Fast and Accurate Software Traffic Shaping With Inter-Flow Batching

Danfeng Shan, Shihao Hu, Yike Liu, Hao Li, Yazhe Tang, Peng Zhang, Wanchun Jiang, *Member, IEEE*, and Fengyuan Ren

***Abstract*—Traffic shaping is a fundamental function at end hosts. While software-based shapers are more flexible to develop and deploy compared to their hardware counterparts, they often struggle to achieve speeds of 40Gbps and beyond. Previous efforts have largely focused on reducing *intrinsic* overheads. However, we find that *extrinsic* overheads, such as PCIe communication and interrupts, pose significant barriers to higher performance. Batching, a common technique in operating systems, can effectively amortize these overheads, but blindly batching degrades the transmission performance as it introduces traffic burst into the network. Diving into the dilemma, we find that *intra-flow burst* is the main culprit for harming the network performance, whereas *inter-flow burst*, consisting of packets from different flows, can often be naturally demultiplexed within the network. Leveraging the insight, we propose 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 — 2.6× higher than state-of-the-art approaches. Large-scale simulations confirm that FlowBundler preserves network performance while effectively batching packet transmissions.**

***Index Terms*—Traffic shaping, rate limiting, batching, traffic burst, data center networks.**

I. Introduction

IN DATA centers, traffic shaping — also referred to as rate limiting or packet pacing — is a critical component at end hosts. For instance, in cloud environments, where numerous tenants and applications share network bandwidth, rate limiting is essential for ensuring bandwidth isolation among tenants and applications [2], [3], [4], [5], [6]. At the transport layer, congestion control relies on traffic shaping for two key purposes. First, many congestion control algorithms are rate-based [7], [8], [9], [10], [11], [12]. They naturally require rate limiters for fine-grained adjustments to transmission rates. Second, packet pacing helps mitigate traffic bursts, which can obscure congestion signals [7], [13] and lead to packet drops in shallow-buffered switches [14].

Traffic shaping can be implemented using either software or hardware. Compared to hardware-based solutions, software-based approaches offer greater flexibility for deployment and development [15], and eliminate the need for dedicated hardware, making them highly appealing for use in production networks [4], [6], [15], [16], [17], [18], [19], [20]. Nevertheless, software traffic shaping incurs high CPU overhead [19]. Recently, several efforts [15], [19] have been made to minimize 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 [6], [12], [19]. Despite recent advances, state-of-the-art approaches still struggle to achieve accurate traffic shaping in high-speed networks (i.e., 40-100Gbps). With intrinsic overheads in software traffic shaping already minimized, one might question whether software traffic shaping have reached its performance ceiling.

In this paper, we argue that significant untapped potential remains for software traffic shaping. In addition to intrinsic overheads, traffic shaping incurs high *extrinsic* overheads that hinders software traffic shaping schemes from achieving higher speed. Specifically, traffic shaping generates massive interrupts and PCIe communications on a per-packet basis. These operations dominate the overhead of shaping traffic in high-speed networks.

In network stacks, batching is a widely used and effective technique for amortizing these overheads by processing multiple packets together, thereby consolidating the interrupts and PCIe communications across several packets. Indeed, our experiment results demonstrate that batching can effectively reduce the CPU overhead by 5.9-9.5× (§III-B). However, despite its efficiency, current traffic shaping schemes usually avoid employing this common technique because it introduces traffic bursts — the opponent of traffic shaping — into the network, which impairs the transmission performance. Our results show that blindly batching can degrade the transmission performance by ∼ 2× (§III-B).

Facing the trade-off, we ask: Is it feasible to achieve the best of both worlds? Diving into the dilemma, we find that

Received 17 January 2025; revised 24 June 2025; accepted 4 August 2025; approved by IEEE TRANSACTIONS ON NETWORKING Editor S. Alouf. Date of publication 27 October 2025; date of current version 30 December 2025. This work was supported in part by the National Key Research and Development Program of China under Grant 2022YFB2901700; and in part by the National Natural Science Foundation of China under Grant 62372363, Grant 62132007, Grant 62172323, Grant 62272382, and Grant 624722451. The preliminary version of this paper was published in the Proceedings of IEEE INFOCOM 2023 [DOI: 10.1109/INFOCOM53939.2023.10229082]. *(Corresponding author: Yazhe Tang.)*

Danfeng Shan, Shihao Hu, Yike Liu, Hao Li, Yazhe Tang, and Peng Zhang are with the School of Computer Science and Technology, Xi'an Jiaotong University, Xi'an 710049, China (e-mail: dfshan@xjtu.edu.cn; hushihaoxjtu@163.com; cn-lyk@stu.xjtu.edu.cn; hao.li@xjtu.edu.cn; yztang@xjtu.edu.cn; p-zhang@xjtu.edu.cn).

Wanchun Jiang is with the School of Information Science and Engineering, Central South University, Changsha 410083, China (e-mail: jiangwc@csu.edu.cn).

Fengyuan Ren is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China (e-mail: renfy@tsinghua.edu.cn).

This article has supplementary downloadable material available at https://doi.org/10.1109/TON.2025.3598059, provided by the authors.

Digital Object Identifier 10.1109/TON.2025.3598059

it is *intra-flow burst*, which consists of packets from the same flow, that should be blamed for harming the network performance. In contrast, *inter-flow burst*, which consists of packets from different flows, is less harmful to the network. Specifically, traffic bursts only impair performance when they cause queue buildup within the network, while *inter-flow burst can be naturally demultiplexed (i.e., separated and routed along different paths) before reaching the congestion point* due to the unique traffic characteristics of the data center networks. For instance, most congestion occurs at the last hop [21], [22], 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 compromising network performance (§IV). FlowBundler schedules packet transmissions at the maximum shaping rate among all flows, ensuring that intra-flow bursts are eliminated for every flow. At each scheduled transmission time, FlowBundler retrieves packets from all flows whose expected departure times are due. This allows packets from different flows to be dequeued and transmitted together in a batch, reducing the extrinsic overheads.

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 overhead of auxiliary data structure 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., 64) are required to cover a wide time horizon (0-256s), and only small memory overhead 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, improving the dequeue efficiency.

We implement FlowBundler in the kernel space as a Linux `qdisc` kernel module and in the user space based on BESS [23]. We evaluate FlowBundler on both 10Gbps and 100Gbps testbeds (§VI). 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× (§VI-C). FlowBundler consumes three orders of magnitude less memory than Carousel and Eiffel to accommodate flows with a wide disparity of shaping rates (§VI-D). Furthermore, FlowBundler can scale to 100K flows (§VI-E). 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 (§VI-F). Simulation results confirm that FlowBundler can batch packet transmissions without degrading the network performance.

The code of FlowBundler is available at https://github.com/ants-xjtu/FlowBundler.

II. Software Traffic Shaping at End Hosts

Traffic shaping, also known as rate limiting and packet pacing, is designed to control packet transmissions at a specified rate. Traffic shaping is non-work-conserving at the global level — they may hold packets even when the link is idle to enforce rate limits. However, within their configured rate limits, traffic shapers can exhibit work-conserving properties. For example, when the aggregate traffic demand is below the configured rate limit, some shaping algorithms allow traffic classes to fully utilize available bandwidth by redistributing unused capacity among active classes. In this section, we provide some background on software traffic shaping mechanisms at end hosts, covering both traditional token-bucket-based approaches and modern EDT-based schemes that form the foundation for our work.

A. Token-Bucket-Based Traffic Shaping

Traditional traffic shaping mechanisms, such as the Token Bucket Filter (TBF) [24] and Hierarchical Token Bucket (HTB) [25], rely on the token bucket algorithm to regulate traffic rate. In the token bucket algorithm, tokens—each representing a unit of bytes—are generated at a rate corresponding to the desired throttling rate. The tokens are accumulated in a bucket with a fixed capacity. When a packet arrives, it is permitted to pass through only if there are enough tokens in the bucket. Otherwise, the packet is either dropped or buffered until enough tokens become available.

While individual shaping algorithms are computationally efficient [19], they face significant scalability challenges. Specifically, employing various rate limiting policies for different traffic classes requires each class to maintain its own shaper and queue. For systems with tens of thousands of traffic classes, this leads to considerable overhead in terms of synchronization and queue management.

B. EDT-Based Traffic Shaping

To eliminate the overhead of inter-traffic-class synchronization, state-of-the-art traffic shaping schemes are based on EDT (Earliest Departure Time) model [26], [27] (e.g., FQ [28], Carousel [19], and Eiffel [15]). This model is built upon two key ideas.

