Burst can be Harmless: Achieving Line-rate Software Traffic Shaping by Inter-flow Batching

Danfeng Shan[1], Shihao Hu[1], Yuqi Liu[1], Wanchun Jiang[2], Hao Li[1],
Peng Zhang[1], Yazhe Tang[1], Huanzhao Wang[1], Fengyuan Ren[3]
[1]*Xi'an Jiaotong University* [2]*Central South University* [3]*Tsinghua University*

IEEE INFOCOM 2023 - IEEE Conference on Computer Communications | 979-8-3503-3414-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/INFOCOM53939.2023.10229082

***Abstract*—Traffic shaping is a common function at end hosts. Compared with hardware ones, software shapers are more flexible to be developed and deployed, and thus are very attractive. Nevertheless, software approaches are still unsatisfactory as they struggle to saturate 40Gbps and higher speed.**

While much effort has been made to reduce the *intrinsic* overhead of software traffic shaping, we find that it is the *extrinsic* overhead, such as PCIe communications and interrupts, that hinders shaping from achieving 40Gbps - 100Gbps speed. Batching is an effective way to amortize these overheads. However, blindly batching can degrade the network performance, as it introduces bursts into the network. Diving into the dilemma, we find that *intra-flow burst* is to blame for harming the network performance, while *inter-flow burst*, consisting of packets from different flows, can be naturally demultiplexed in the network.

Based on the insight, we present FlowBundler, which can achieve efficient traffic shaping by inter-flow batching. Testbed experiments show that FlowBundler can achieve an accurate shaping of 98Gbps with a single CPU core, which is 2.6× better than state-of-the-art approaches. Large-scale simulations show that FlowBundler can batch packet transmissions without harming the network performance.

***Index Terms*—Traffic shaping, Batching, Traffic burst, Data center networks**

I. INTRODUCTION

In data centers, traffic shaping (or rate limiting/pacing) is a crucial component at end hosts. For example, in the cloud, massive tenants and applications share network bandwidth. Cloud providers need to employ rate limiting to provide bandwidth isolation among tenants and applications [1]–[5]. At the transport layer, congestion control needs packet pacing for two purposes. First, lots of congestion control algorithms are rate-based [6]–[11]. They need to adjust the transmission rate at fine granularity. Second, congestion control also needs packet pacing to eliminate traffic bursts, which can obscure congestion signals [6], [12] and even result in packet dropping at shallow-buffered switches [13].

Traffic shaping can either be implemented in software or hardware. Although hardware approaches are faster, software traffic shaping has several advantages [14]. First, software traffic shaping is much more flexible for development and deployment. Software traffic shaping can be quickly developed with high-level programming languages and does not need any dedicated hardware. Thus, it can be easily deployed on any host. Second, software traffic shaping has access to richer resources (e.g., memory), enabling them to support more complicated network policies. As a result, software approaches are very attractive in production networks [3], [5], [14]–[19].

Nevertheless, software traffic shaping incurs high CPU overhead [18]. Recently, several efforts [14], [18] have been made to reduce the *intrinsic* overhead of software traffic shaping. They have significantly reduced the CPU overhead and can scale to tens of thousands of flows. Some of them have already been deployed in production networks [5], [11], [18]. However, they still struggle to achieve accurate traffic shaping in high-speed networks (i.e., 40-100Gbps).

In this paper, we find that the *intrinsic* overhead of software traffic shaping has reached minimal with recent advances. Rather, it is the *extrinsic* overhead that hinders software traffic shaping schemes from achieving 40-100Gbps speed. Specifically, traffic shaping incurs massive interrupts and PCIe communications on a per-packet basis. These operations dominate the overhead of shaping traffic in high-speed networks.

To further improve the CPU efficiency, it is essential to utilize *batching* (i.e., processing several packets at a time) to amortize these extrinsic overheads. However, there is a dilemma when directly employing batching. On the one hand, batching can effectively reduce the CPU overhead by 5.9-9.5× (§III-B). On the other hand, batching introduces traffic bursts into the network, which impairs the transmission performance. Specifically, we show that blindly batching can degrade the transmission performance by ∼2× (§III-B). As a result, current traffic shaping schemes usually avoid batching, impeding them to achieve higher shaping speed.

In this paper, we show that it is feasible to achieve the best of both worlds. We find that it is *intra-flow burst*, which consists of packets from the same flow, that should be blamed for harming the network performance. On the other hand, *inter-flow burst*, which consists of packets from different flows, is harmless to the network. The reason is that burst is harmful only when it causes queue buildup inside the network, while *inter-flow burst can be naturally demultiplexed before the congestion point*. This is due to the unique traffic characteristics of the data center networks. Specifically, most congestion occurs at the last hop [20], [21], where the flows heading for different destinations have already been demultiplexed. Thus, traffic shaping schemes only need to focus on eliminating intra-flow bursts, while taking advantage of *inter-flow batching* to achieve fast packet processing.

In light of this insight, we present FlowBundler, which leverages *inter-flow batching* to achieve efficient traffic shaping without harming the network performance (§IV). FlowBundler schedules packet transmissions at a speed of the maximum shaping rate among all flows. In this way,

intra-flow bursts can be eliminated for any flow. At each scheduled transmission, FlowBundler dequeues all packets whose expected departure time is no further than now. In this way, packets from different flows can be dequeued together and sent in a batch.

However, we face a challenge to truly make FlowBundler practical. The flow shaping rate varies spatially and temporally. Temporally, the shaping rate of a flow can be time-varying. For example, a congestion control algorithm can dynamically adjust a flow's sending rate every several RTTs. As a result, the time interval of sending two successive packets is varying accordingly. Regardless of the varying packet interval, the queue structure should densely arrange packets so that FlowBundler can quickly find the next packet to be dequeued. Spatially, there can be a wide disparity of shaping rates among different flows. As a result, the departure time between successive packets can be distributed across a long time horizon. The queue structure should arrange packets with fine time granularity to eliminate intra-flow bursts for all flows. Meanwhile, it should also accommodate all packets with an acceptable memory footprint regardless of the long time horizon.

To address the challenge, we propose a Multi-Level Timing Wheel (MLTW) structure (§IV). MLTW contains multiple Timing Wheels with hierarchical granularities[1] (e.g., 1μs, 2μs, 4μs, etc.). Each Timing Wheel is a time-indexed queue with a small memory footprint (i.e., 2KB). In this way, only a few Timing Wheels (i.e., 32) are required to cover a wide time horizon (0-256s), and only small memory footprint is needed to accommodate a wide disparity of shaping rates. Furthermore, packets are placed into a proper Timing Wheel whose granularity best matches the flow shaping rate. In this way, MLTW can densely arrange packets despite the varying shaping rate.

We implement FlowBundler in the kernel space as a Linux `qdisc` kernel module and in the user space based on BESS [22]. We evaluate FlowBundler on both 10Gbps and 100Gbps testbeds (§V). We demonstrate that FlowBundler can achieve high-accuracy traffic shaping at 98Gbps speed with a single CPU core, which outperforms Carousel and Eiffel by over 2.6× (§V-B). FlowBundler consumes three orders of magnitude less memory than Carousel and Eiffel to accommodate flows with a wide disparity of shaping rates (§V-C). Furthermore, FlowBundler can scale to 100K flows (§V-D). To examine the impact of FlowBundler on the network performance, we conduct large-scale simulations in a 10/40Gbps leaf-spine network with 144 hosts (§V-E). Simulation results confirm that FlowBundler can batch packet transmissions without degrading the network performance.

