Fast, Scalable, and Accurate Rate Limiter for RDMA NICs

Zilong Wang[1] Xinchen Wan[1] Luyang Li[2] Yijun Sun[1] Peng Xie[3]
Xin Wei[3] Qingsong Ning[3] Junxue Zhang[1] Kai Chen[1]
[1] *iSING Lab, Hong Kong University of Science and Technology*
[2] *ICT/CAS* [3] *ByteDance*

ABSTRACT

RDMA NICs desire a rate limiter that is accurate, scalable, and fast: to precisely enforce the policies such as congestion control and traffic isolation, to support a large number of flows, and to sustain high packet rates. Prior works such as SENIC and PIEO can achieve accuracy and scalability, but they are not fast enough, thus fail to fulfill the performance requirement of RNICs, due primarily to their *monolithic* design and *one-packet-per-sorting* transmission. We present Tassel, a *hierarchical* rate limiter for RDMA NICs that can deliver high packet rates by enabling *multiple-packet-per-sorting* transmission, while preserving accuracy and scalability. At its heart, Tassel renovates the workflow of the rate limiter hierarchically: by first applying scalable rate limiting to the flows to be scheduled, followed by accurate rate limiting to the packets to be transmitted, while leveraging adaptive batching and packet filtering to improve the performance of these two steps. We integrate Tassel into the RNIC architecture by replacing the original QP scheduler module and implement the prototype of Tassel using FPGA. Experimental results show that Tassel delivers 125 Mpps packet rate, outperforming SENIC and PIEO by 3.6×, while supporting 16 K flows with low resource usage, 7.5% - 25.6% as compared to SENIC and PIEO, and preserving high accuracy, precisely enforcing rate limits from 100 Kbps to 100 Gbps.

CCS CONCEPTS

• **Networks** → **Data center networks**; **Network services**; • **Hardware** → **Networking hardware**.

KEYWORDS

Rate Limiter, RDMA NIC, Congestion Control

ACM Reference Format:
Zilong Wang, Xinchen Wan, Luyang Li, Yijun Sun, Peng Xie, Xin Wei, Qingsong Ning, Junxue Zhang, and Kai Chen. 2024. Fast, Scalable, and Accurate Rate Limiter for RDMA NICs. In *ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24), August 4–8, 2024, Sydney, NSW, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3651890.3672215

ACM SIGCOMM '24, August 4–8, 2024, Sydney, NSW, Australia

ACM ISBN 979-8-4007-0614-1/24/08
https://doi.org/10.1145/3651890.3672215

1 INTRODUCTION

RDMA NICs (RNICs) desire an accurate, scalable, and fast rate limiter. Specifically, RNICs expect the rate limiter to be accurate and scalable to precisely execute policies such as congestion control and traffic isolation, enforcing a specific rate on each flow and injecting inter-packet gaps for smoother traffic, across tens of thousands of flows at end hosts [30, 43, 55]. This helps evolving congestion controls to address the challenges of the bursty and unpredictable traffic in data centers [13, 14, 30, 32, 55], as well as the network traffic isolation among users to ensure stable application performance and user experience [22, 23, 26, 29]. In addition, RNICs require the rate limiter to sustain high packet rate to ensure its integration into the data path does not degrade the throughput of RNICs. Any slowdown in the rate limiter can degrade RNIC's performance.

Prior works SENIC and PIEO [41, 46] can achieve accuracy and scalability, but they are not fast enough, *e.g.* only achieving 31.4% of the desired packet rate for RNICs (§2.3). The root cause of their performance issue is that they follow a *monolithic* design, directly applying rate limiting to the flows to be scheduled, and consequently transmit only one packet after sorting all flows (*one-packet-per-sorting* transmission), while sorting a large number of flows is slow [46, 48, 52]. For instance, sorting 16 K flows only achieves 34.5 Mpps packet rate, in contrast to RNICs which can achieve 110 Mpps [3, 42].

In this paper, we present Tassel, a *hierarchical* rate limiter for RDMA NICs that can deliver high packet rate by enabling the transmission of multiple packets after sorting all flows (*multiple-packet-per-sorting* transmission). At its core, Tassel renovates the workflow of the rate limiter hierarchically: by first applying scalable rate limiting to the flows to be scheduled, followed by accurate rate limiting to the packets to be transmitted, while incorporating customized optimization strategies to improve the performance of these two steps (§3.2). Specifically, Tassel leverages adaptive batching (§3.2.1), which fetches multiple packets from scheduled flows to hide PCIe and sorting latency, to improve the packet rate for flow-level rate limiting. Meanwhile, it employs packet filtering (§3.2.2), which reduces the number of packets requiring sorting to increase the sorting rate, to enhance the performance of packet-level rate limiting. In this way, Tassel's hierarchical design can achieve high performance, scalability, and accuracy simultaneously.