(1) The stipulation of shaping policies is decoupled from the execution of traffic throttling, as shown in Fig. 1. The stipulation of shaping policies can be at arbitrary position (e.g., transport layer and network layer), and is reflected by

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

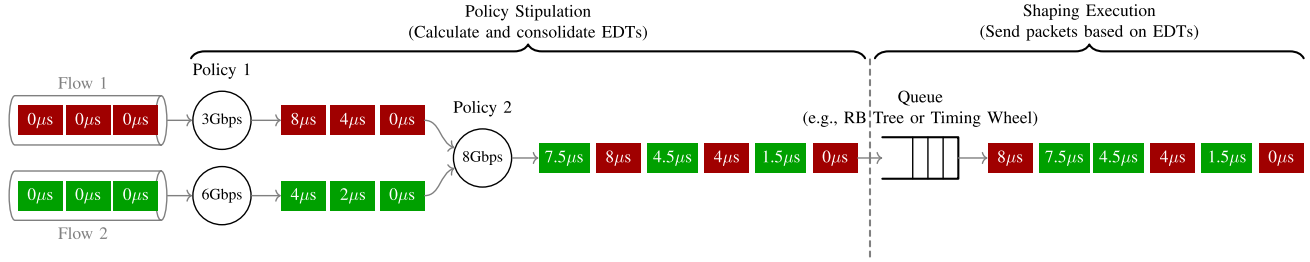


Fig. 1. EDT-based traffic shaping.

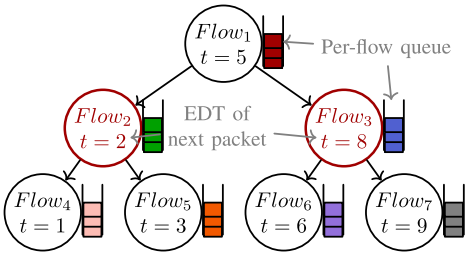


Fig. 2. Red-black tree in FQ.

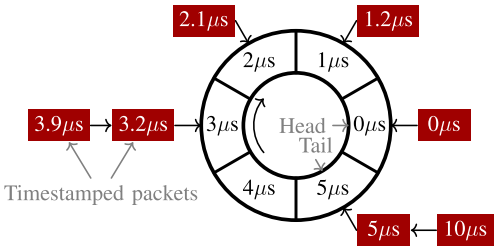


Fig. 3. A Timing Wheel with $1\mu s$ granularity and $5\mu s$ horizon.

timestamping each packet with the earliest departure time (EDT) allowed by this 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. For example, in Fig. 1, Flow 1 and Flow 2 are initially shaped at 3Gbps and 6Gbps, respectively. With 1500-byte packets, this translates to inter-packet intervals of $4\mu s$ for Flow 1 and $2\mu s$ for Flow 2. As packets traverse multiple shaping policies, each retains the largest EDT, representing the earliest permissible transmission time that satisfies all policies. For example, in Fig. 1, both flows then encounter a second policy that shapes them together at 8Gbps, requiring a minimum inter-packet interval of $1.5\mu s$. After traversing the policy, the packets of Flow 1 keep their original EDTs because they are later than the allowed transmission times calculated by the 8Gbps policy. In contrast, Flow 2 packets must update their EDTs, as their original EDTs are earlier than the allowed transmission times calculated by the 8Gbps policy.

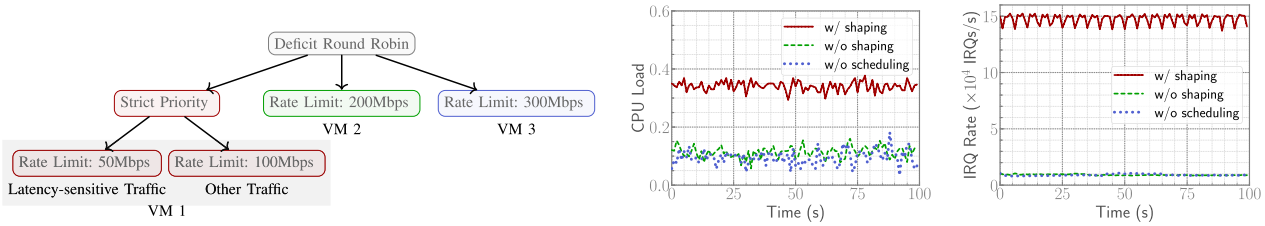
(2) The traffic shaper leverages a centralized queue to organize all arriving packets based on their EDTs. A packet is dequeued only when its EDT is earlier than the current time. Since packets typically do not arrive in EDT order, a simple FIFO queue is insufficient. Consequently, specialized queue structures are employed to effectively manage packets based on their EDTs. There are two notable queue structures.

①Red-black Tree: The Linux queueing discipline (`qdisc`) FQ utilizes a Red-black tree to organize packets based on EDTs, as shown in Fig. 2. Incoming packets are first placed in per-flow queues. The Red-black tree maintains the EDT of the first packet in each per-flow queue. When transmitting a packet, the Red-black tree is searched for the smallest departure time. Although effective, this approach still organizes packets with multiple queues thereby suffering from synchronization overhead.

②Timing Wheel: Carousel [19] and Eiffel [15] address the limitations of FQ by employing a Timing Wheel structure [29], which organizes packets in a single queue, thereby eliminating the need for per-flow structures. As a result, this design truly liberates the traffic shaper from maintaining massive queues and makes the shaper scalable to a large number of flows. As shown in Fig. 3, a Timing Wheel is a time-indexed circular queue that divides a time period into multiple slots Each slot holds packets whose EDTs fall within that time interval.

In Timing Wheel, two parameters need to be carefully configured as they determine the performance of traffic shaping. ① Granularity, which is the interval between adjacent time slots. Granularity should be fine enough to eliminate traffic bursts. For example, to limit the length of traffic bursts within one MTU, then the granularity should be no coarser than MTU/C, where C is the NIC speed. ② Horizon, which is the time span covered by the Timing Wheel. Horizon determines how far into the future packets can be queued. Horizon should be long enough to evenly space packets awaiting future transmission. For example, the Linux kernel limits the queued data to 128KB [30]. If the minimum supported shaping rate is $R_{\min}$, the horizon should be at least $128\text{KB}/R_{\min}$. Given a granularity g and a horizon h, the Timing Wheel contains $N = h/g$ slots. Enqueuing a packet with an EDT of t_{EDT} involves placing it in the slot indexed by $t_{EDT}/g \mod N$. For corner cases where the EDT is earlier or later than the valid time range, the packet is placed in the head or tail slot, respectively.

The enqueue operations for Timing Wheel are quite fast, requiring only $O(1)$ time to find the slot. Dequeue operations, on the other hand, vary between Carousel [19] and Eiffel [15]: Carousel dequeues all packets from the head slot when the current time exceeds the slot's time However, this approach may be inefficient when packets are sparsely distributed across slots, where the majority of the dequeue operations is meaningless. Eiffel enhances the dequeue efficiency of Carousel by leveraging the Find First Set (FFS) operation, which can quickly identify the first non-empty slot. Modern CPUs can perform this operation in approximately 3–5 cycles, making it significantly faster than Carousel's method.



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

Fig. 4. Overhead of different packet scheduling policies.

III. Motivation

In this section, we use a series of experimental observations to motivate FlowBundler.

A. 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 the `tc` tool, as shown in Fig. 4(a). The topmost policy enforces fair bandwidth sharing among VMs with a Deficit Round Robin (DRR) scheduler. VM 1 uses a Strict Priority (SP) 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 [19] (details of testbed are in §VI-A).

Fig. 4(b) shows the CPU overhead with different scheduling policies. The experiment results reveal that traffic shaping accounts for 92.5% of the CPU overhead in the packet scheduling system. Specifically, without any scheduling policies (blue line, labeled "w/o scheduling"), the CPU load is 9.52%, serving as the baseline that reflects the overhead of TCP/IP stack.[2] When non-work-conserving scheduling policies are applied (green line, labeled "w/o shaping"), the CPU load increases slightly to 11.36%. However, when traffic shaping policies are introduced (red line, labeled "w/ shaping"), the CPU load increases dramatically to 34.05%. This significant increase can be attributed to two main factors.

(1) Traffic shaping introduces a significant number of interrupts. Unlike work-conserving packet schedulers (e.g., DRR, SP), 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. 4(c)).

(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 [31]. To achieve 40Gbps rate for 1500B packets, a packet needs to be sent every 300ns, yet a single PCIe write operation can take up to ∼900ns [31], [32], [33]. Modern NICs and their drivers introduce a number of optimizations to mitigate this bottleneck, such as batched enqueueing of TX descriptors, pre-fetching descriptors, and interrupt moderation. However, all these low-level optimizations work on the basis of batched transmission, while traffic shaping inherently requires sending packets one at a time to prevent traffic bursts, which nullifies these optimizations. As a result, the network stack's achievable bandwidth is dramatically decreased [31].

Observation 1: A significant portion of a traffic shaper's overhead stems from extrinsic operations such as interrupts and PCIe communications. Optimizing these operations presents a substantial opportunity to minimize 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.