II. Background of Software Traffic Shaping

Traffic shaping, also known as rate limiting and packet pacing, is a kind of non-work-conserving packet scheduling algorithm that enforces packet transmissions at a specified rate. Traditional traffic shaping schemes use the token bucket algorithm to throttle the traffic rate (e.g., TBF [23] and HTB [24]). For this kind of traffic shaping schemes, the shaping algorithm itself incurs low cost [18]. However, it has poor scalability. Specifically, to employ various rate limiting policies for different traffic classes, each traffic class has to be associated with a separate shaper (and queue). With tens of thousands of traffic classes, the overhead of synchronization and queue maintenance is very high.

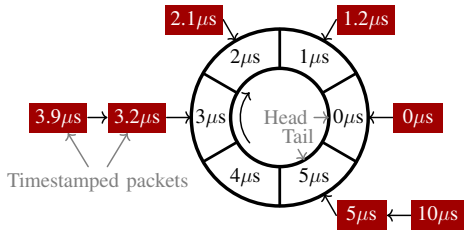


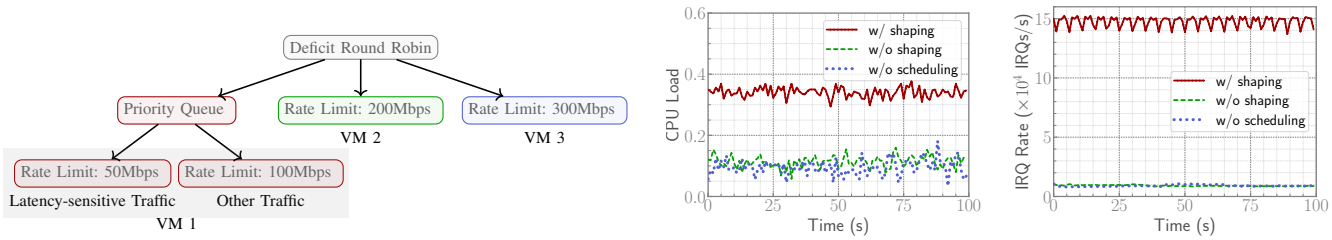
Fig. 1: A Timing Wheel with 1μs granularity and 5μs horizon

To eliminate the overhead of inter-traffic-class synchronization, state-of-the-art traffic shaping schemes are based on EDT (Earliest Departure Time) model [25], [26] (e.g., FQ [27], Carousel [18], and Eiffel [14]). In this model, the stipulation of traffic rates is decoupled from the execution of traffic shaping. When passing through the shaping policies, each packet is timestamped with EDT, which indicates the allowed transmission time conforming to the shaping policy. For example, if a policy limits the traffic rate by R, then the EDT of i-th arriving packet is calculated by $EDT_{i-1} + L_i/R$, where EDT_{i-1} is the EDT of the previous packet and L_i is the length of the i-th packet. Finally, the traffic shaper puts all arriving packets into a single queue based on their EDTs. At the dequeue side, the traffic shaper transmits any packet whose EDT becomes smaller than the current time. In this way, a single centralized shaper is needed to enforce traffic shaping for all traffic classes.

In the EDT model, the queue does not schedule packets in a First-In-First-Out (FIFO) manner. Rather, the queue should enqueue and dequeue packets based on EDT. There are two typical queue structures to achieve this. The Linux queueing discipline (`qdisc`) FQ utilizes a Red-Black (RB) tree to organize packets based on EDTs. Specifically, FQ puts all incoming packets into per-flow queues. To achieve packet pacing for outgoing packets, FQ uses an RB tree to maintain the EDT of the first packet in each per-flow queue. When transmitting a packet, it searches the RB tree to find the packet with the smallest departure time. The packet is allowed to be sent if its departure time is smaller than now.

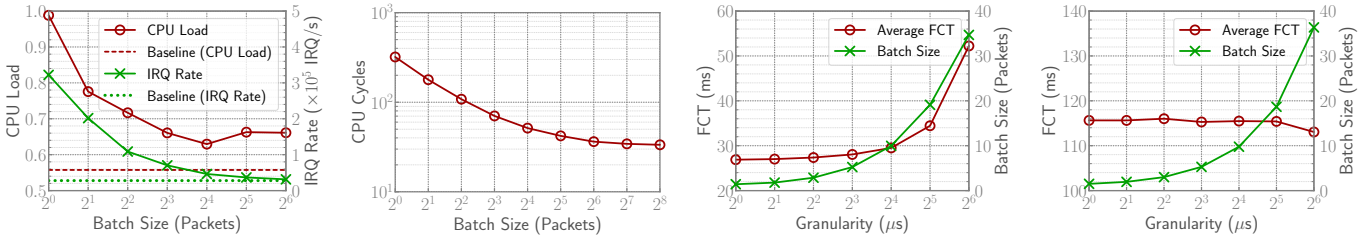
FQ cannot reach the full potential of EDT-based shaping, as it still organizes packets with multiple queues and thus suffers from synchronization overhead. More efficiently, Carousel [18] and Eiffel [14] use a single queue to shape packets of all flows, which truly liberates the traffic shaper from maintaining per-queue structures and makes the shaper scalable to a large number of flows. Specifically, they employ a Timing Wheel structure [28] to queue packets according to their EDTs (as shown in Fig. 1). Timing Wheel is a time-indexed circular

[1]By granularity, we mean the interval between two successive dequeue operations.



(a) Packet scheduling policy (b) CPU load (c) Software interrupt rate

Fig. 2: Overhead of different packet scheduling policies



(a) CPU load (Kernel) (b) CPU Cycles (DPDK) (c) FCT: one-to-one traffic (d) FCT: fanout traffic

Fig. 3: Effect of batch size

queue. It splits a period of time into multiple time slots. Each element of the queue represents a time slot, holding all packets whose EDTs are within the time slot.

Due to the high efficiency of Timing-Wheel-based approaches, many modern networking systems have deprecated the traditional token-bucket-based approaches and adopted the Timing-Wheel-based traffic shaping [5], [9], [11], [18].

III. Motivation

A. High Overhead of Software Traffic Shaping

Despite the endeavor to improve the efficiency of software traffic shaping schemes, they still incur high CPU overhead in high-speed networks.

We use an experiment to illustrate this. We establish a hierarchical packet scheduling policy with `tc`, as shown in Fig. 2a. The topmost policy enforces fair share of bandwidth among VMs with a Deficit Round Robin (DRR) scheduler. VM 1 uses a priority scheduler to prioritize latency-sensitive traffic. All kinds of traffic are rate-limited. We use TBF for rate limiting as it incurs the lowest overhead with a small number of flows [18] (details of testbed are in §V-A).