We have integrated Tassel into RNIC architecture, by replacing the original QP scheduler and leveraging a combination of data structures tailored for Tassel's two-step sorting (§3.3), and implemented the prototype with FPGA (§4). Experiments show that Tassel achieves high packet rate while preserving high scalability and accuracy (§5). Specifically, Tassel can achieve 125 Mpps packet

rate when supporting 16 K flows, making it sufficient to transmit 100 B packets at 100 Gbps link rate, outperforming SENIC and PIEO by 3.6× (§5.1). With the increasing flow capacity, Tassel achieves a high and stable clock frequency, ensuring high performance. Meanwhile, Tassel demonstrates its high scalability with low resource usage. When supporting 16 K flows, the computing and memory resources consume less than 5% and 1% of our FPGA, respectively, 7.5% to 25.6% as compared to SENIC and PIEO (§5.2). In addition, Tassel preserves high accuracy as it precisely enforces the rate limits ranging from 100 Kbps to 100 Gbps for tens of thousands of concurrent flows, and proportionally shares bandwidth when the link is oversubscribed (§5.3).

This paper makes the following contributions:

- We experimentally analyze the performance issues of previous works and reveal that the root cause stems from their monolithic design and one-packet-per-sorting transmission (§2.3).
- We design Tassel, a hierarchical rate limiter for RNICs, which addresses the performance challenge while maintaining accuracy and scalability (§3.2). We integrate Tassel into RNIC architecture by replacing the original QP scheduler (§3.3).
- We implement the Tassel prototype with FPGA. Experiments demonstrate that Tassel achieves our design goal of performance, scalability, and accuracy (§4 and §5).

This work does not raise any ethical issues.

2 BACKGROUND AND MOTIVATION

In this section, we first motivate the need for rate limiter in RNICs and summarize its desired properties. Then, we discuss the limitations of existing works and analyze their root causes. Finally, we present our insight and basic idea.

2.1 The Need for Rate Limiter in RNICs

RNICs can use a rate limiter to 1) enforce a specific rate on each flow, and 2) inject inter-packet gaps to smooth traffic, a.k.a., pacing [43]. Specifically, RNICs use the rate limiter to execute the congestion control policies (§2.1.1). These policies precisely allocate bandwidth across massive flows and help mitigate network congestion in high-speed networks. Additionally, the rate limiter is used for the network traffic isolation as required by practical cloud management (§2.1.2).

2.1.1 Congestion Control

RNICs incorporate the congestion control mechanisms to manage complex environments and address challenges such as high link rate, low fabric latency, large infrastructure scale, and incast in high-speed networks [10, 11, 21, 27, 30], ensuring high-performance and stable network services [11, 32, 36, 55]. Within RNIC, the congestion control module computes the sending rate for potentially tens of thousands of flows, and then utilizes the rate limiter to execute these policies, limiting each flow's rate and pacing the traffic [14, 41].

Evolving congestion control algorithms. Congestion control has always been a critical area of research, with various developed algorithms relying on the support of rate limiters:

- Rate-based congestion control [35, 55] requires the rate limiter to enforce the calculated rate for each flow and pace the traffic accordingly.
- Window-based congestion control [13, 30, 32] needs the rate limiter to pace packets especially when setting the congestion window smaller than one to handle the large-scale incast [30, 33]. The sender translates the fractional congestion window to an inter-packet delay and uses it to pace packets into the network. For example, a cwnd of 0.5 results in sending a packet after a delay of $2 \times$ RTT. Production results show that pacing is crucial for maintaining low latency and loss at scale. When a large incast occurs, supporting a congestion window of less than 1 can achieve line-rate throughput with low latency and zero loss. However, disabling pacing leads to a 29% loss rate and 18× average RTT [30].
- Centralized congestion control [39], in contrast to the above decentralized methods, uses a centralized network arbiter to allocate bandwidth for individual senders according to the traffic pattern. Senders then utilize the rate limiter to enforce the allocated sending rate. Effective traffic planning and pacing help eliminate network bottlenecks, resulting in a fascinating congestion-free network.

Programmable congestion control framework. The rate limiter is also a necessary component of programmable congestion control (PCC) frameworks [8, 14, 38]. As there is no panacea for congestion control design, PCC is gaining popularity. Its attractiveness stems from the flexibility to modify and deploy continuously evolving algorithms on hardware, addressing various application scenarios and network characteristics. These PCC frameworks support diverse congestion control algorithms mentioned above, and require the rate limiter to execute the congestion control policies.

2.1.2 Network Traffic Isolation

The rate limiter can additionally be used to implement native hardware rate limiting for network traffic isolation. Practical cloud management requires isolating network traffic among users, VMs, and applications [22, 23, 26, 29], to ensure stable application performance and user experience [15, 26]. In practice, providers use predictable and scalable bandwidth arbitration systems [22, 26, 29] to assign rates to VM pairs, which are then enforced at end systems via the rate limiter. As implementing rate limiting in software can be CPU-intensive (consuming 30% CPU in some cases [31]) and contradicts latency-sensitive services' preference to bypass software hypervisor [41], using the rate limiter to implement native hardware rate limiting can be beneficial [49].

2.2 Desired Properties of Rate Limiter

RNICs desire a rate limiter to be accurate, scalable, and fast:

- *Accurate.* The rate limiter should precisely enforce the given rate on each flow[1] and inject inter-packet gaps to smooth the traffic, preventing bursts.
- *Scalable.* The rate limiter must support rate limiting for tens of thousands of flows at end hosts.