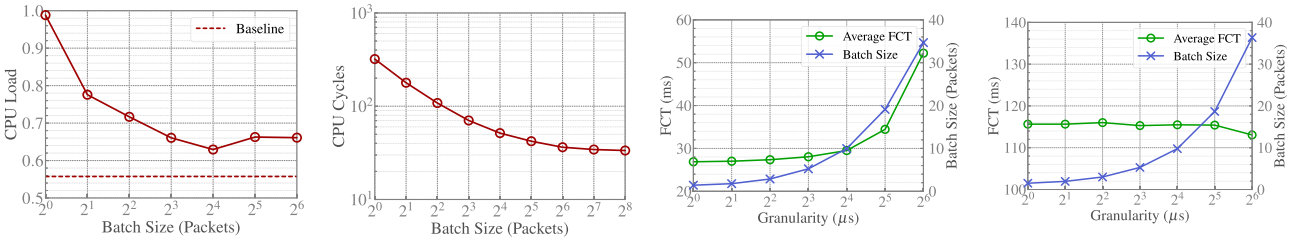
We conduct two experiments to quantitatively illustrate the effectiveness of batching. In the first one, we use TBF to throttle the traffic rate to 8Gbps and measure the CPU load when sending a flow with different batch sizes. To exclude the CPU load other than traffic shaping, we also measure a baseline of CPU load by setting up a FIFO packet scheduler at the sender and throttling traffic outside the host (i.e., at a switch). As shown in Fig. 5(a), batching 16 packets can reduce the CPU load of traffic shaping by $\sim 5.9\times$.[3] This is mainly due to the reduction of software interrupts. Specifically, the number of software interrupts is reduced by $\sim 7\times$ by batching 16 packets.

In the second experiment, we show that batching can still significantly reduce the CPU overhead without software interrupts. Fig. 5(b) shows the average CPU cycles of transmitting a packet with different batch sizes in Intel DPDK [34], which relies on a poll mode driver to send and receive packets. Batching 256 packets can reduce the CPU overhead by $9.5\times$. 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 [7], [13] and even overwhelm buffers [19], degrading the transmission performance. We conduct a

[2]For fair comparison, we throttle the overall traffic rate to 650 Mbps at the switch for scenarios without shaping nodes.

[3]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.



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

Fig. 5. Effect of batch size.

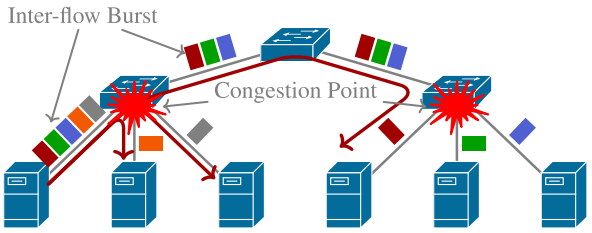


Fig. 6. Inter-flow burst can be naturally demultiplexed before congestion point.

large-scale ns-3 [35] 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 §VI-F). 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 [36]. The rate of each flow is limited to 5Gbps by Carousel. Fig. 5(c) shows the average flow completion time (FCT) and batch size with different shaping granularities. Batching 34 packets can extend the FCT by $\sim 2\times$.

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

To achieve efficient shaping without harming the network, an elaborate batching scheme is required. Our observation is that 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 bursts* are to blame for harming the network.[4] In contrast, *inter-flow bursts*, which consist of packets from different flows destined for different destinations, *offer a great opportunity to be naturally demultiplexed before queue buildup*. This is due to the unique traffic characteristics in the data center networks, as shown in Fig. 6.

(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 [37], [38]. 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 [21], [22] 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. 5(d) 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.

The way to create inter-flow burst is inter-flow batching, i.e., batching packets from different flows, which 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 less harmful to the network performance. A traffic shaping scheme should mainly 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

[4]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. To distinguish between a flow and a TCP/UDP flow, 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.

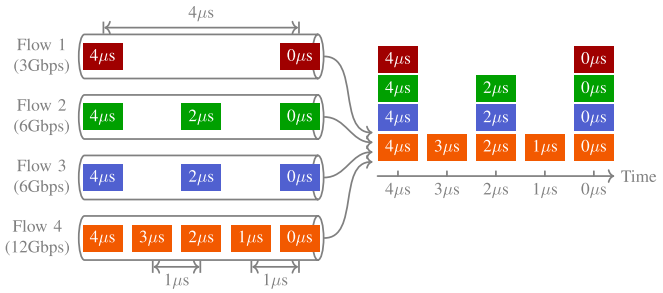


Fig. 7. An example of inter-flow batching.

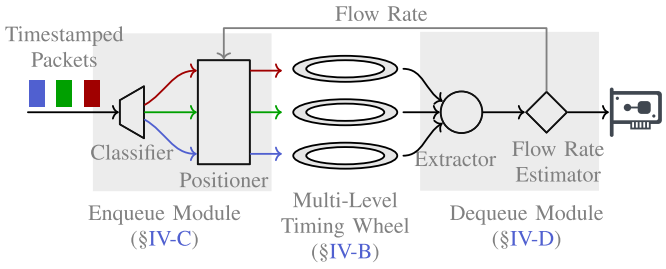


Fig. 8. FlowBundler overview.

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

FlowBundler aims to achieve the following three goals.

1) **Batching inter-flow packets:** When delivering packets to NIC, FlowBundler should batch the transmissions of packets belonging to different flows to reduce per-packet CPU overhead.
2) **Eliminating intra-flow burst:** When sending packets, 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.
4) **Performance safeguard:** When inter-flow bursts unexpectedly lead to congestion, FlowBundler should also eliminate inter-flow bursts to prevent degradation of transmission performance.

In the next section, we first explain the basic idea to achieve these goals. Then we describe each part of FlowBundler in detail.

A. Design Rationale

To achieve inter-flow batching while eliminating intra-flow burst (i.e., Goal 1 and Goal 2), 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. 7, four flows are shaped by 3Gbps, 6Gbps, 6Gbps, and 12Gbps, respectively. To eliminate intra-flow bursts for all flows (assuming that all packets are MTU-sized[5]), packet transmissions should be scheduled at a rate of 12Gbps (i.e., every 1500B/12Gbps $= 1\mu$s), 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, four inter-flow-batched packets are dequeued at 0μs and 4μs, and three are dequeued at 2μs. Formally, if the shaping rate of flow f is R_f and the maximum shaping rate is $R_{\max} = \max\{R_1, R_2, \ldots, R_N\}$, then packet transmissions are scheduled at an interval of $MTU/R_{\max}$, where MTU is the maximum transmission unit. Flow f send a packet every $R_{\max}/R_f$ scheduled transmissions (assuming that all packets are MTU-sized).

To achieve O(1) enqueue and dequeue time (i.e., Goal 3), 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 (more details in Supplementary Material A).

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 overhead. 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).

Finally, to guarantee transmission performance when inter-flow bursts lead to congestion (i.e., Goal 4), FlowBundler continuously monitors the congestion state of each flow. It then binds all congested flows together, treating them as a single flow. These flows are shaped collectively to eliminate the burst caused by them.

Putting them together, we design FlowBundler with the structure depicted in Fig. 8. As Carousel and Eiffel, FlowBundler is based on the EDT model. Specifically, it assumes that the EDT of each packet has already been determined upon its arrival. FlowBundler can be divided into three parts: an enqueue module, an MLTW queue, and a dequeue module. The enqueue module classifies incoming packets by their destinations and places them into the appropriate positions within the MLTW queue. The dequeue module retrieves packets from the MLTW at a rate corresponding to the maximum shaping rate across all flows. Then it estimates the shaping rate for each flow and delivers the packets to the Network Interface Card (NIC).

B. Multi-Level Timing Wheel

The arrangement of packets in the queue significantly impacts both the CPU and memory overhead of FlowBundler.

[5]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.

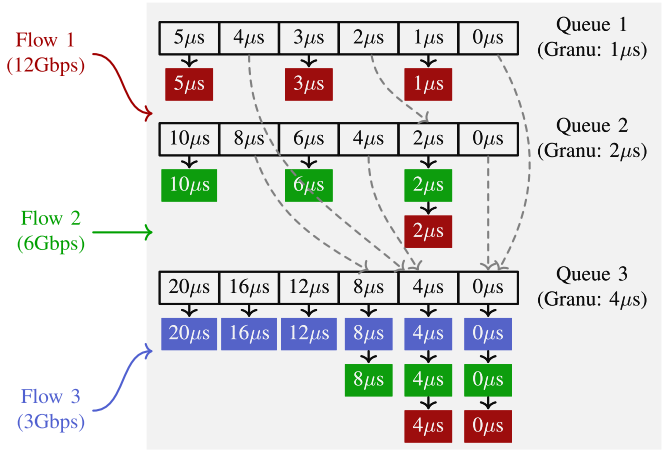


Fig. 9. An example of Multi-Level Timing Wheel with 1μs minimum granularity and 3 Timing Wheels.