Fig. 2b shows the CPU overhead with different scheduling policies. Experiment results indicate that 92.5% of the CPU overhead in the packet scheduling system is attributed to traffic shaping. The reason is two-fold.

(1) Traffic shaping introduces lots of interrupts. Different from work-conserving packet schedulers, a traffic shaper needs to insert time gaps between successive packets. In the Linux kernel, this is achieved by software interrupts. Specifically, after sending a packet, the shaper sets up a timer to delay the transmission of the next packet. When the timer expires, a software interrupt will be invoked to transmit the packet. In comparison, work-conserving packet schedulers immediately send any packets waiting in the queue, introducing much fewer interrupts. Our experiments show that traffic shaping introduces 15× more interrupts (Fig. 2c).

(2) Traffic shaping makes low-level optimizations ineffective. In high-speed networks, the overhead of PCIe communication between operating system and NIC is non-negligible [29]. To achieve 40Gbps rate for 1500B packets, a packet needs to be sent every 300ns while a separate PCIe write operation may take up to ∼900ns [29]–[31]. To improve the efficiency of PCIe communication, modern NICs along with their drivers introduce a number of optimizations (e.g., enqueueing TX descriptors in batches, pre-fetching descriptors, interrupt moderation). However, all these low-level optimizations work on the basis of batched transmission, while traffic shaping needs to send packets one by one to eliminate traffic bursts. Without these optimizations, the achievable bandwidth of the network stack is dramatically decreased [29].

Observation 1: *A large amount of traffic shaper's overhead comes from extrinsic operations such as interrupts and PCIe communications. Optimizing these operations has considerable potential for minimalizing the overhead of traffic shaping.*

B. Dilemma of Batching

In the network stack, batching is essential to achieve line-rate packet transmissions in high-speed network. It reduces the number of interrupts and amortizes the cost of context switches and PCIe operations.

We conduct two experiments to quantitatively illustrate the benefits of batching. In the first one, we use TBF to throttle the traffic rate to 8Gbps and measure the CPU load when sending a flow with different batch sizes. To exclude the CPU load other than traffic shaping, we also measure a baseline of CPU load by setting up a FIFO packet scheduler at the sender and throttling traffic at the switch. As shown in Fig. 3a, batching 16

packets can reduce the CPU load of traffic shaping by ∼5.9×[2]. This is mainly due to the reduction of software interrupts. Specifically, the number of software interrupts is reduced by ∼7× by batching 16 packets.

In the second experiment, we show that batching can still significantly reduce the CPU overhead without software interrupts. Fig. 3b shows the average CPU cycles of transmitting a packet with different batch sizes in Intel DPDK [32], which relies on a poll mode driver to send and receive packets. Batching 256 packets can reduce the CPU overhead by 9.5×. This is because batching amortizes the overhead of PCIe communications and context switches.

However, current traffic shaping schemes usually avoid batched packet transmissions. This is because batching introduces traffic bursts into the network, which can obscure congestion signals [6], [12] and even overwhelm buffers [18], degrading the transmission performance. We conduct a large-scale ns-3 [33] simulation to show the impairment of batching. We establish a 144-host leaf-spine topology with 9 leaf switches and 4 spine switches (more details in §V-E). Flows are generated in a one-to-one manner — in each host, one flow is generated at a time and its destination is randomly chosen. The flow size is chosen based on web search workload [34]. The rate of each flow is limited to 5Gbps by Carousel. Fig. 3c shows the average flow completion time (FCT) and batch size with different shaping granularities. Batching 34 packets can extend the FCT by ∼2×.

Observation 2: *Batching can significantly reduce the CPU overhead. However, current traffic shaping schemes usually avoid batched packet transmissions, as blindly employing batching will eventually impair the network performance, undermining the benefits brought by traffic shaping.*

C. Best of Both Worlds can be Achieved by Inter-flow Batching

Taking a closer look at the dilemma, we find that batching is not always harmful under any circumstances. Batching is harmful only when the induced traffic burst causes queue buildup inside the network. In other words, if the traffic burst does not result in queue buildup, it can be harmless.

We find that it is possible to deliberately create such bursts in data center networks. Our insight is that *intra-flow burst* is to blame for harming the network[3]. On the other hand, *inter-flow burst*, in which the packets are from different flows heading for different destinations, *can be naturally demultiplexed before queue buildup*. This is due to the unique traffic characteristics in the data center networks.

[2]Note that the CPU load of traffic shaping is given by the difference between the overall CPU load and the baseline CPU load. The baseline denotes the CPU load without traffic shaping. It is measured when the shaping is conducted in our switch.

[3]In this paper, we define a *flow* as a sequence of packets from a source host to a destination host. If each host has a unique IP address, then a flow can be identified by source IP address and destination IP address. Besides, we define a *connection* as a communication channel from a source socket to a destination socket. A connection is identified by a five-tuple: source IP address, source TCP/UDP port, destination IP address, destination TCP/UDP port, and IP protocol.

(1) In a host, flows heading for different destinations tend to take different routes. For some data center services, most traffic is destined for the servers in the same rack [35], [36]. In such services, flows with different destinations are demultiplexed at the ToR switch. Furthermore, data center networks usually have various paths between hosts. Load balance schemes such as ECMP will spread flows to different paths.

(2) Even if flows heading for different destinations partly share the same route, they can be finally demultiplexed before the congestion point. Studies [20], [21] have shown that most congestion in the data center network occurs at the last hop, where flows heading for different destinations have already been demultiplexed.

We conduct a large-scale simulation to demonstrate that inter-flow batching does not result in performance degradation. The simulation settings are the same as the previous ones, except that flows are generated in a fan-out manner — in each host, 16 flows are generated at a time, whose destinations are randomly chosen. Fig. 3d shows the average FCT with different shaping granularities. The results indicate that batching 36 packets achieves similar FCT performance to batching 1.5 packets. Note that this does not mean that there is no congestion, but inter-flow bursts are demultiplexed before the congestion points and thus do not lead to performance degradation.

Furthermore, inter-flow batching has become feasible with recent advances. Traditional shaping schemes employ separate shapers for different traffic classes, with which batching can inevitably result in intra-flow bursts. In contrast, recent advances have centralized the execution of traffic shaping. In other words, all flows are shaped through the same shaper. As a result, an elaborate batching scheme is feasible to batch inter-flow packets without inducing intra-flow bursts.

Observation 3: *Inter-flow burst is harmless to the network performance. A traffic shaping scheme only needs to focus on eliminating intra-flow bursts, while utilizing inter-flow batching to reduce its CPU overhead without degrading the network performance.*

D. Summary

Batching is essential to achieve fast traffic shaping on high-speed networks. Current traffic shaping schemes avoid batched packet transmissions to eliminate traffic bursts as they can lead to performance degradation. In contrast, we argue that traffic shaping schemes only need to focus on eliminating intra-flow bursts, and can utilize inter-flow batching to reduce CPU overhead without harming the network performance.

IV. FlowBundler Design

A. Design Rationale

FlowBundler aims to achieve the following three goals.

1) **Batching inter-flow packets:** FlowBundler should batch the transmissions of packets belonging to different flows to reduce per-packet CPU overhead.