[1]We use flows, queues, and queue pairs (QPs) interchangeably in the paper.

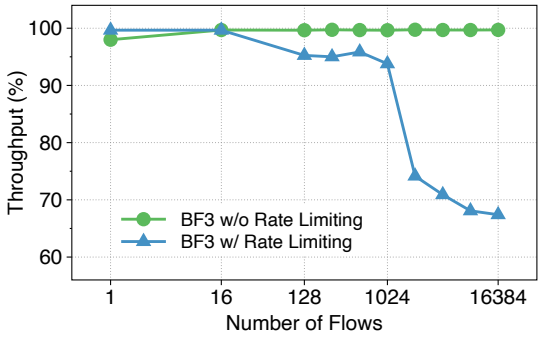


Figure 1: Performance issue with current RNIC's rate limiter. The aggregate throughput of the RNIC collapses when the rate limiting is enabled for more than 1K flows.

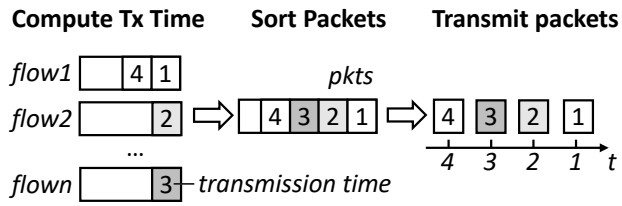
- *Fast.* Upon achieving high accuracy and scalability, the rate limiter should operate fast and achieve a high packet rate, ensuring its integration into RNIC's data path does not degrade its high performance.

Accuracy. Effective congestion control and traffic isolation desire a highly accurate rate limiter, which calculates the transmission time for each flow's packets and transmits them precisely at the calculated time. Enforced rate limiting often demands nanosecond-level precision [25, 39, 47], and inaccurate rate limiting can result in traffic bursts and worsen network congestion [32]. Furthermore, an accurate rate limiter should support a wide range of rate limits for each flow, *e.g.* 100 Kbps to the line rate, and allow for fine-grained adjustments instead of restricting to several fixed values [14]. For example, small rate limits like 100 Kbps[2] [30] are used in congestion control to handle large-scale incast, and large rate limits like 100 Gbps link speed are used when congestion control aims to saturate the network link.

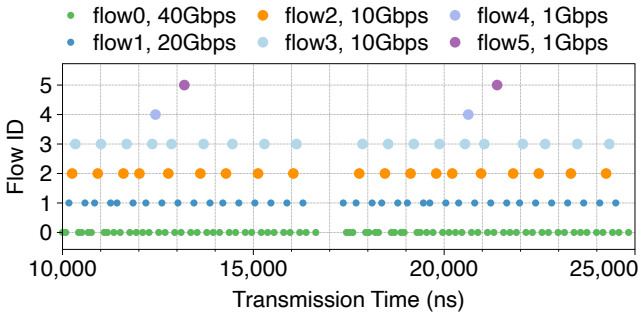
Scalability. RNICs need to apply rate limiting for tens of thousands of flows. As the memory and computing resources in hardware are restricted and scarce [51], RNICs desire a scalable rate limiter that can support rate limiting for such a large number of flows while meeting the hardware resource constraints.

High-performance (fast). RNICs are known for their high performance. For example, they can execute up to 110 Mpps using a single NIC port, achieving 100G bps with packets of 100 B [3, 42]. RNICs need to integrate the rate limiter into their data path to deploy it, as the rate limiter is responsible for scheduling and transmitting packets. Consequently, the performance of the rate limiter directly impacts the performance of RNIC. If the rate limiter is slow in processing and transmitting packets, it leads to a reduction in RNIC performance. Therefore, the rate limiter needs to be fast enough to achieve a high packet rate, *e.g.* capable of transmitting a packet every 8 ns. In scenarios with higher bandwidth, such as 400 or 800 Gbps, the rate limiter must have a sufficiently high packet rate to fully utilize the link capacity.

[2]We calculate the minimal rate based on the minimal window size in Swift. Using typical values, we calculate $min\_rate = min\_window\_size * MTU/RTT = 0.001 * 1024B/40us = 204.8Kbps$. Hence we choose 100 Kbps as the minimal rate and the adjustment granularity for the rate limiter.



(a) Workflow. The rate limiter 1) computes the transmission time for all packets, 2) sorts, and 3) transmits them on time.



(b) Ideal packet transmission scenario.

Figure 2: Illustration of accurate rate limiting.

In summary, RNICs desire an accurate, scalable, and fast rate limiter. However, existing solutions fail to achieve this objective, as will be elaborated next.