The queue must organize packets based on the shaping rates of their corresponding flows. However, shaping rates vary both temporally and spatially. Temporally, a flow's shaping rate can change over time, as congestion control algorithms dynamically adjust sending rates based on observed congestion levels. The queue structure must support efficient enqueue and dequeue operations, regardless of these fluctuations. Spatially, shaping rates can differ widely among flows. For instance, one congested flow might need to send packets at 1Mbps, while another uncongested flow might operate at 10Gbps. The queue structure must accommodate the diversity with an acceptable memory overhead.

To address these challenges, we introduce a novel queue structure called the Multi-Level Timing Wheel (MLTW),[6] which enables fast enqueue and dequeue operations, even with varying flow shaping rates, and achieves this with a minimal memory overhead.

Basic structure: As shown in Fig. 9, MLTW consists of multiple Timing Wheels with exponentially growing granularities. For example, if the minimum granularity is set to 1ns — small enough to support a maximum shaping rate of 12Tbps — 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, the coarsest granularity would be 1500B/10Kbps = 1.2s, requiring 32 queues (2^{31}ns $>$ 1.2s). The length of each queue is determined by the maximum resident traffic volume per flow. For instance, in Linux, TSQ [30] typically limits in-stack data per connection to 128KB. Thus, each queue requires 128 slots. Multiple packets can share a slot if their combined size is below 1500B. In total, MLTW requires only 4096 slots to support 1ns granularity over a time range of [0, 256s], which is far less than Timing Wheel (as analyzed in Supplementary Material A).

Packet placement: To eliminate intra-flow bursts and accommodate packets across a wide time range, packets are placed in the queue with a granularity that best matches the flow's shaping rate. Specifically, for a flow with a shaping rate of R_f, packets are placed in the $(\lfloor \log_2 MTU/R_f/g_{\min} \rfloor + 1)$-th queue, where $g_{\min}$ is the minimum granularity. Note that this logarithmic calculation can be efficiently implemented with bit operations (§IV-C). For example, in Fig. 9, Flow 1, with a shaping rate of 12Gbps, requires packets to be sent every 1μs and is assigned to Queue 1, which has 1μs granularity. Similarly, packets of Flow 2 (6Gbps shaping rate) and Flow 3 (3Gbps shaping rate) are placed in Queue 2 (2μs granularity) and Queue 3 (4μs granularity), respectively. In this way, the intra-flow burst are constrained to a maximum of 2 MTUs.[7]

Furthermore, to enhance dequeue efficiency, MLTW consolidates inter-flow-batched packets into the same slot. This is achieved by aggregating packets that share the same slot time, even if they are located in different queues. For example, in Fig. 9, Queue 1-3 all contain a 4μs slot. Executing separate dequeue operations for each queue at the same time is inefficient. Instead, MLTW consolidates all packets in these overlapping slots into the slot of the coarsest-granularity queue (i.e., the queue with the largest index). Note that this approach does not introduce shaping errors because the transmission time of the aggregated slot remains unchanged. For example, in Fig. 9, 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

The enqueue module of FlowBundler processes each incoming packet by classifying it into its respective group and positioning it in the appropriate slot within the MLTW queue.

Classifying packets: FlowBundler needs to quickly classify packets into flows based on their destination host. In this paper, we assume that a destination host can be identified by its IP address. Nevertheless, this approach can also handle scenarios where a host has multiple IP addresses by using explicit mappings between IP addresses and host IDs.

A straightforward method for classification is hashing. However, simple hashing introduces non-negligible overhead, as a server may need to communicate with thousands of destinations [19], which requires either a large hash table or frequent handling of hash collisions.

To address this, FlowBundler leverages two key observations: (1) The hash table does not need to be excessively large. While larger tables can differentiate more flows and batch more inter-flow packets, the marginal benefit diminishes as the number of batched packets increases. For example, experiments show that batching more than 128 packets yields negligible benefits (Fig. 5(b)). Therefore, FlowBundler only needs to classify packets into hundreds of categories, making a hash table with 512 buckets sufficient.

(2) Resolving hash collisions is unnecessary. Hash collisions are common when the number of flows is comparable to the number of entries in the hash table. Given so many

[6]MLTW shares some similarities with the Hierarchical Timing Wheel structure [29], which is used by operating system to hold a wide range of timers. Hierarchical Timing Wheel is not CPU efficient enough for fast traffic shaping, since it needs to frequently trigger interrupts at its minimum granularity and move timers between queues. MLTW differs from it on packet placement and the corresponding enqueue and dequeue operations.

[7]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 8 Gbps flow sends a packet every 1.5μ s. It will be put into the 1μ s-spaced Timing Wheel and two packets will be put into the same slot every other slot (assuming that all packets are MTU-sized).

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
```

flows, the size of the batched packets is typically substantial. A hash collision does not bring any negative effects,[8] except that two flows are treated as one, reducing the number of batched packets by just one. This minor reduction has no significant impact on CPU overhead, as adding a single packet to an already sizable batch does not appreciably decrease processing costs. Consequently, the effect of hash collisions can be considered negligible.

Positioning packets: Once classified, packets are placed into the appropriate slots in the MLTW. Algorithm 1 shows the pseudocode of the process, which involves three key steps:

(1) Identifying the best-match queue: The queue granularity that best matches the flow's shaping rate is identified (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-matching queue ID is calculated as $\lfloor \log_2(MTU/R_f/g_{\min}) \rfloor$, where $g_{\min}$ is the minimum granularity (i.e., the granularity of Queue 1). Since logarithmic calculations can be computationally intensive, they are replaced with the equivalent FLS (Find Last Set) operation, $\text{FLS}(MTU/g_{\min}) - \text{FLS}(R_f)$, 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 [15].

(2) Handling extraordinary EDT values: Occasionally, the EDT of the packet can exceed the range of the current queue. For example, when the shaping rate of a flow 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, the packet is moved to a queue with coarser granularity that accommodates its EDT (Line 11-15).

(3) Batching inter-flow packets: To optimize dequeue efficiency, FlowBundler places packets into the coarsest-granularity queue containing a slot with the same slot time as the current queue (Lines 16-21). Specifically, let qid and $slotid$ represent the current queue and slot indices, respectively. The $(qid+1)$-th queue contains a slot with the same slot time if and only if $slotid$ is even. Thus, the packet is moved to the $(qid+1)$-th queue, at $slotid/2$-th slot. This process is repeated recursively until $slotid$ becomes odd. To streamline this step, we can leverage the characteristic that the number of trailing zeros in the binary representation of $slotid$ determines how many times to increment qid. Thus, the target queue ID can be given by $qid + \text{FFS}(slotid)$, where FFS (Find First Set) is another fast operation supported by modern CPUs [15].

D. Dequeue Module

The dequeue module in FlowBundler is responsible for extracting packets ready for transmission and dynamically estimating the shaping rate for each flow.

Extracting packets: The dequeue module extracts packets whose Earliest Departure Times (EDTs) are less than or equal to the current time. The process, outlined in Algorithm 2, operates at the frequency corresponding to the maximum shaping rate among active flows. The procedure involves the following steps:

(1) Determining the queue with the maximum shaping rate. FlowBundler identifies the queue corresponding to the smallest non-empty index, as this queue represents the highest shaping rate among active flows (Line 3).

(2) Iterating through slots. FlowBundler iterates through each slot of the selected queue to check for packets (Lines 5–16). For each iteration, FlowBundler identifies the actual slot containing packets (Lines 6–8). If packets are found, they are extracted for transmission (Line 12).

Estimating flow shaping rate: When a flow is shaped by a single policy, its shaping rate can be directly retrieved.

[8]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 3 Estimate Flow Shaping Rate