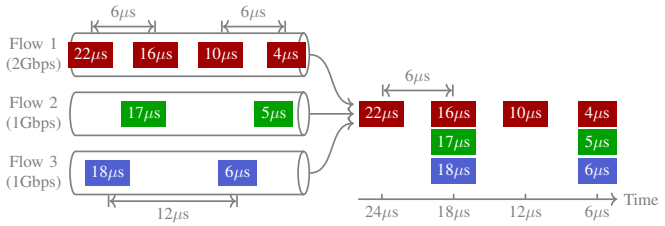


Fig. 4: An example of the basic idea

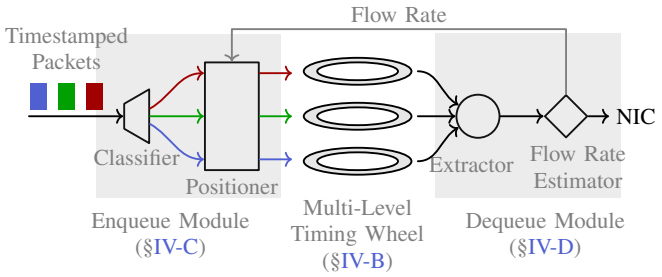


Fig. 5: FlowBundler overview

2) **Eliminating intra-flow burst:** FlowBundler should evenly space the packets belonging to the same flow.
3) **High CPU efficiency:** FlowBundler should finish every enqueue and dequeue operation within $O(1)$ time regardless of the number of concurrent flows.

To achieve these goals, our basic idea is to schedule packets based on both the shaping rates of their corresponding flows and the EDT of each packet. For example, as shown in Fig. 4, three flows are shaped by 2Gbps, 1Gbps, and 1Gbps, respectively. To eliminate intra-flow bursts for all flows (assuming that all packets are MTU-sized[4]), packet transmissions should be scheduled at a rate of 2Gbps (i.e., every 1500B/2Gbps=6μs), which is the *maximum shaping rate* among all flows. To batch inter-flow burst, when a transmission is scheduled, *all packets* whose EDTs are no further than now are dequeued and sent out. In this example, three inter-flow-batched packets are dequeued at 6μs and 18μs, and one packet of the 2Gbps flow is dequeued at 12μs and 24μs.

To realize the basic idea, FlowBundler needs an elaborate queue structure to properly arrange packets so that packets can be enqueued and dequeued quickly. Currently, the most efficient approach for EDT-based scheduling is Timing Wheel. However, Timing Wheel is not suitable for FlowBundler. This is because FlowBundler should schedule packets based on both flow shaping rate and EDT, while Timing Wheel only considers EDT. As a result, Timing Wheel cannot gather inter-flow-batched packets together and multiple dequeue operations are needed to dequeue these packets, which is inefficient.

To make FlowBundler aware of the flow shaping rate, we propose a Multi-Level Timing Wheel (MLTW) structure to schedule packets with high CPU efficiency and low memory footprint. MLTW contains multiple Timing Wheels with exponentially growing granularities. Each packet is placed into a proper Timing Wheel whose granularity best matches the flow shaping rate. In this way, FlowBundler can properly position and transmit packets based on their shaping rate, while accommodating various flows with a wide disparity of shaping rates (more details in §IV-B).

Putting them together, we design FlowBundler with a structure depicted in Fig. 5. As Carousel and Eiffel, FlowBundler is based on the EDT model. Specifically, FlowBundler assumes that the EDT of each packet has already been determined at the time of its arrival. The EDT can either be calculated at per-connection level for connection-based traffic shaping or at per-destination level for destination-based traffic shaping. FlowBundler can be divided into three parts: an enqueue module, an MLTW queue, and a dequeue module. The enqueue module classifies incoming packets according to their destinations and puts them into the proper position in the MLTW. The dequeue module extracts packets from MLTW at a frequency of the maximum shaping rate among all flows. Then it estimates the flow shaping rate and sends the packets to the NIC.

[4]If not all packets are MTU-sized, FlowBundler can generate intra-flow bursts consisting of several packets. Nevertheless, the maximum intra-flow burst is no longer than 2 MTUs.

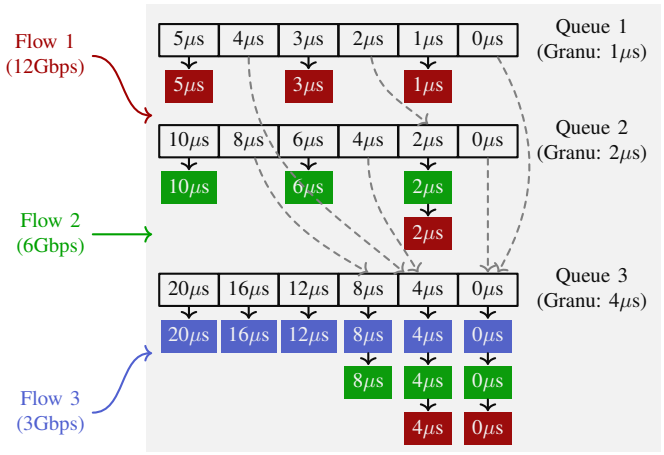


Fig. 6: An example of Multi-Level Timing Wheel with 1μs minimum granularity and 3 Timing Wheels

B. Multi-Level Timing Wheel

As is mentioned at §I, how packets are arranged in the queue determines both the CPU and memory overhead of FlowBundler. In this part, we present a novel queue structure, Muti-Level Timing Wheel (MLTW), which can quickly enqueue and dequeue packets despite the varying flow shaping rate with a small memory footprint.

Basic structure: As shown in Fig. 6, MLTW is comprised of multiple Timing Wheels with exponentially growing granularities (the Timing Wheel structure has been described in §II and we do not repeat here). For example, if the minimum granularity is simply set to 1ns (which is small enough as it can support a maximum shaping rate of 12Tbps), then the granularity of i-th queue is 2^{i-1}ns. The number of queues depends on the minimum supported shaping rate. For example,

if the minimum supported rate is 10Kbps, then the coarsest granularity is 1500B/10Kbps = 1.2s, and 32 queues are sufficient (as 2^{31}ns > 1.2s). The length of each queue depends on the largest volume of resident traffic for each flow. In Linux, TSQ [37] usually limits the amount of data inside the network stack to 128KB per connection by default. Thus, 128 slots per queue are sufficient (note that multiple packets are put into the same slot if the packet size is smaller than 1500B). In total, an MLTW only needs 4096 slots, with which it can support 1ns granularity while holding packets in a time range of [0, 256s].

Packet placement: To eliminate intra-flow bursts and accommodate packets with wide time ranges, a packet is put into the queue whose granularity best matches the flow shaping rate Specifically, for a flow with a shaping rate of R_f, the packets are put into the $(\lfloor \log_2 MTU/R_f/g_{\min} \rfloor + 1)$-th queue, where $g_{\min}$ is the minimum granularity (the logarithm calculation can be achieved by bit operations, as shown in §IV-C). For example, in Fig. 6, the shaping rate of Flow 1 is 12Gbps and its packets should be sent every 1μs. The packets of the flow are enqueued to Queue 1 (1μs granularity). In the same way, packets of Flow 2 (6Gbps shaping rate) and Flow 3 (3Gbps shaping rate) are enqueued to Queue 2 (2μs granularity) and Queue 3 (4μs granularity), respectively. In this way, the intra-flow burst is no larger than 2 MTUs[5].