2.3 Existing Works & Limitations

2.3.1 Commercial RNICs

Commercial RNICs allow rate limiting for each queue pair (QP) to support congestion control. Users can configure a rate limit for each QP via RNICs' programmable congestion control (PCC) framework [8]. However, as shown in Figure 1, we observe performance degradation when enabling rate limiting for more than 1K flows in NVIDIA BlueField-3 [7], which integrates ConnectX-7 [4]. Specifically, we directly connect two BlueField-3 and use the PCC framework to configure the rate limit for each QP. Each QP is assigned the same rate limit, calculated by dividing the link bandwidth by the number of QPs. For example, with 1K QPs, each QP is limited to 0.2 Gbps, resulting in a total rate limit of 200 Gbps. We then run the standard *perftest* [5] benchmarks with default settings and measure the aggregate throughput via *mlnx_perf* [9]. As shown in Figure 1, the aggregate throughput of BlueField-3 drops 32.4% (from 199.3 to 134.8 Gbps) when the number of QPs increases from 1 to 16384. In comparison, BlueField-3 can saturate the link bandwidth when the rate limiting and PCC are disabled.

This performance anomaly indicates potential performance issues with BlueField-3's rate limiter[3]. We exclude the possibility of the PCC framework being the performance bottleneck, as the RNIC can achieve the line rate when 10K QPs are configured with a large rate limit, 100 Gbps.

[3]Since commercial RNICs are blackbox, we are not able to analyze their micro-architecture to identify the root cause of the performance issue. A possible reason could be the relatively low processing rate of the rate limiter and the related scheduler.

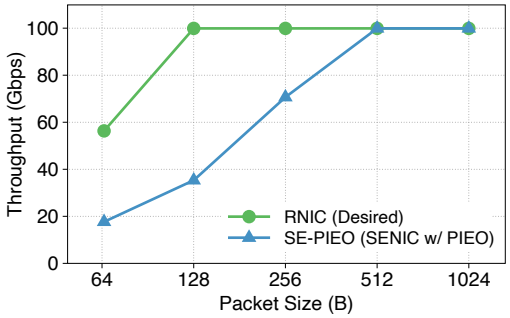


Figure 3: The throughput of SE-PIEO with 16K flow capacity, comparing to the desired performance in RNICs.

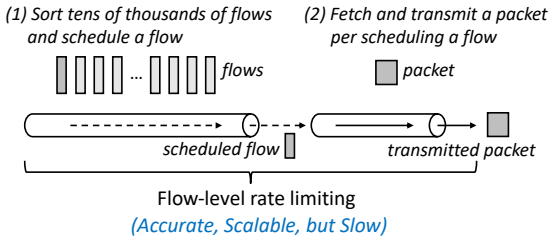


Figure 4: Throughput model of monolithic rate limiting. The rate limiter sorts tens of thousands of flows, schedules a flow, transmits the head packet, updates the transmission time for that flow, and then sorts again to send the next packet.

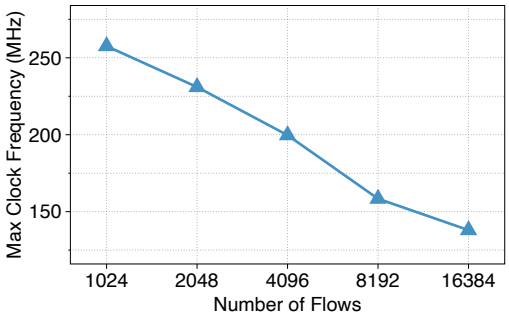


Figure 5: Maximum clock frequency achieved by SE-PIEO varying the flow number it supports.

2.3.2 Recent Studies

WF^{2}Q+ [18] is an accurate and classic time-based rate limiting algorithm, widely adopted in rate limiters to ensure accuracy [41, 46, 54]. As illustrated in Figure 2a, WF^{2}Q+ precisely calculates the transmission time of the packets among various flows, sorts them according to the calculated time, and transmits them on time. Figure 2b shows the simulated ideal packet transmission scenario using WF^{2}Q+. Two machines are connected directly, and the sender initiates several large flows, each configured with different rate limits. As shown in Figure 2b, WF^{2}Q+ enforces accurate rate limiting for each flow. Packets within each flow are sent at intervals determined by the rate limit, ensuring no bursts. In theory, WF^{2}Q+ can limit the per-flow traffic rate and ensure that the transmission delay for each packet is bounded. Thus, WF^{2}Q+ is widely considered accurate [18, 41, 46].

Recent studies, SENIC [41] and PIEO [46], implement scalable and accurate rate limiters for end-host NICs. They implement WF^{2}Q+ in hardware, thus ensuring accuracy. Additionally, they improve scalability by saving hardware memory and computing resources, thereby supporting rate limiting for tens of thousands of flows. Specifically, SENIC optimizes the memory usage as it stores just flow metadata in hardware, rather than the packets themselves. PIEO, on the other hand, saves the computing resources by designing a two-dimensional compare-and-shift architecture to implement the sorting in WF^{2}Q+. PIEO reduces the consumption of the computing resources, *i.e.* flip-flops and comparators, from $O(N)$ to $O(\sqrt{N})$, where N is the number of flows. Furthermore, a system that integrates both SENIC and PIEO can be more resource-efficient and scalable. It inherits SENIC's architecture design and replaces the packet scheduler module with PIEO. In the remaining part of this paper, we refer to this system as *SE-PIEO* (SENIC with PIEO).