```
function EstimateRate(packets, slotid)
    for all pkt ∈ packets do
        flow ← Classify(pkt)
        flow.min_edt ← Min(flow.min_edt, pkt.edt)
        flow.max_edt ← Max(flow.max_edt, pkt.edt)
        if pkt.is_app_limited then ▷ Prevent under-estimation
            flow.min_edt ← now
            flow.max_edt ← now
            flow.sent_bytes ← 0
        else if flow.sent_bytes > 5 · MTU and
        flow.prev_slot ≠ slotid then
            T_r ← flow.max_edt − flow.min_edt
            rate ← flow.sent_bytes/T_r
            flow.rate ← (1 − α) · flow.rate + α · rate
            flow.min_edt ← pkt.edt
            flow.max_edt ← pkt.edt
            flow.sent_bytes ← pkt.length
        else
            flow.sent_bytes ← flow.sent_bytes+pkt.length
        flow.prev_slot ← slotid
```

However, in more complex scenarios, determining the overall shaping rate becomes challenging. For instance, a flow may be shaped by multiple policies, or multiple connections to the same destination may be aggregated into a single flow.

In these cases, FlowBundler dynamically estimates the overall shaping rate using Algorithm 3, which is based on the traffic departure rate. The process works as follows. FlowBundler monitors the amount of outgoing traffic (denoted by S) over a period of T_r. The instantaneous traffic rate can be given by S/T_r. To smooth out transient fluctuations in the traffic rate, FlowBundler updates the shaping rate using an exponentially weighted moving (low-pass filter):

$$R_f \leftarrow (1-\alpha)\cdot R_f + \alpha \cdot \frac{S}{T_r} \tag{1}$$

where α is the filter's time constant. This filter helps eliminate short-term oscillations, providing a more stable shaping rate.

Furthermore, to ensure timely and accurate rate estimation, T_r is dynamically adjusted based on the flow's shaping rate. Specifically, T_r is set to the duration of 5 MTU-sized packets. In this way, T_r shortens for flows with a low shaping rate and lengthens for those with a high shaping rate. This dynamic adjustment improves responsiveness to varying traffic conditions while preserving estimation accuracy.

E. Performance Safeguard

To maintain consistent transmission performance under unexpected scenarios, FlowBundler incorporates a safeguard mechanism to address cases where inter-flow bursts fail to demultiplex before reaching the congestion point, leading to unexpected queue buildup. This mechanism involves continuous congestion monitoring and inter-flow burst elimination for affected flows.

Monitoring congestion: FlowBundler employs two methods to monitor congestion. ① In the network stack, congestion states can be directly derived from TCP states (i.e., `ca_state`). In this case, FlowBundler monitors the TCP state of each packet upon arrival to detect congestion. ② When TCP states are unavailable, congestion status is inferred from ACK packets using Explicit Congestion Notification (ECN) markings or by analyzing round-trip times (RTT). The ECN states can be directly retrieved from the ECN-Echo (ECE) bit in the TCP header, whereas the RTT can be calculated using TCP's timestamp option.

Eliminating inter-flow burst: When congestion is detected, FlowBundler groups all congested flows into a single classification. In this way, FlowBundler will treat inter-flow bursts within the group as intra-flow bursts, which FlowBundler is designed to eliminate. By doing so, FlowBundler effectively mitigates further queue buildup and maintains stable performance even in congested conditions.

Overhead analysis: For TCP state monitoring, FlowBundler leverages the existing `ca_state` field maintained by the kernel, requiring only a single memory access per packet. ECN-based monitoring is similarly lightweight, examining just the ECE bit in TCP headers during regular packet processing. While RTT calculation using TCP timestamps involves arithmetic operations, the computational cost remains negligible. To group congested flows, FlowBundler can simply add a single bit flag to each bucket in the hash table, indicating whether the corresponding flow should be grouped. With a typical 512-bucket hash table, this requires only 64 bytes of additional memory. The CPU cost for checking and updating these flags is also minimal. Under heavy congestion, the CPU overhead actually remains constant per-packet, as the monitoring and classifying operations do not scale with congestion level. In summary, the congestion-aware safeguard introduces minimal CPU and memory overhead.

F. Integrating FlowBundler With Other Scheduling Policies

FlowBundler is specifically designed for rate limiting and does not natively implement other scheduling algorithms, such as weighted fair queueing or strict priority. However, when various scheduling policies coexist, FlowBundler can be seamlessly integrated into existing scheduling frameworks using established techniques [15], [33], [39]. For instance, Loom [33] demonstrates how rate limiting can be decoupled from work-conserving scheduling policies by employing an independent module that transmits packets based solely on their transmission times. FlowBundler's MLTW structure efficiently implements such a shaping queue, enabling precise rate limiting while allowing other work-conserving scheduling policies to function independently within the same system.

V. Implementation

We implemented FlowBundler in both Linux kernel and a userspace software NIC based on DPDK. The Linux kernel paces packets by software interrupts, while DPDK operates in a polling manner. Both can benefit from inter-flow batching.

A. Kernel Implementation

FlowBundler is implemented as a new Linux queueing discipline (`qdisc`) kernel module, which does not touch any existing network stack implementations. After inserting the kernel module, FlowBundler can be enabled using the Linux `tc` command, and no server reboot is required, allowing for easy deployment.

The `qdisc` resides in the traffic control layer, which sits between the TCP/IP stack and the NIC driver. It assumes that the EDT for each packet has been set by the upper layers. In our experiments, we set the flow shaping rate using the `SO_MAX_PACING_RATE` socket option, which enables TCP to calculate and write the EDT for each packet into the packet metadata (i.e., `skb_mstamp_ns` in the `skb` structure).

Each `qdisc` consists of an `enqueue` and a `dequeue` hook function. The `enqueue` function is triggered whenever a packet arrives. Since packets often contain more than one MTU-sized segment due to GSO/TSO (Generic Segmentation Offload/TCP Segmentation Offload), and GSO/TSO cannot be turned off via external APIs since Linux 4.17 [40], FlowBundler segments these GSO packets into MTU-sized packets and enqueues all of them, which eliminates intra-flow bursts.

The `dequeue` function extracts packets from MLTW and sends them to the NIC driver for transmission. When no packets are immediately ready for transmission, the `dequeue` function schedules the next packet using `qdisc_watchdog_schedule_ns()`. This function leverages Linux's internal timer infrastructure (i.e., `hrtimer`) to schedule a future transmission event. When the timer expires, the kernel's timer subsystem triggers a software interrupt that invokes the `dequeue` function again.

B. Userspace Implementation

We also implement FlowBundler in userspace with BESS [23] (previous known as SoftNIC [17]), a software NIC based on Intel DPDK [34]. BESS operates in a busy-polling mode. During each polling loop, five key steps are performed:

(1) Generating packets. To avoid overflow in the MLTW queue, a backpressure mechanism is implemented to limit the number of queued packets. Initially, a number of packets for each connection is generated and enqueued. Then this mechanism utilizes a dynamically maintained list of connection IDs to track connections which have packets dequeued. Only connections present in this list are permitted to produce additional packets, ensuring controlled packet generation. Specifically, when a packet is dequeued, its corresponding connection ID is added to the list, indicating that the connection is eligible to generate new packets. Once a packet is generated for a connection, its ID is removed from the list, requiring the connection to wait until the next packet departure to rejoin the list.

(2) Time-stamping packets. Packets are timestamped based on the flow shaping rate. The timestamp of each packet is calculated as $t = \max(now, prev\_t + L/R)$, where L is the packet length, R is the shaping rate of the connection, and $prev\_t$ is the timestamp of the previous packet.

(3) Enqueueing packets (including classifying packets). Packets are classified and enqueued according to Algorithm 1.