Furthermore, to achieve high dequeue efficiency, MLTW gathers inter-flow-batched packets into the same slot. Specifically, as one has noticed, multiple queues can have the same slot time. For example, in Fig. 6, Queue 1-3 all contain a 4μs slot. Thus, at time 4μs, a dequeue operation should be executed three times, which is inefficient. To reduce the dequeue overhead, MLTW accumulates all packets in these slots into the slot on the coarsest-granularity queue. (i.e., the queue with the largest index). Note that this does not incur any shaping error, as the transmission time of the target slot is the same as the original one. For example, in Fig. 6, all packets in 4μs slots are placed into Queue 3. In this way, MLTW can gather inter-flow-batched packets in the same slot, reducing the dequeue overhead.

C. Enqueue Module

For each incoming packet, the enqueue module first classifies them into different flows. Then it places the packet into the proper position in the MLTW queue.

Classifying packets: FlowBundler needs to quickly classify packets into flows according to their destination host. We assume that a host can be identified by its IP address. Nevertheless, our approach can also extend to scenarios that a host contains more than one IP address. In such scenarios, FlowBundler can classify packets based on explicit mappings between IP addresses and host ids.

A straightforward way of classifying is hashing. However, simply employing hashing can incur non-negligible overhead. This is because an individual server may communicate with thousands of destinations [18]. FlowBundler needs to employ a large hash table or handle frequent hash collisions.

We observe two facts that to simplify the hashing. (1) The hash table does not need to be too large. A larger hash table can differentiate more flows and thus can batch more inter-flow packets. Nonetheless, the benefit brought by batching is undermined as the increase of the number of batched packets. For example, our experiments show that batching more than 128 packets does not bring more benefits (Fig. 3b). Thus, even though there might be thousands of destinations, FlowBundler does not need to exactly differentiate them. Instead, FlowBundler only needs to classify packets into hundreds of categories. Thus, a hash table with 512 buckets is sufficient.

(2) There is no need to resolve hash collisions. With a hash collision, two flows are considered as the same one. This does not bring any negative effects[6] except that the number of batched packets is reduced by one. More often than not, this has little impact on the CPU overhead as the number of flows is usually not small (e.g., with 10 flows and 512 buckets, the probability of at least one hash collision 8.4%).

Positioning packets: The enqueue module puts every incoming packet into the proper slot in the MLTW. Algorithm 1 shows the pseudocode. Overall, there are three steps.

(1) FlowBundler finds the queue id whose granularity best matches the flow shaping rate (Line 9). More precisely, for a flow with shaping rate R_f, the time gap between its successive MTU-sized packets is MTU/R_f. The best-match queue id is given by $\lfloor \log_2(MTU/R_f/g_{\min}) \rfloor$, where $g_{\min}$ is the minimum granularity (i.e., the granularity of Queue 1). As calculating logarithm is non-trivial, we replace it with the equivalent FLS (Find Last Set) operation as $\text{FLS}(MTU/g_{\min}) - \text{FLS}(R_f)$, where FLS is a fast bit operation to find the position of the most significant set bit. Modern CPUs contain instructions to finish this operation in several CPU cycles [14].

(2) Occasionally, the EDT of the packet can be larger than the time of the tail slot. For example, when the shaping rate of a connection is dramatically decreased, the inter-packet interval can be so large that the EDT of the latter arrived packets can be even larger than the time of the tail slot. In this case, FlowBundler moves the packet to a queue with coarser granularity whose time range can surround EDT (Line 11-15).

(3) To accumulate inter-flow-batched packets, FlowBundler puts the packet into the coarsest-granularity queue that contains a slot with the same transmission time as the current queue (Line 16-21). Let qid denote the index of the current queue and $slotid$ denote the slot index. The $(qid+1)$-th queue contains a slot with the same transmission time if and only

[5]If the shaping rate does not fit a Timing Wheel's granularity exactly and has to be rounded, the intra-flow burst can reach 2 MTUs. For example, an 8Gbps flow sends a packet every 1.5μs. It will be put into the 1μs-spaced Timing Wheel and two packets will be put into the same slot every other slot (assuming that all packets are MTU-sized).

[6]Note that considering two flows as the same one has no impact on the shaping accuracy. As long as the packets of each flow have been correctly time-stamped with EDT, FlowBundler can accurately shape the flow's traffic based on its desired shaping rate. Classifying only affects how packets are batched, while when a packet is transmitted is determined by its EDT.

Algorithm 1 Enqueue a packet to MLTW

```
1: function ENQUEUE(packet, rate)
2:     packet.is_app_limited = (packet.edt ⩽ now)
3:     ▷ Classify packets according to destination IP address ◁
4:     flow ← CLASSIFY(packet.dst_ip)
5:     ▷ Clamp EDT in the time range of MLTW ◁
6:     packet.edt ← MAX(packet.edt, front_time)
7:     packet.edt ← MIN(packet.edt, front_time+slot_num×
       max_gran)
8:     ▷ Find the queue to put the packet in. ◁
9:     qid ← FLS(max_supported_rate) − FLS(flow.rate)
10:    ▷ If EDT exceeds the time range of current queue, move
         the packet to lower queues ◁
11:    if packet.edt > GETTAILTIME(qid) then
12:        howfar ← packet.edt − front_time
13:        timerange ← slot_num × (min_gran ≪ qid)
14:        descend ← FLS(howfar/timerange)
15:        qid ← qid + descend
16:    slotid ← FINDSLOTID(qid, packet.edt)
17:    ▷ Accumulate inter-flow-batched packets in the same slot ◁
18:    real_qid ← qid + FFS(slotid)
19:    real_sid ← FINDSLOTID(real_qid, packet.edt)
20:    ▷ Put the packet into the slot ◁
21:    MLTW[real_qid][real_sid%slot_num].APPEND(packet)
22:    ▷ Each bit of bitmap indicates whether a queue is empty ◁
23:    bitmap ← bitmap | (1 ≪ real_qid)
24: function FINDSLOTID(qid, send_time)
25:    granularity ← min_gran ≪ qid
26:    return send_time/granularity
```

Algorithm 2 Dequeue packets from MLTW

```
1: function DEQUEUE()
2:     ▷ Find the queue id with the maximum shaping rate ◁
3:     highest_qid ← MIN(FFS(bitmap), queue_num)
4:     granularity ← min_gran ≪ highest_qid
5:     while now ⩾ front_time do
6:         slotid ← FINDSLOTID(highest_qid, front_time)
7:         real_qid ← highest_qid + FFS(slotid)
8:         real_slotid ← FINDSLOTID(real_qid, front_time)
9:         if MLTW[real_qid][real_slotid].ISEMPTY() then
10:            front_time ← front_time + granularity
11:        else
12:            packets ← MLTW[real_qid][real_slotid].POP()
13:            if MLTW[real_qid].ISEMPTY() then
14:                bitmap ← bitmap & ∼(1 ≪ real_qid)
15:            ESTIMATERATE(packets, real_slotid)
16:            return packets
17:    return NULL
```

if $slotid$ is even (recall that the granularity of $(qid+1)$-th queue is twice the granularity of the qid-th queue). The corresponding slot index of $(qid+1)$-th queue is $slotid/2$. The target queue and slot can be found by recursively seeking the slot of the next queue until the slot index becomes odd. To condense to one step, the number of trailing zeros of $slotid$'s binary representation determines how many times should we increment the queue index. Thus, the target queue id can be given by $qid + \text{FFS}(slotid)$, where FFS is a fast bit operation to find the first set bit. Modern CPUs contain instructions to finish this operation in several CPU cycles [14].