Performance Issue. SE-PIEO can achieve high accuracy and scalability, however, it is not fast enough and fails to address the challenging performance requirement of RNICs. We have implemented a prototype of SE-PIEO and evaluated its throughput as well as the packet rate when transmitting 16 K flows. Figure 3 shows the throughput it achieved with the varying packet sizes, compared to the desired performance of RNICs referring to their high packet rate. Results show that SE-PIEO can only achieve as much as 31.4% and 35.4% throughput compared to RNIC under 64 B and 128 B packet size, respectively. Such throughput gap indicates the low packet rate achieved by SE-PIEO than RNIC, *i.e.*, 34.5 Mpps for supporting 16 K flows, which is 31.4% of RNIC's capability. As a result, SE-PIEO will become the critical performance bottleneck if integrated into RNIC, primarily due to its poor packet rate. The above issue can be further amplified when RNIC bandwidth reaches 400 or 800 Gbps. Specifically, given a higher MAC capacity, e.g., 400 Gbps, SE-PIEO can only achieve 282.6 Gbps with its 34.5 Mpps low packet rate under 1024 B packet size, insufficient to saturate the link bandwidth.

Analysis. Given the above performance results, we systematically analyze the workflow of SE-PIEO, and reveal that the root cause is it follows a *monolithic* design, directly applying rate limiting to the flows to be scheduled, and consequently transmits only one packet after sorting all flows, while sorting a large number of flows is slow. Specifically, SE-PIEO computes the transmission interval for packets within each flow, and sorts the flows based on the transmission time of each flow's head packet. Then it identifies the flow with the nearest transmission time, and sends its head packet on time. In such a manner, SE-PIEO realizes the rate limiting at flow level, but must sort all flows to send a packet. We call this manner as the *monolithic* flow-level rate limiting, as illustrated in Figure 4. Consequently, it transmits one packet per sorting, and the sorting performance determines the rate limiter's performance.

However, it is inherently challenging to achieve high efficiency when sorting a large number of elements in hardware. Several approaches, such as BMW Tree [52], PIEO [46], and PIFO [48] have been designed to improve the efficiency of sorting, but they have yet struggled to meet the high demands of RNICs. For instance, BMW Tree achieves as much as 50 Mpps, which is only 45.5% of the RNIC's capacity.

The primary challenge in sorting lies in the fact that the increasing number of flows to be sorted increases the complexity of combinational circuit, which then naturally decreases the maximum achievable clock frequency of the sorting circuit [46]. The sorting rate is determined by 1) the clock frequency of the sorting circuit, and 2) the number of cycles required for sorting. Thereinto, the number of cycles required for sorting is determined by the algorithm design and remains constant, *e.g.* 4 cycles to enqueue an element into an ordered list. As a result, decreased clock frequency degrades the sorting performance. As an evidence, we measure the maximum clock frequency achieved by SE-PIEO with varying flow capacity and show the results in Figure 5. When the flow capacity increases from 1 K to 16 K, the achieved clock frequency decreases from 258 MHz to 138 MHz, resulting in a 46.5% frequency reduction and an ultimate 138/4=34.5 Mpps packet rate, given that SE-PIEO consumes the constant 4 cycles for sorting and transmits a packet afterwards.

Consequently, SE-PIEO's monolithic flow-level rate limiting is accurate, scalable, but slow. It supports accurate rate limiting for tens of thousands of flows, but sorting a large number of flows degrades the packet transmission rate.

2.4 Insight and Idea

Instead of transmitting only the head packet, we seek to transmit multiple packets after sorting all flows to improve performance, while maintaining accuracy and scalability. Our key insight is that the workflow of monolithic rate limiting consists of two steps, flow-level scheduling and packet-level transmission, which can inspire a redesign into a hierarchical approach as follows.

Initially, we inherit the aforementioned flow-level rate limiting to sort flows and schedule the nearest flow. Once a flow is scheduled, we fetch multiple packets. Then, we additionally apply rate limiting to this batch of fetched packets to ensure accuracy, via sorting and transmitting them strictly based on their transmission time. This extra tier of rate limiting ensures the accuracy of packet transmission and injects inter-packet gaps for smoother traffic, and we call this packet-level rate limiting. To guarantee the accuracy and packet rate of this packet-level rate limiting, it is essential to ensure that the total number of packets, fetched from all flows and requiring rate limiting, is small.

Fortunately, we observe a chance to reduce this number from tens of thousands to just a few hundred. Specifically, despite each flow may fetch several packets, amounting to tens of thousands in total, RNIC only needs to store and apply rate limiting to a small number of *imminent* packets that are near to be sent, typically a few hundred (3.2.2). The rest of the packets (which are actually the packet descriptors) can be dropped and retrieved when their transmission times are near. This filtering mechanism reduces the packet number and thus relaxes the scalability requirement for packet-level rate limiting, enabling a fast and accurate packet-level rate limiter.

Figure 6 illustrates the hierarchical rate limiting, including scalable flow-level rate limiting and accurate and fast packet-level rate limiting. This hierarchical approach improves the performance limits by shifting from sorting tens of thousands of flows to sorting hundreds of imminent packets.