(4) Dequeueing packets (including estimating flow rate). Packets are dequeued and sent according to Algorithm 2. Packets within the same slot are dequeued together and transmitted as a burst. Additionally, we find that the BESS module responsible for sending packets to the NIC (`PortOut`) does not guarantee the successful delivery of all packets, dropping them if the ring buffer is full. At 100Gbps, this results in substantial packet loss. We modified the module to implement a retry mechanism, ensuring reliable delivery of all packets.

(5) Updating connection list for generating packets. For each dequeued packet, the connection ID is appended to the connection list, ensuring that the connection can generate packets in the next loop.

VI. Evaluation

We use testbed experiments and event-driven simulations to evaluate the performance of FlowBundler. The result highlights include:

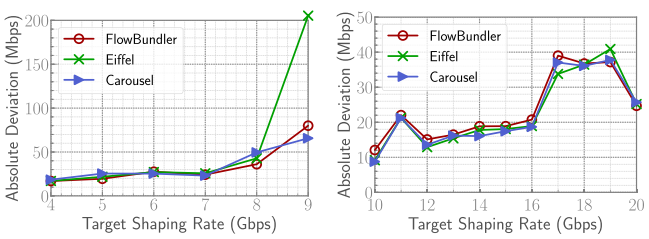
- **CPU efficiency:** FlowBundler can support a shaping rate of 98Gbps, which is 2.6× and 2.7× better than Carousel and Eiffel, respectively.
- **Memory efficiency:** FlowBundler only requires 1.1MB memory to accommodate flows with wide disparities, which is three orders of magnitude less than both Carousel and Eiffel.
- **Scalability:** FlowBundler can scale to 100K concurrent connections.
- **Transmission performance:** FlowBundler can batch packet transmissions without harming the network performance.

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.

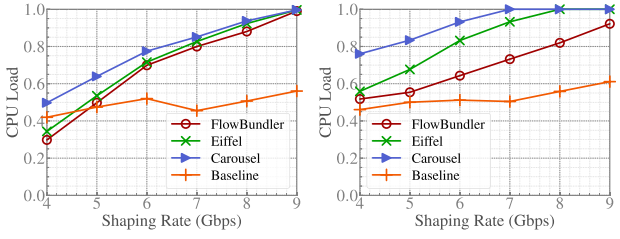
The 100Gbps testbed contains two servers. Each server is equipped with an NVIDIA Mellanox ConnectX-6 Dx dual-port 100GbE NIC, a 6-core Intel i5-10400 2.90GHz CPU, and 16GB memory, We evaluate the userspace performance on the 100Gbps testbed. As our testbed only contains one physical sender and one physical receiver, we vary the destination IP addresses to emulate multiple (virtual) destinations.

Compared schemes: We compare FlowBundler to Carousel [19] and Eiffel [15]. For kernel experiments, we use the open source code provided by authors [41], [42]. Their implementations are based on Linux 4.10, and they patched the Linux kernel source code to support EDT. Specifically, they modified the `sk_buff` structure to record the transmission time calculated by TCP. As the EDT feature has been officially supported by vanilla Linux kernel from Linux 4.20, we modified the corresponding code so that Carousel and Eiffel can be enabled by simply inserting a kernel module without patching the kernel source code. For userspace experiments, we implement them on BESS following the papers and their kernel implementations. In particular, for Eiffel, we implement the cFFS-based version because it achieves higher performance than the approximate-queue-based version [15].



(a) Kernel (b) Userspace

Fig. 10. Rate conformance.



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

Fig. 11. [Kernel] CPU load and frequency of software interrupts with different numbers of flows.

Parameter settings: FlowBundler employs 64 Timing Wheels, each containing 1,024 slots.[9] The minimum granularity is 1 nanosecond. This fine granularity is used only to demonstrate memory efficiency across a wide range of time granularities. In practice, a granularity of 100ns is sufficient to support a shaping rate of 100Gbps. For the rate estimation mechanism, we set $\alpha = 0.25$, with the initial estimated flow rate initialized to 20Gbps. For Carousel and Eiffel, the granularity is set to 1μs in a 10Gbps network and 100ns in a 100Gbps network. These configurations are sufficient to eliminate intra-flow bursts in these systems. Additionally, the horizons for both Carousel and Eiffel are set to 10 seconds.

Traffic generation: For Linux kernel performance, we generate traffic with `iPerf3` and `neper` [43], which can set the shaping rate for each connection with the `SO_MAX_PACING_RATE` socket option. For userspace performance, we implement a custom traffic generator based on the `FlowGen` module provided by BESS.

B. Rate Conformance

In this experiment, we evaluate whether FlowBundler can accurately shape traffic based on the given shaping rate. We measure the absolute deviation from the target rate, calculated as $|\text{observed\_rate} - \text{target\_rate}|$.

Kernel performance: we established a single TCP connection and monitored its throughput every 10ms on the server-emulated switch. Throughput was calculated using `libpcap`, which counts the number of received bytes every 10ms at the network interface connected to the sender. As shown in Fig. 10(a), FlowBundler achieves a mean absolute deviation comparable to Eiffel and Carousel, despite utilizing Timing Wheels with coarser granularity than the latter two.[10] Specifically, the deviation is always within 0.9%.

Userspace performance: We monitor the throughput every 1ms at the receiver using a customized packet-receiving program based on DPDK. Other settings keep unchanged. Fig. 10(b) illustrates that FlowBundler achieves a mean absolute deviation within 0.3%, which is on par with Carousel and Eiffel.

[9]These values are more than what is mentioned previously. We enlarge the number of slots as in BESS experiments a sender can generate 1 MB 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.

[10]Recall that FlowBundler selects a Timing Wheel whose granularity best matches the shaping rate, while the granularity of Carousel and Eiffel is always 1μ s.

C. 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 non-negligible. 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 that may introduce extra TCP processing overhead at the sender.

Fig. 11 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, with 16 flows, FlowBundler achieves ∼12-28% and ∼7-20% lower CPU load than Carousel and Eiffel, respectively. This improvement stems from FlowBundler's ability to send packets in batches, thereby reducing interrupts, context switches, and PCIe operations.

However, these measurements include the overhead of the Linux kernel stack processing, which dominates the total CPU consumption. To better understand FlowBundler's efficiency gains, we isolate the traffic shaping overhead by subtracting the baseline CPU load (kernel stack processing without traffic shaping). When we exclude this kernel overhead, FlowBundler demonstrates a more significant CPU load reduction of ∼35-56% compared to Eiffel and ∼43-66% compared to Carousel.

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.

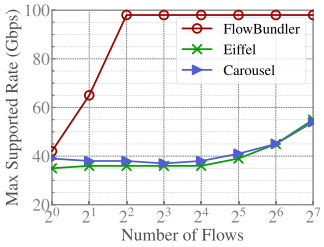


Fig. 12. [Userspace] Max supported shaping rate.

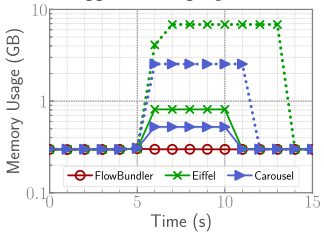


Fig. 13. [Kernel] Memory efficiency of FlowBundler.

Fig. 12 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.

D. 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`.

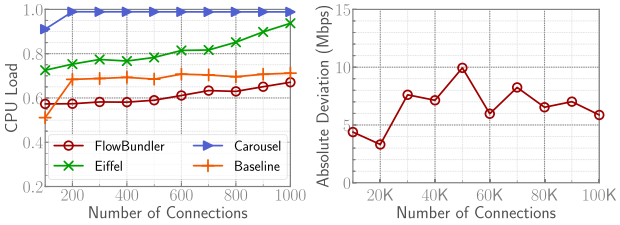
As shown in Fig. 13, FlowBundler consumes little extra memory. It only requires ∼1.1MB memory to achieve a very fine granularity of 1ns and a broad horizon of 1.8×10^{10} seconds. In comparison, Eiffel and Carousel consume a considerable amount of memory. With a granularity of 1μs (solid line), Eiffel 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.

E. Scalability

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

Kernel performance: Following [15], 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. 14(a) 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



(a) Kernel (b) Userspace

Fig. 14. Scalability.

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. 14(b) shows the absolute deviation of FlowBundler with different numbers of connections. FlowBundler can consistently achieve high shaping accuracy with 10K - 100K connections.

F. Large-Scale Simulations

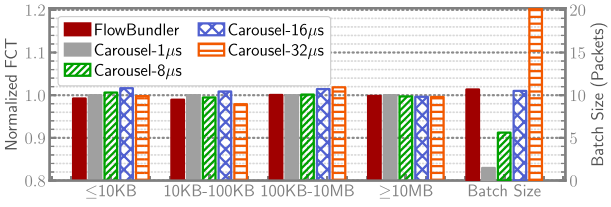
In this part, we use ns-3 [35] simulations to demonstrate that FlowBundler can batch packet transmissions without degrading the network performance. The simulations are conducted on a Huawei Taishan 200 server, equipped with two Kunpeng 920 CPUs and 192GB of memory.

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 [36] as the transport protocol, and the ECN threshold is set to 17KB, as suggested in [44].

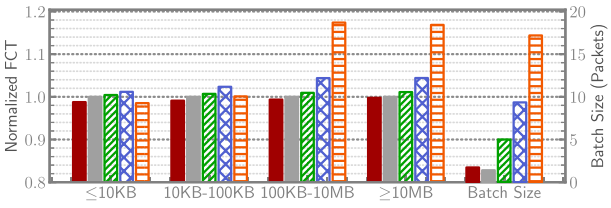
Workloads: There are two kinds of traffic patterns. (i) One-to-one traffic: a sender establishes a connection to a receiver. (ii) Fanout traffic: a sender establishes 16 concurrent connections to different receivers. Half servers are generating the former kind of traffic. The others are generating the 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 [36]. 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. 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.

Simulation results: Fig. 15 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. 15(a)) and one-to-one flows (Fig. 15(b)). Meanwhile, FlowBundler can batch ∼ 7.3× more packets than Carousel with 1μs granularity for fanout flows. In