D. Dequeue Module

The dequeue module first extracts the packets allowed to be sent from MLTW, and then it estimates the flow shaping rate.

Extracting packets: The dequeue module extracts packets whose EDTs are no larger than now. Algorithm 2 shows the pseudocode. At its heart, the dequeue module visits MLTW at a frequency of maximum shaping rate among active flows. This improves FlowBundler's adaptability to the drastic change of shaping rate. Specifically, FlowBundler first finds the queue index with the maximum shaping rate (Line 3), which is the non-empty queue with the smallest index. Then FlowBundler iteratively visits each slot of the queue to check whether it contains packets (Line 5-16). In each iteration, FlowBundler locates the actual slot containing packets (Line 6-8) and extracts packets, if any, from it (Line 12).

Estimating flow shaping rate: When a flow is only shaped by one policy, the flow shaping rate can be directly obtained. However, in some cases, it is not easy to directly determine the overall shaping rate of a flow. For example, FlowBundler can aggregate multiple connections to the same flow if they head for the same destination.

In these cases, FlowBundler uses a simple algorithm to dynamically estimate the overall shaping rate for each flow based on the traffic departure rate. Specifically, FlowBundler monitors the amount of outgoing traffic (denoted by S) every T_r time. The instantaneous traffic rate can be given by S/T_r. Then FlowBundler uses a low-pass filter (i.e., exponentially weighted moving average) to update the shaping rate by $R_f \leftarrow (1-\alpha)\cdot R_f + \alpha \cdot \frac{S}{T_r}$, where α is the time constant of the low pass filter. The low-pass filter can smooth transient traffic rate oscillations.

Furthermore, to quickly and accurately estimate the traffic rate, we make T_r a dynamical value based on the shaping rate. Specifically, FlowBundler calculates the shaping rate for every 5 sent packets (assuming that packets are MTU-sized).

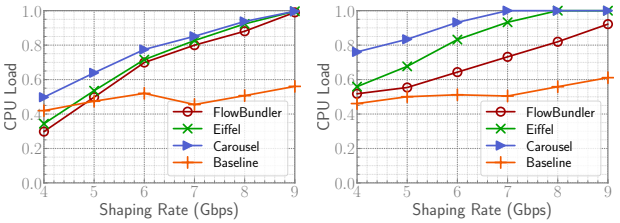
V. Implementation and Evaluation

We implement FlowBundler in kernel space as a new Linux queueing discipline (`qdisc`) kernel module and in user space with BESS [22] (previously known as SoftNIC [16]), which is a software NIC based on Intel DPDK [32].

A. Methodology

Testbed setup: We set up two testbeds (with 10Gbps and 100Gbps speed) for evaluation. The 10Gbps testbed contains two servers and a server-emulated switch. Each server is equipped with an Intel 82599 10GbE NIC, an 8-core Intel Xeon E5-2620 v4 2.10GHz CPU, and 16GB memory. The 10GbE NICs are connected to the server-emulated switch, which is equipped with an Intel XL710 quad-port 10GbE NIC. We evaluate the kernel performance on the 10Gbps testbed as the kernel performance is not able to achieve 100Gbps shaping rate with a single core.

The 100Gbps testbed contains two servers. Each server is equipped with an NVIDIA Mellanox ConnectX-6 Dx dual-port 100GbE NIC, a 6-core Intel i5-10400 2.90GHz CPU, and



(a) CPU load: 1 flow (b) CPU load: 16 flows

Fig. 7: [Kernel] CPU load and frequency of software interrupts with different numbers of flows

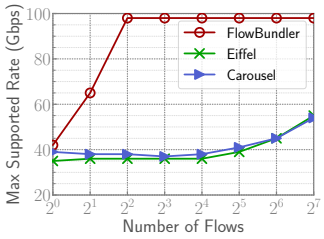


Fig. 8: [Userspace] Max supported shaping rate

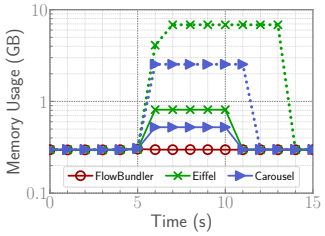


Fig. 9: [Kernel] Memory efficiency of FlowBundler

16GB memory, We evaluate the userspace performance on the 100Gbps testbed.

As our testbed only contains one physical sender and one physical receiver, we vary the destination IP addresses to emulate multiple (virtual) destinations.

Compared Schemes: We compare FlowBundler to Carousel [18] and Eiffel [14]. For kernel experiments, we use the open source code provided by authors [38], [39]. For userspace experiments, we implement them on BESS following the papers and their kernel implementations.

Parameter Settings: For FlowBundler, the minimum granularity is set to 1ns. There are 64 Timing Wheels and each Timing Wheel has 1024 slots[7]. For the rate estimator, we set $\alpha = 0.25$ and the initial value of the estimated flow rate to 20Gbps. For Carousel and Eiffel, we set the granularity to 1μs in the 10Gbps network and 100ns in the 100Gbps network. The reason for this setting is that it is sufficient for Carousel and Eiffel to eliminate intra-flow bursts and the granularity of 1ns can raise unacceptable overhead for them. Their horizons are set to 10s.

B. CPU Efficiency

Kernel performance: We use `mpstat` to measure the overall CPU overhead and frequency of software interrupts. To ensure that all packets are processed in a single CPU core, we bind the process of `iPerf3` and all RX interrupts to the same CPU. The measured overall overhead includes TCP/IP stack processing, which is not a small part. To quantify such overhead, we conduct a baseline experiment in which traffic is shaped at the server-emulated switch. Specifically, we install a FIFO `qdisc` at the sender, whose overhead is minimal. We throttle the overall traffic rate at the server-emulated switch with the TBF `qdisc`. The TBF `qdisc` is configured with a 1GB buffer to avoid packet loss, which may introduce extra TCP processing overhead at the sender.

Fig. 7 shows the CPU load with 1 flow and 16 flows. In the case of 1 flow, FlowBundler does not batch packets. Nevertheless, it still achieves slightly lower CPU overhead than both Eiffel and Carousel, whose enqueue and dequeue operations are very fast (i.e., $O(1)$). This indicates that FlowBundler's overheads of enqueue and dequeue operations are as small as those of Eiffel and Carousel.