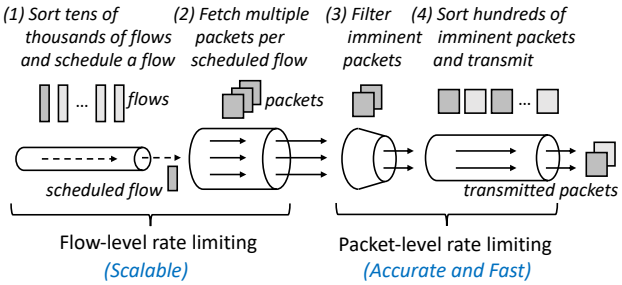


Figure 6: Throughput model of hierarchical rate limiting (Tassel). Apply scalable flow-level rate limiting, and then fetch multiple packets per scheduled flow. Simultaneously, pick imminent packets and apply accurate and fast packet-level rate limiting for them.

3 TASSEL DESIGN

Based on the above insight, we design Tassel, a fast, scalable, and accurate rate limiter for RDMA NICs. Tassel's design consists of the hierarchical rate limiting algorithm (§3.2) and the architecture design of the rate limiter in RNIC (§3.3).

3.1 Design Goal and Overview

The design goal of Tassel is to improve the performance of the rate limiter while maintaining accuracy and scalability. To achieve this goal, we follow three design guiding principles: (1) enable the transmission of multiple packets after sorting all flows to improve packet rate; (2) utilize the time-based rate limiting algorithm to compute the transmission time and order for all flows' packets, and transmit them accordingly to ensure accuracy; and (3) save memory and computing resources in hardware as much as possible, by avoiding unnecessary buffering or too complicated data structures, to improve scalability.

Guided by the above principles, Tassel designs a two-tier hierarchical rate limiting by first applying scalable flow-level rate limiting (§3.2.1), followed by accurate and fast packet-level rate limiting (§3.2.2). Such a hierarchical rate limiter can be easily integrated into the architecture of RNIC by replacing the original QP scheduler (§3.3.2) and leveraging a combination of data structures tailored to meet the diverse requirements at different stages in Tassel (§3.3.3).

3.2 Hierarchical Rate Limiting

As illustrated in Figure 7, Tassel's hierarchical design consists of flow-level rate limiting and packet-level rate limiting, and follows the following workflow:

(1) Sort flows. The flow scheduler first sorts tens of thousands of flows by their scheduling time, which is the transmission time of their head packet, respectively, and then identifies the latest flow.
(2) Monitor and schedule. The flow scheduler monitors current system time, and schedules the latest flow when the current time approaches its scheduling time.
(3) Fetch packets. The flow scheduler fetches multiple packets for the subsequent packet-level rate limiting.

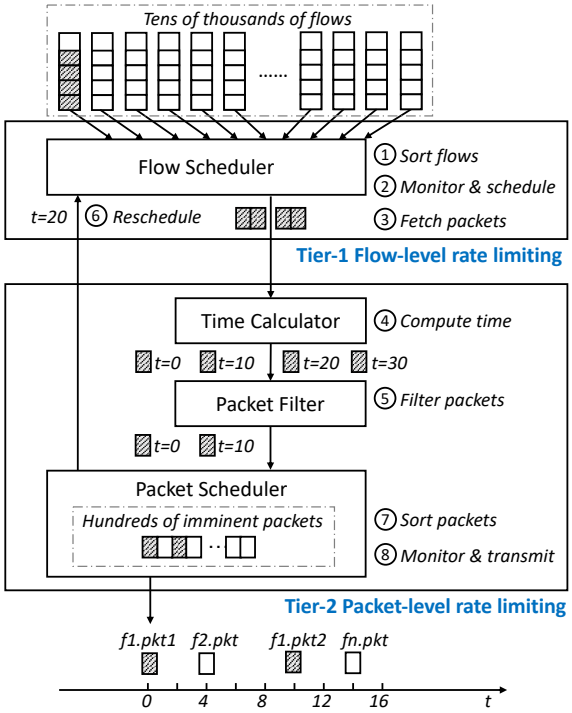


Figure 7: Tassel's hierarchical rate limiting design.

(4) Compute transmission time. Upon arriving the fetched packets, the time calculator computes the packets' transmission time based on the WF^{2}Q+ algorithm.
(5) Pick *imminent* packets. From fetched packets, the packet filter picks imminent packets whose transmission time is close to the current system time and are hence imminent to get transmitted (detailed in §3.2.2). The rest of the packets (which are actually the packet descriptors) are dropped and retrieved next time.
(6) Reschedule. The rest fetched packets dropped in the packet filter need to be retrieved. The packet scheduler identifies the packet with the nearest transmission time among dropped packets and notifies the flow scheduler to 1) set this packet as the new head packet for the corresponding flow; 2) use its transmission time to update the flow's scheduling time; and 3) reschedule the flow.
(7) Sort packets. The packet scheduler then takes over the transmission of imminent packets from all flows. It sorts all imminent packets based on the transmission time and identifies the nearest packet for transmission. The number of imminent packets is bounded within a few hundred, as detailed in §3.2.2.
(8) Monitor and transmit. The packet scheduler monitors the current system time and transmits the nearest packet on time when it is permitted to be sent. If multiple packets are eligible to be sent simultaneously, the packet scheduler prioritizes and transmits the packet which will finish transmitting first.

Consequently, packets across a large number of flows are precisely transmitted according to the calculated transmission time.