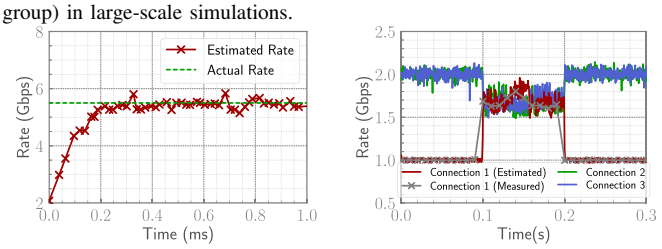


(a) Fanout flows



(b) One-to-one flows

Fig. 15. FCT across different flow sizes (first 4 groups) and batch size (last group) in large-scale simulations.



(a) Scenario 1 (b) Scenario 2

Fig. 16. Accuracy of rate estimator.

comparison, Carousel with coarser granularity (i.e., 8μs, 16μs, and 32μs) batches more packets at the expense of degrading FCT performance.

G. FlowBundler Deep Dive

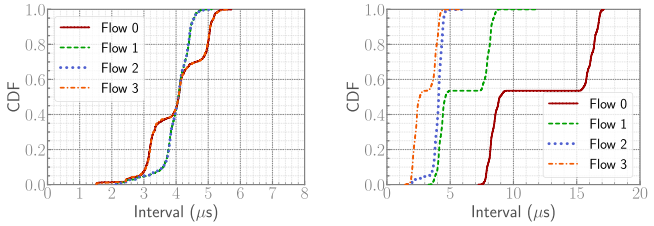
We use a series of targeted experiments to examine the effectiveness of FlowBundler's internal mechanisms.

Accuracy of the shaping rate estimator:

In this experiment, we examine whether our rate estimator can accurately and quickly determine flow shaping rates under two scenarios.

In Scenario 1, ten connections are individually shaped at rates ranging from 0.1Gbps to 1.0Gbps, with an overall flow shaping rate of 5.5Gbps. These connections are classified into the same flow. The initial rate of the rate estimator is set to 2Gbps. We record the estimated shaping rate whenever it changed. Fig. 16(a) shows that FlowBundler accurately estimates the shaping rate within 200μs.

In Scenario 2, three connections are subjected to two shaping policies sequentially: First, each connection is shaped individually. Connection 1's rate varied as follows: 1Gbps for the first 100ms, 6Gbps for the next 100ms, and 1Gbps for the final 100ms. Connections 2 and 3 are shaped at 2Gbps throughout. Second, all three connections are collectively shaped at 5Gbps. These connections are classified into three different flows. The estimated shaping rate is monitored every 60μs at the sender, and the actual rate is measured every 10ms at the receiver. Fig. 16(b) illustrates that the estimated rate closely matched the measured rate. While minor jitter ($\sim$300Mbps) is observed in the estimated rate, it does not



(a) Scenario 1 (b) Scenario 2

Fig. 17. Distribution of interval between alternate packets.

impact enqueue decisions due to the $2\times$ granularity difference between adjacent queues. In other words, rate variations within 50% still ensure proper packet placement into queues.

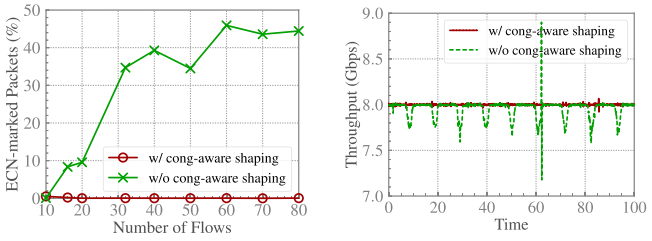
Burst elimination under the microscope: To assess FlowBundler's ability to eliminate intra-flow bursts, we connect hosts to a Barefoot Tofino switch, where each passing packet is timestamped with the time it arrives. The packet arrival intervals are collected and analyzed at the receiver. The analysis focuses on whether intra-flow bursts remained within 2 MTUs by examining the intervals between the i-th and $(i+2)$-th packets.

We consider two scenarios. In Scenario 1, four flows are individually shaped at 6Gbps. The CDF of packet intervals (Fig. 17(a)) shows that FlowBundler evenly spaces alternate packets around 4μs, confirming that intra-flow bursts do not exceed 2 MTUs.

In Scenario 2, four flows are shaped at 2Gbps, 4Gbps, 6Gbps, and 8Gbps, respectively. The CDF of packet intervals is illustrated in Fig. 17(b). For the 6Gbps flow (Flow 2), intervals are evenly spaced. For other flows, the intervals are distributed around two values. For example, intervals of the 2Gbps flow (Flow 0) cluster around 8μs and 16μs. This is because packets of Flow 0 are enqueued to the Timing Wheel with 8.192μs granularity, while the expected packet interval is 6μs. As a result, some packets will be put into the same slot, resulting in occasional back-to-back packets in the network. Nevertheless, the burst length remained within 2 MTUs, as alternate packet intervals are always over 7μs. These results demonstrate that FlowBundler effectively mitigates intra-flow bursts, ensuring packet spacing within the desired limits.

Effectiveness of congestion-aware shaping: To evaluate FlowBundler's ability to safeguard performance under congestion caused by inter-flow bursts, we conduct an experiment using the kernel implementation. The setup involves three servers connected to a Huawei CE6865 switch via Intel 82599 10Gbps NICs. The network employs DCTCP as the congestion control algorithm, with the ECN marking threshold set to 10KB. We generate multiple concurrent flows from different senders to a single receiver, ensuring that inter-flow bursts caused queue buildup at the switch. On each host, the overall shaping rate is set to 4Gbps.

Fig. 18(a) shows the fraction of packets marked with ECN, with different numbers of concurrent flows. With congestion-aware shaping, FlowBundler can effectively eliminate ECN marking. Additionally, Fig. 18(b) shows the overall throughput at the receiver with 20 flows per host. The results confirm that FlowBundler maintains 100% throughput with congestion-aware shaping, highlighting its capability



(a) Fraction of ECN marking (b) Throughput

Fig. 18. Effectiveness of performance safeguard mechanism in case of congestion.

to ensure performance stability when inter-flow bursts do not demultiplex. The results demonstrate that FlowBundler's congestion-aware shaping effectively mitigates congestion.

VII. Related Work

We classify the literature on efficient traffic shaping into two categories: software approaches and hardware approaches.

Software-based traffic shaping: In Linux, TBF (Token Bucket Filter) [24] and HTB (Hierarchy Token Bucket) [25] use the token bucket algorithm to limit the traffic rate. FQ (Fair Queuing) [28] schedules packets based on EDT. They have poor scalability due to synchronization overhead across multiple queues. hClock [16] is a hierarchical bandwidth allocation scheme that supports rate limit, minimum bandwidth guarantee, and bandwidth share simultaneously. As the above approaches, it puts packets into per-traffic-class queue and thus also incurs high overhead with a large number of traffic classes. Carousel [19] and Eiffel [15] use a single queue to shape all flows. FlowBundler further improves their CPU efficiency by inter-flow batching and their memory efficiency by the Multi-Level Timing Wheel structure. FairPolicer [45], BC-PQP [46], and ScalaTap [47] are designed to provide efficient overall rate control using the token bucket algorithm. However, they can introduce bursts at finer granularities, potentially impacting performance at the microscopic level.

Hardware-based traffic shaping: Several approaches [48], [49], [50], [51] are proposed to deal with a limited number of rate limiters in hardware. vShaper [48] maps vast traffic classes to available shapers on the data plane. SENIC [49] employs software-hardware co-design to support 10s of thousands of rate limiters. SwRL [50], Nimble [51], and Wang et al. [52] focus on scalable rate limiting schemes on programmable switches. Recently, a number of hardware primitives [39], [53], [54], [55] are proposed to support programmable packet scheduling, including traffic shaping. PIFO [39], PIEO [53], and PIPO [55] can achieve traffic shaping by specifying when a packet is eligible for scheduling. Programmable Calendar Queues [54] works similarly to Carousel when used for shaping. Tassel [56] proposes a hierarchical rate-limiting approach for RDMA NICs. It employs flow-level rate limiting to achieve scalability while leveraging packet-level rate limiting to ensure precision and high performance. These approaches need dedicated hardware, while software approaches can be deployed anywhere.

Batching for high-speed packet processing: Recent work has explored batching techniques to improve the efficiency of high-speed software packet processing. Batchy [57] and Reframer [58] propose to re-batch packets for fast packet processing. Notably, Reframer batches intra-flow packets, which contrasts with FlowBundler's inter-flow batching approach.

Nonetheless, FlowBundler is orthogonal and complementary to these systems. The key distinction lies in where and why batching occurs: FlowBundler operates at the transmission stage (when packets leave the host) to improve packet transmission efficiency, while Batchy and Reframer operate at the reception or processing stages (when packets arrive at or traverse the host) to improve packet processing efficiency. This fundamental difference in objectives means that these approaches can coexist — Batchy and Reframer optimize the packet processing pipeline, while FlowBundler optimizes the packet transmission without obscuring network congestion signals.

VIII. 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. FlowBundler uses a Multi-Level Timing Wheel to queue inter-flow batched packets together, while dynamically adjusting its dequeue frequency based on the flow shaping rate. Experiments show that FlowBundler achieves much higher CPU and memory efficiency than state-of-the-art approaches.

References