With more flows, FlowBundler incurs a lower CPU load than Eiffel and Carousel. Specifically, in the case of 16 flows, FlowBundler achieves ∼12-28% and ∼7-20% lower CPU load than Carousel and Eiffel, respectively. This is because FlowBundler can send packets in batch, reducing the number of interrupts, context switches, and PCIe operations.

Userspace performance: As DPDK works in polling mode, the CPU overhead reported by `mpstat` is always 100%. Thus, we use the maximum supported shaping rate to reflect the CPU overhead. The maximum supported shaping rate is the maximum shaping rate with which a shaping scheme is able to achieve high shaping accuracy. More precisely, we increase the shaping rate and measure the absolute deviation between the achieved shaping rate and the target shaping rate at the receiver. We mark the maximum shaping rate when the deviation is within 1% as the maximum supported rate.

Fig. 8 shows the maximum supported rate with different numbers of flows. FlowBundler can support a shaping rate of 98Gbps with 4 flows, which is 2.6× better than Carousel and 2.7× better than Eiffel.

C. Memory Efficiency

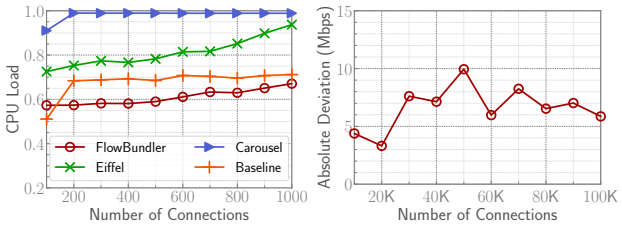
In this experiment, we compare the memory efficiency of the internal data structures in FlowBundler, Eiffel, and Carousel, respectively. We first monitor the memory usage of a FIFO `qdisc`, which requires minimal memory usage, for 5 seconds. Then we insert the `qdisc` of FlowBundler / Eiffel / Carousel at $t = 5$s and monitor the memory usage for another 5 seconds. Finally, we delete the `qdisc`.

As shown in Fig. 9, FlowBundler consumes little extra memory. It only requires ∼1.1MB memory to achieve a very fine granularity of 1ns and a broad horizon of 1.8×10^{10} seconds. In comparison, Eiffel and Carousel consume a considerable amount of memory. With a granularity of 1μs (solid line), Eiffel and Carousel require ∼540MB and ∼236MB memory, respectively. To evenly space packets in 40/100Gbps network, Eiffel and Carousel need finer granularity. With a granularity of 0.1μs (dashed line), Eiffel and Carousel require ∼6.9GB and ∼2.3GB memory, respectively.

D. Scalability

In this part, we evaluate whether FlowBundler can scale to a large number of connections.

[7]These values are more than what is mentioned previously. We enlarge the number of slots as in BESS experiments a sender can generate 1MB data for each connection at each time to reduce the packet generation overhead. We enlarge the number of Timing Wheels in case a packet's EDT is too far (from now) to be surrounded by MLTW.



(a) Kernel (b) Userspace

Fig. 10: Scalability

Kernel performance: Following [14], we use `neper` to generate traffic, which allows us to generate up to 1,000 concurrent TCP connections. The overall traffic rate is throttled to 3Gbps. Fig. 10a shows the CPU load with different numbers of connections. FlowBundler achieves better scalability than Carousel and Eiffel. Specifically, when the number of concurrent connections grows from 100 to 1,000, the CPU load increment is ∼10% for FlowBundler, which is ∼2.1× better than Eiffel.

Userspace performance: We use our own traffic generator to generate up to 100K connections. We throttle the overall traffic rate to 50Gbps and measure the shaping rate every 100ms at the receiver. We examine the absolute deviation of measured shaping rate from target shaping rate. As Eiffel and Carousel struggle to achieve a 50Gbps rate, we only evaluate the performance of FlowBundler. Fig. 10b shows the absolute deviation of FlowBundler with different numbers of connections. FlowBundler can consistently achieve high shaping accuracy with 10K - 100K connections.

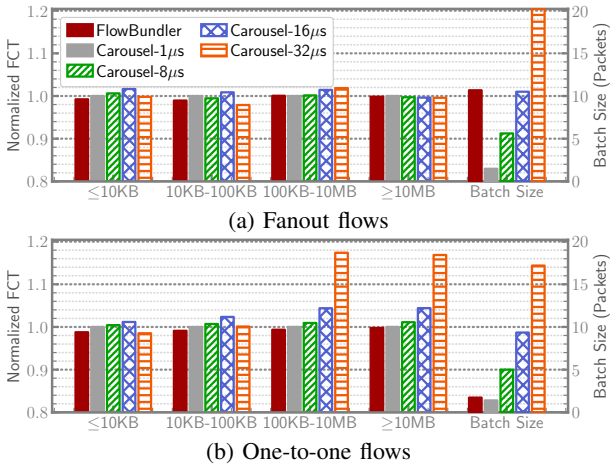
E. Large-scale Simulations

In this part, we use ns-3 [33] simulations to demonstrate that FlowBundler can batch packet transmissions without degrading the network performance.

Topology: We establish a 144-host leaf-spine topology with 9 leaf (ToR) switches and 4 spine (core) switches. Each leaf switch has 16 10Gbps downlinks connecting to hosts and 4 40Gbps uplinks connecting to spine switches, forming a non-blocking network. We employ ECMP to load balance flows among various paths. Each switch has a 512KB buffer per port. The end-to-end round-trip time across the spine is 85.2μs. We use DCTCP [34] as the transport protocol, and the ECN threshold is set to 17KB, as suggested in [40].

Workloads: There are two kinds of traffic patterns. (i) One-to-one traffic: a sender establishes a connection to a receiver. (ii) Fanout traffic: a sender establishes 16 concurrent connections to different receivers. Half servers are generating the former kind of traffic. The others are generating the latter. The sources and destinations are randomly chosen. The flow size distribution follows a realistic workload derived from a data center running web search service [34]. Flow arrivals follow a Poisson process. The network load is 80%. We generate 111,733 flows in each simulation.

Performance metrics: We use the Flow Completion Time (FCT) to denote the transmission performance. We normalize the FCT to the values achieved by Carousel with 1μs granularity for clear comparison. We cannot directly measure the CPU overhead of end hosts in ns-3[8]. Instead, we use the number of batched packets to reflect the CPU overhead. It is measured by counting the number of packets in each slot.



(a) Fanout flows

(b) One-to-one flows

Fig. 11: FCT across different flow sizes (first 4 groups) and batch size (last group) in large-scale simulations

Simulation results: Fig. 11 shows the average flow completion time (FCT) across different flow sizes as well as the number of batched packets at end hosts. FlowBundler can batch packets without degrading the FCT performance. Specifically, FlowBundler achieves the shortest FCT performance for both fanout flows (Fig. 11a) and one-to-one flows (Fig. 11b). Meanwhile, FlowBundler can batch ∼7.3× more packets than Carousel with 1μs granularity for fanout flows. In comparison, Carousel with coarser granularity (i.e., 8μs, 16μs, and 32μs) batches more packets at the expense of degrading FCT performance.