3.2.1 Scalable Flow-level Rate Limiting

Flow-level rate limiting supports scalable flow scheduling and employs an adaptive batching mechanism to hide the scheduling latency.

Flow scheduling (① & ②). Tassel maintains each flow's scheduling time and sorts the flows accordingly. Simultaneously, Tassel maintains a global timer that records the current system time. The timer is easier to implement in hardware compared to software, requiring just a little parallel logic to increment the counter each cycle instead of relying on CPU polling. Tassel continuously checks the timer to determine if it's time to schedule the closest flow. Upon reaching the scheduled time, Tassel schedules this flow and fetches multiple packets. The number of packets fetched is based on the adaptive batch mechanism, which is described in detail later.

Scheduling latency (⑥). Scheduling latency is the time consumed to execute one scheduling decision, including scheduling a flow and fetching packets from the host. This usually takes several PCIe transactions, while the PCIe round-trip latency between RNIC and host memory is high (around 1us in FPGA-based RNIC). This high scheduling latency affects the accuracy of rate limiting. To ensure transmitting packets at the right time, we should schedule the flow per scheduling latency in advance. Tassel achieves this by adding a typical scheduling latency to the timer's system time before comparing it with the flow's scheduling time.

Meanwhile, high scheduling latency degrades the packet rate achieved by rate limiting. Specifically, a flow must wait for its last scheduling decision to be fully executed before it can be rescheduled. For example, if trying to reschedule a flow before its previous scheduling is complete, the flow scheduler doesn't know how many packets were sent last time, which packet should be sent next, or when to reschedule this flow. So, high scheduling latency causes scheduling a single flow slowly, decreasing this flow's packet rate as well as the throughput. Using multiple concurrent flows can improve the aggregate packet rate but it's still blocked by the slow flow scheduling rate. To address this performance challenge in both single and multiple flow scenarios, we design the adaptive batching mechanism described below.

Adaptive batching (③). Adaptive batching fetches multiple packets from scheduled flow after sorting all flows, which can hide high scheduling latency and improve the packet rate of flow-level rate limiting.

The goal of adaptive batching is to keep the link as busy as possible based on the varying rate limits, avoiding the achieved throughput falls short of the given rate limit. If too few packets are fetched, the next-layer packet-level rate limiting transmits them quickly, causing no packets to be sent until this flow is rescheduled after the scheduling latency passes. As a result, the throughput of this flow remains low, potentially failing to reach the configured rate limit. Conversely, fetching an excessive number of packets at once can lead to a large transmission duration, potentially blocking the packet scheduler.

The ideal packet number to fetch is the number that this flow is allowed to send within the scheduling latency. This number, denoted as N, can be calculated based on the rate limit, the flow's typical packet size, and the scheduling latency as follows:

$$N = \frac{rate\_limit}{typical\_pkt\_size} * scheduling\_latency$$

For example, as shown in Figure 8, assuming the rate limit is 25 Gbps, the typical packet size is 1024 B, and the scheduling latency is 1 us,

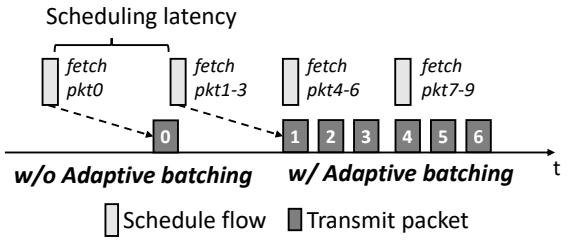


Figure 8: Adaptive batching mechanism. Adaptive batching hides the scheduling latency and improves flow's packet rate.

then N equals 3. This means that we should fetch 3 packets each time a flow is scheduled. Without adaptive batching, *i.e.* $N = 1$, this flow can only achieve 8.2 Gbps, significantly below the configured rate limit.

Adaptive batching improves the packet input rate for the packet scheduler module, enabling a fast and accurate packet-level rate limiting.

3.2.2 Fast & Accurate Packet-level Rate Limiting

Packet-level rate limiting supports fast and accurate packet transmission and consists of the following parts:

Time computation (④). Once the fetched packets arrive, we calculate the transmission time for each packet following the WF^{2}Q+ algorithm. The transmission time includes the start (S) and finish (F) time for each packet given the flow rate limit (R). A packet is considered eligible (*eligibility evaluation*) when its start time is less than the current system time (T). After the calculation, the algorithm transmits those eligible packets in increasing order of their finish time (*rank sorting*) and achieves high accuracy [41]. Eligibility evaluation and rank sorting are realized in the packet scheduler below.

Tassel maintains each flow's scheduling time (S_{flow_i}), which is the expected transmission time of their head packets, as mentioned earlier. When $flow_i$ is scheduled, Tassel fetches n packets and computes their start transmission time (S_{pkt_j}) and finish transmission time (F_{pkt_j}) as follows:

$$S_{pkt_j} = \begin{cases} S_{flow_i}, & j = 0 \\ F_{pkt_{j-1}}, & j > 0 \end{cases}, \quad F_{pkt_j} = S_{pkt_j} + \frac{L_{pkt_j}}{R_{flow_i}}$$

where L_{pkt_j} is the length of $packet_j$ and R_{flow_i} is the rate limit of $flow_i$.