[1] D. Shan et al., "Burst can be harmless: Achieving line-rate software traffic shaping by inter-flow batching," in *Proc. IEEE Conf. Comput. Commun.*, May 2023, pp. 1–10.

[2] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg, "EyeQ: Practical network performance isolation at the edge," in *Proc. USENIX NSDI*, 2013, pp. 297–312.

[3] A. Kumar et al., "BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing," in *Proc. ACM SIGCOMM*, 2015, pp. 1–14.

[4] K. Jang, J. Sherry, H. Ballani, and T. Moncaster, "Silo: Predictable message latency in the cloud," in *Proc. ACM Conf. Special Interest Group Data Commun.*, Aug. 2015, pp. 435–448.

[5] M. Dalton et al., "Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization," in *Proc. USENIX NSDI*, 2018, pp. 373–387.

[6] P. Kumar et al., "PicNIC: Predictable virtualized NIC," in *Proc. ACM SIGCOMM*, 2019, pp. 351–366.

[7] 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 *Proc. USENIX NSDI*, 2012, pp. 1–19.

[8] Y. Zhu et al., "Congestion control for large-scale RDMA deployments," in *Proc. ACM Conf. Special Interest Group Data Commun.*, Aug. 2015, pp. 523–536.

[9] R. Mittal et al., "TIMELY: RTT-based congestion control for the datacenter," in *Proc. ACM Conf. Special Interest Group Data Commun.*, Aug. 2015, pp. 537–550.

[10] A. Kalia, M. Kaminsky, and D. G. Andersen, "Datacenter RPCs can be general and fast," in *Proc. USENIX NSDI*, 2019, pp. 1–16.

[11] W. Cheng, K. Qian, W. Jiang, T. Zhang, and F. Ren, "Re-architecting congestion management in lossless Ethernet," in *Proc. USENIX NSDI*, 2020, pp. 19–36.

[12] G. Kumar et al., "Swift: Delay is simple and effective for congestion control in the datacenter," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Architectures, Protocols Comput. Commun.*, Jul. 2020, pp. 514–528.

[13] D. Shan and F. Ren, "Improving ECN marking scheme with micro-burst traffic in data center networks," in *Proc. IEEE Conf. Comput. Commun.*, May 2017, pp. 1–9.

[14] N. Beheshti, Y. Ganjali, M. Ghobadi, N. McKeown, and G. Salmon, "Experimental study of router buffer sizing," in *Proc. 8th ACM SIGCOMM Conf. Internet Meas.*, Oct. 2008, pp. 197–210.

[15] A. Saeed et al., "Eiffel: Efficient and flexible software packet scheduling," in *Proc. USENIX NSDI*, 2019, pp. 17–32.

[16] J.-P. Billaud and A. Gulati, "HClock: Hierarchical QoS for packet scheduling in a hypervisor," in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, Apr. 2013, pp. 309–322.

[17] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "SoftNIC: A software NIC to augment hardware," Dept. Elect. Eng. Comput. Sci., Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2015-155, 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html

[18] M. Shahbaz et al., "PISCES: A programmable, protocol-independent software switch," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 525–538.

[19] A. Saeed, N. Dukkipati, V. Valancius, V. The Lam, C. Contavalli, and A. Vahdat, "Carousel: Scalable traffic shaping at end hosts," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 404–417.

[20] M. Marty et al., "Snap: A microkernel approach to host networking," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, Oct. 2019, pp. 399–413.

[21] A. Singh et al., "Jupiter Rising: A decade of clos topologies and centralized control in Google's datacenter network," in *Proc. ACM SIGCOMM*, 2015, pp. 183–197.

[22] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy, "High-resolution measurement of data center microbursts," in *Proc. Internet Meas. Conf.*, Nov. 2017, pp. 78–85.

[23] *BESS: Berkeley Extensible Software Switch*. Accessed: Jan. 17, 2025. [Online]. Available: http://span.cs.berkeley.edu/bess.html

[24] *TC-TBF*. Accessed: Jan. 17, 2025. [Online]. Available: https://man7.org/linux/man-pages/man8/tc-tbf.8.html

[25] *TC-HTB*. Accessed: Jan. 17, 2025. [Online]. Available: https://man7.org/linux/man-pages/man8/tc-htb.8.html

[26] *TCP: Switch to Early Departure Time Model*. Accessed: Jan. 17, 2025. [Online]. Available: https://lwn.net/Articles/766564/

[27] S. Fomichev, E. Dumazet, W. de Bruijn, V. Dumitrescu, B. Sommerfeld, and P. Oskolkov, "Replacing HTB with EDT and BPF," in *Proc. Netdev 0x14*, Jul./Aug. 2020 [Online]. Available: https://netdevconf.info//0x14/pub/papers/55/0x14-paper55-talk-paper.pdf

[28] *PKT_Sched: FQ: Fair Queue Packet Scheduler*. Accessed: Jan. 17, 2025. [Online]. Available: https://lwn.net/Articles/565421/

[29] G. Varghese and T. Lauck, "Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility," in *Proc. 11th ACM Symp. Operating Syst. Princ.*, 1987, pp. 25–38.

[30] *TCP Small Queues*. Accessed: Jan. 17, 2025. [Online]. Available: https://lwn.net/Articles/507065/

[31] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, "Understanding PCIe performance for end host networking," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2018, pp. 327–341.

[32] M. Flajslik and M. Rosenblum, "Network interface design for low latency request-response protocols," in *Proc. USENIX ATC*, 2013, pp. 333–346.

[33] B. Stephens, A. Akella, and M. M. Swift, "Loom: Flexible and efficient NIC packet scheduling," in *Proc. USENIX NSDI*, 2019, pp. 33–46.

[34] *Intel DPDK*. Accessed: Jan. 17, 2025. [Online]. Available: https://www.dpdk.org/

[35] *NS-3*. Accessed: Jan. 17, 2025. [Online]. Available: https://www.nsnam.org/

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

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

[38] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (Datacenter) network," in *Proc. ACM Conf. Special Interest Group Data Commun.*, Aug. 2015, pp. 123–137.

[39] A. Sivaraman et al., "Programmable packet scheduling at line rate," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 44–57.

[40] *TCP: Switch to GSO Being Always on*. Accessed: Jan. 17, 2025. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=0a6b2a1dc2a2105f178255fe495eb914b09cb37a

[41] *Kernel Implementation of Carousel*. Accessed: Jan. 17, 2025. [Online]. Available: https://github.com/saeed/eiffel_linux/tree/v4.10-gq

[42] *Kernel Implementation of Eiffel*. Accessed: Jan. 17, 2025. [Online]. Available: https://github.com/saeed/eiffel_linux/tree/working_ffs-based_qdisc

[43] *Neper: A Linux Networking Performance Tool*. Accessed: Jan. 17, 2025. [Online]. Available: https://github.com/google/neper

[44] M. Alizadeh, A. Javanmard, and B. Prabhakar, "Analysis of DCTCP: Stability, convergence, and fairness," in *Proc. ACM SIGMETRICS joint Int. Conf. Meas. Model. Comput. Syst.*, Jun. 2011, pp. 73–84.

[45] D. Shan, P. Zhang, W. Jiang, H. Li, and F. Ren, "Towards the fairness of traffic policer," in *Proc. IEEE Conf. Comput. Commun.*, May 2021, pp. 1–10.

[46] A. Tahir, P. Goyal, I. Marinos, M. Evans, and R. Mittal, "Efficient policy-rich rate enforcement with phantom queues," in *Proc. ACM SIGCOMM Conf.*, Aug. 2024, pp. 1000–1013.

[47] Z. Chen et al., "ScalaTap: Scalable outbound rate limiting in public cloud," in *Proc. IEEE Conf. Comput. Commun.*, May 2025, pp. 1–10.

[48] G. Kumar, S. Kandula, P. Bodik, and I. Menache, "Virtualizing traffic shapers for practical resource allocation," in *Proc. USENIX HotCloud*, 2013, pp. 1–6.

[49] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat, "SENIC: Scalable NIC for end-host rate limiting," in *Proc. USENIX NSDI*, 2014, pp. 475–488.

[50] Y. He, W. Wu, X. Wen, H. Li, and Y. Yang, "Scalable on-switch rate limiters for the cloud," in *Proc. IEEE Conf. Comput. Commun.*, May 2021, pp. 1–10.

[51] V. S. Thapeta, K. Shinde, M. Malekpourshahraki, D. Grassi, B. Vamanan, and B. E. Stephens, "Nimble: Scalable TCP-friendly programmable in-network rate-limiting," in *Proc. ACM SIGCOMM Symp. SDN Res. (SOSR)*, Oct. 2021, pp. 27–40.

[52] S.-Y. Wang, H.-W. Hu, and Y.-B. Lin, "Design and implementation of TCP-friendly meters in P4 switches," *IEEE/ACM Trans. Netw.*, vol. 28, no. 4, pp. 1885–1898, Aug. 2020.

[53] V. Shrivastav, "Fast, scalable, and programmable packet scheduler in hardware," in *Proc. ACM Special Interest Group Data Commun.*, Aug. 2019, pp. 367–379.

[54] N. K. Sharma et al., "Programmable calendar queues for high-speed packet scheduling," in *Proc. USENIX NSDI*, 2020, pp. 685–699.

[55] C. Zhang et al., "PIPO: Efficient programmable scheduling for time sensitive networking," in *Proc. IEEE 29th Int. Conf. Netw. Protocols (ICNP)*, Nov. 2021, pp. 1–11.

[56] Z. Wang et al., "Fast, scalable, and accurate rate limiter for RDMA NICs," in *Proc. ACM SIGCOMM Conf.*, Aug. 2024, pp. 568–580.

[57] T. Lévai, F. Németh, B. Raghavan, and G. Rétvári, "Batchy: Batch-scheduling data flow graphs with service-level objectives," in *Proc. USENIX NSDI*, 2020, pp. 633–649.

[58] H. Ghasemirahni et al., "Packet order matters! Improving application performance by deliberately delaying packets," in *Proc. USENIX NSDI*, 2022, pp. 807–827.