19] whose priority demotion threshold is 100KB to classify small flows from large flows. We evenly classify the 144×143 communication pairs into 7 services and each service has its own service queue. Different services use different traffic distributions in Figure 2. We generate 10K flows by varying traffic load from 30% to 80%.

Results: Figure 13 shows the results. Due to space limitation, we only discuss the average FCT of overall flows and the 99th percentile FCT of small flows. In the average FCT for overall flows, we observe that DynaQ has mixed results compared to BestEffort and outperforms PQL. This is similar to the results in the testbed experiments. The gaps with BestEffort are within $0.98\times \sim 1.01\times$. For the

99th percentile FCT of small flows, we can see that DynaQ achieves similar performance to the compared schemes. PQL slightly outperforms DynaQ that the maximum gap is $0.98\times$. In the testbed experiments with 1Gbps links, BestEffort results in a performance degradation when loads are high. However, with 10Gbps links, DynaQ beats BestEffort only by up to $1.01\times$ due to the increased link capacity.

VI. Conclusion

This paper proposed DynaQ, a multi-queue management solution for multi-queue data centers. DynaQ is the first protocol-independent solution that can isolate service queues with generic transport protocols through dynamic packet dropping thresholds. We have discussed how DynaQ can be implemented on hardware. To compare DynaQ with existing solutions, we have implemented a software prototype of DynaQ as a Linux qdisc module. We conducted a series of testbed experiments and simulations. Our evaluation results demonstrated that DynaQ ensures work conservation, weighted fair sharing, and low latency without protocol dependency.

Acknowledgement

This research was partly sponsored by the National Research Foundation of Korea (NRF) grant funded by the Ministry of Science and ICT (No. 2019R1A2C2088812), Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (No. 2017M3C4A7083676). Wonjun Lee is the corresponding author.

References

[1] W. Bai, L. Chen, K. Chen, and H. Wu, "Enabling ecn in multi-service multi-queue data centers," in *Proc. of USENIX NSDI*, 2016, pp. 537–549.

[2] W. Bai, K. Chen, L. Chen, C. Kim, and H. Wu, "Enabling ecn over generic packet scheduling," in *Proc. of ACM CoNEXT*, 2016, pp. 191–204.

[3] Y. Pan, C. Tian, J. Zheng, G. Zhang, H. Susanto, B. Bai, and G. Chen, "Support ecn in multi-queue datacenter networks via per-port marking with selective blindness," in *Proc. of IEEE ICDCS*, July 2018, pp. 33–42.

[4] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, and M. Yu, "Hpcc: High precision congestion control," in *Proc. of ACM SIGCOMM*. New York, NY, USA: ACM, 2019, pp. 44–58. [Online]. Available: http://doi.acm.org/10.1145/3341302.3342085

[5] I. Cho, K. Jang, and D. Han, "Credit-scheduled delay-bounded congestion control for datacenters," in *Proc. of ACM SIGCOMM*. New York, NY, USA: ACM, 2017, pp. 239–252. [Online]. Available: http://doi.acm.org/10.1145/3098822.3098840

[6] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats, "Timely: Rtt-based congestion control for the datacenter," in *Proc. of ACM SIGCOMM*, 2015, pp. 537–550.

[7] C. Lee, C. Park, K. Jang, S. Moon, and D. Han, "Accurate latency-based congestion feedback for datacenters," in *Proc. of USENIX ATC*, Jul. 2015, pp. 403–415.

[8] "Tofino programmable switch," https://www.barefootnetworks.com/.

[9] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pfabric: Minimal near-optimal datacenter transport," in *Proc. of ACM SIGCOMM*, 2013, pp. 435–446.

[10] G. Kim and W. Lee, "Absorbing microbursts without headroom for data center networks," *IEEE Communications Letters*, vol. 23, no. 5, pp. 806–809, May 2019.

[11] L. Zheng, Z. Qiu, S. Sun, W. Pan, Y. Gao, and Z. Zhang, "Design and analysis of a parallel hybrid memory architecture for per-flow buffering in high-speed switches and routers," *Journal of Communications and Networks*, vol. 20, no. 6, pp. 578–592, Dec 2018.

[12] G. Kim and W. Lee, "Enabling service queue isolation in multi-tenant data centers," *IEEE Communications Letters*, vol. 23, no. 11, pp. 1949–1952, Nov 2019.

[13] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. M. Watson, A. W. Moore, S. Hand, and J. Crowcroft, "Queues don't matter when you can jump them!" in *Proc. of USENIX NSDI*. USA: USENIX Association, 2015, p. 1–14.

[14] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *Proc. of ACM SIGCOMM*, 2010, pp. 63–74.

[15] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: A scalable and flexible data center network," in *Proc. of ACM SIGCOMM*, 2009, pp. 51–62.

[16] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proc. of ACM SIGCOMM*, 2015, pp. 123–137.

[17] G. Judd, "Attaining the promise and avoiding the pitfalls of TCP in the datacenter," in *Proc. of USENIX NSDI*, 2015, pp. 145–157.

[18] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller, "Safe and effective fine-grained tcp retransmissions for datacenter communication," in *Proc. of ACM SIGCOMM*, 2009, pp. 303–314.

[19] W. Bai, K. Chen, H. Wang, L. Chen, D. Han, and C. Tian, "Information-agnostic flow scheduling for commodity data centers," in *Proc. of USENIX NSDI*, May 2015, pp. 455–468.