VI. Conclusion

Software traffic shaping incurs high CPU overhead. Blindly batching packet transmissions can reduce CPU overhead while resulting in traffic bursts, which can degrade network performance. In this paper, we argue that inter-flow burst can be naturally demultiplexed and is thus harmless. We present FlowBundler, which can reduce the CPU overhead of traffic shaping through inter-flow batching. Experiments show that FlowBundler achieves much higher CPU and memory efficiency than state-of-the-art approaches.

Open source. The authors have provided public access to their code and/or data at https://github.com/ants-xjtu/FlowBundler.

Acknowledgments. We thank the anonymous reviewers for their constructive comments. This work is supported by the National Key R&D Program of China (No. 2022YFB2901700), the National Natural Science Foundation of China (No. 61902307, 61972421, 62172323, 62272382, and 61872208), and the Fundamental Research Funds for the Central Universities (No. xzy012020014). Peng Zhang is the corresponding author.

[8]Our simulation focuses on evaluating the transmission performance, and the overheads of PCIe communications and interrupts are ignored.

References

[1] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, A. Greenberg, and C. Kim, "EyeQ: Practical Network Performance Isolation at the Edge," in *USENIX NSDI*, 2013.

[2] A. Kumar, S. Jain, U. Naik, *et al.*, "BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing," in *ACM SIGCOMM*, 2015.

[3] K. Jang, J. Sherry, H. Ballani, and T. Moncaster, "Silo: Predictable Message Latency in the Cloud," in *ACM SIGCOMM*, 2015.

[4] M. Dalton, D. Schultz, J. Adriaens, *et al.*, "Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization," in *USENIX NSDI*, 2018.

[5] P. Kumar, N. Dukkipati, N. Lewis, *et al.*, "PicNIC: Predictable Virtualized NIC," in *ACM SIGCOMM*, 2019.

[6] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda, "Less is More: Trading a Little Bandwidth for Ultra-low Latency in the Data Center," in *USENIX NSDI*, 2012.

[7] Y. Zhu, H. Eran, D. Firestone, *et al.*, "Congestion Control for Large-Scale RDMA Deployments," in *ACM SIGCOMM*, 2015.

[8] R. Mittal, V. T. Lam, N. Dukkipati, *et al.*, "TIMELY: RTT-based Congestion Control for the Datacenter," in *ACM SIGCOMM*, 2015.

[9] A. Kalia, M. Kaminsky, and D. Andersen, "Datacenter RPCs can be General and Fast," in *USENIX NSDI*, 2019.

[10] W. Cheng, K. Qian, W. Jiang, T. Zhang, and F. Ren, "Re-architecting Congestion Management in Lossless Ethernet," in *USENIX NSDI*, 2020.

[11] G. Kumar, N. Dukkipati, K. Jang, *et al.*, "Swift: Delay is Simple and Effective for Congestion Control in the Datacenter," in *ACM SIGCOMM*, 2020.

[12] D. Shan and F. Ren, "Improving ECN Marking Scheme with Micro-burst Traffic in Data Center Networks," in *IEEE INFOCOM*, 2017.

[13] N. Beheshti, Y. Ganjali, M. Ghobadi, N. McKeown, and G. Salmon, "Experimental Study of Router Buffer Sizing," in *ACM IMC*, 2008.

[14] A. Saeed, Y. Zhao, N. Dukkipati, *et al.*, "Eiffel: Efficient and Flexible Software Packet Scheduling," in *USENIX NSDI*, 2019.

[15] J.-P. Billaud and A. Gulati, "hClock: Hierarchical QoS for Packet Scheduling in a Hypervisor," in *ACM EuroSys*, 2013.

[16] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "SoftNIC: A Software NIC to Augment Hardware," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-155, May 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html.

[17] M. Shahbaz, S. Choi, B. Pfaff, *et al.*, "PISCES: A Programmable, Protocol-Independent Software Switch," in *ACM SIGCOMM*, 2016.

[18] A. Saeed, N. Dukkipati, V. Valancius, V. The Lam, C. Contavalli, and A. Vahdat, "Carousel: Scalable Traffic Shaping at End Hosts," in *ACM SIGCOMM*, 2017.

[19] M. Marty, M. de Kruijf, J. Adriaens, *et al.*, "Snap: A Microkernel Approach to Host Networking," in *ACM SOSP*, 2019.

[20] A. Singh, J. Ong, A. Agarwal, *et al.*, "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network," in *ACM SIGCOMM*, 2015.

[21] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy, "High-resolution Measurement of Data Center Microbursts," in *ACM IMC*, 2017.

[22] "BESS: Berkeley Extensible Software Switch." (), [Online]. Available: http://span.cs.berkeley.edu/bess.html.

[23] "tc-tbf." (Dec. 13, 2001), [Online]. Available: https://man7.org/linux/man-pages/man8/tc-tbf.8.html.

[24] "tc-htb." (Jan. 10, 2002), [Online]. Available: https://man7.org/linux/man-pages/man8/tc-htb.8.html.

[25] "tcp: switch to Early Departure Time model." (Sep. 21, 2018), [Online]. Available: https://lwn.net/Articles/766564/.

[26] S. Fomichev, E. Dumazet, W. de Bruijn, V. Dumitrescu, B. Sommerfeld, and P. Oskolkov, "Replacing HTB with EDT and BPF," in *Netdev 0x14*, 2020.

[27] "pkt_sched: fq: Fair Queue packet scheduler." (Aug. 29, 2013), [Online]. Available: https://lwn.net/Articles/565421/.

[28] G. Varghese and T. Lauck, "Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility," in *ACM SOSP*, 1987.

[29] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, "Understanding PCIe Performance for End Host Networking," in *ACM SIGCOMM*, 2018.

[30] M. Flajslik and M. Rosenblum, "Network Interface Design for Low Latency Request-Response Protocols," in *USENIX ATC*, 2013.

[31] B. Stephens, A. Akella, and M. Swift, "Loom: Flexible and Efficient NIC Packet Scheduling," in *USENIX NSDI*, 2019.

[32] "Intel DPDK." (), [Online]. Available: https://www.dpdk.org/.

[33] "ns-3." (), [Online]. Available: https://www.nsnam.org/.

[34] M. Alizadeh, A. Greenberg, D. A. Maltz, *et al.*, "Data Center TCP (DCTCP)," in *ACM SIGCOMM*, 2010.

[35] T. Benson, A. Akella, and D. A. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," in *ACM IMC*, 2010.

[36] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the Social Network's (Datacenter) Network," in *ACM SIGCOMM*, 2015.

[37] *TCP small queues*, Jul. 17, 2012. [Online]. Available: https://lwn.net/Articles/507065/.

[38] "Kernel Implementation of Carousel." (), [Online]. Available: https://github.com/saeed/eiffel_linux/tree/v4.10-gq.

[39] "Kernel Implementation of Eiffel." (), [Online]. Available: https://github.com/saeed/eiffel_linux/tree/working_ffs-based_qdisc.

[40] M. Alizadeh, A. Javanmard, and B. Prabhakar, "Analysis of DCTCP: Stability, Convergence, and Fairness," in *ACM SIGMETRICS*, 2011.