Packet filtering (⑤). To bound the packet number for packet-level rate limiting, Tassel facilitates packet filtering that only filters a handful of packets that are the latest for transmission and discards the rest yet numerous packets. Specifically, we define *imminent window* as the time period within a scheduling latency from current system time T, and the *imminent packets* as those whose start time S_{pkt} falls within the imminent window. Figure 9 illustrates an example of packet filtering.

We observe that the maximum number of imminent packets (W) in RNIC is bounded, which is:

$$W = system\_packet\_rate \times scheduling\_latency$$

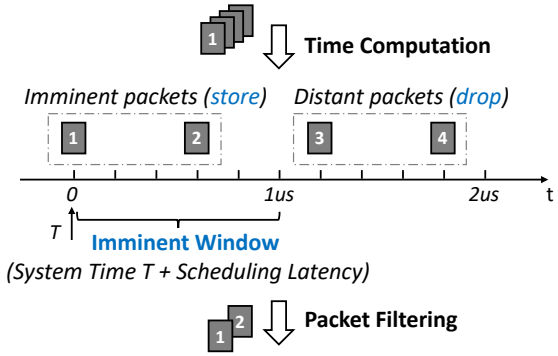


Figure 9: An example of packet filtering. After time computation, the first two packets are identified as imminent, as their start transmission time falls within the imminent window. In contrast, the remaining two packets are considered distant. Tassel then stores these first two imminent packets and forwards them to the packet scheduler for packet-level rate limiting, while discarding the distant packets.

In RNIC with a high packet rate of 110 Mpps and a scheduling latency of 1 µs, the total number of imminent packets typically amounts to a few hundred. It is worth noting that these packets can belong to different flows. Therefore, both large flows and multiple small flows can provide a sufficient number of imminent packets, allowing Tassel to achieve a high packet rate.

Packet filtering effectively reduces the number of packets that require packet-level rate limiting. After packet filtering, packet-level rate limiting only needs to manage a few hundred imminent packets. This allows packet-level rate limiting to be both fast and accurate.

Besides, packet filtering also reduces memory consumption and improves scalability. Specifically, Tassel only stores the packet metadata for imminent packets, including the transmission time and packet descriptors, while the remaining distant packets are dropped. Note that these dropped packets can be retrieved from the host in time when this flow is scheduled again, ensuring there is no compromise to performance and accuracy. Thus they can be dropped for now to conserve memory resources and improve scalability. In summary, by designing packet filtering, Tassel avoids storing unnecessary packets, which makes it memory-efficient and scalable. The overhead of this fetch-and-drop policy is low because the wasted PCIe bandwidth resulting from discarding descriptors of distant packets constitutes only a small fraction of the bandwidth used for transmitted packet data (detailed in §3.3.2).

After packet filtering, the packet filter passes the information required to reschedule this flow to the packet scheduler below. The information includes the transmission time and data offset of the dropped packet with the nearest transmission time, *e.g. packet3* in the example of Figure 9. This, in turn, notifies the flow scheduler to update and reschedule this flow. This scheduling time for this flow is set to the transmission time of the nearest dropped packet. There is a special case. When the configured rate limit is very high, it might happen that all fetched packets are identified as imminent, and no distant packets to drop. In this case, the scheduling time of this flow is updated to the current system time, indicating that the

flow should be scheduled immediately to fetch packets for the next transmissions.

Packet scheduling (⑦ & ⑧). The packet scheduler applies fast and accurate rate limiting for these imminent packets. Similar to the flow scheduler, the packet scheduler also sorts all imminent packets from various flows, and monitors the global timer to transmit packets on time.

Specifically, the packet scheduler maintains two ordered lists, one for sorting the start times (for *eligibility evaluation*) and another for sorting the finish times (for *rank sorting*). Tassel recognizes the eligible packets whose start times are smaller than the system time. Eligible packets then are dequeued from the sorted list of start time and enqueued into the sorted list of finish time. Whenever the link is idle, Tassel dequeues and transmits the head packet from the sorted list of finish time. Note that Tassel only dequeues a packet from the sorted list of finish time when the link is idle [4]. The packet scheduler deduces the link state by computing the transmission time of the last transmitted packet $packet\_length/link\_rate$.

Tassel's packet scheduler supports strict rate limiting with a fallback to weighted sharing by adjusting the timing rate in the global timer. When the aggregate flows' rate limits are lower than the link rate (*i.e.* the link is not oversubscribed), each cycle, the timer increases the system time by one clock time, *e.g.* 4 ns with a clock frequency of 250 MHz. When the aggregate flows' rate limits are larger than the link rate, the link is oversubscribed, and the timer proportionally slows down the system time. This enables the packet scheduler to transmit more packets per actual unit of time, thereby preventing packet accumulation and queuing resulting from oversubscription. The timer adjusts the timing rate according to the rate oversubscription factor Φ, which is defined as the sum of the active flows' rate limits divided by the link rate. After the time period δ, the timer updates the system time as follows:

$$T(t+\delta) = T(t) + \delta \times \max(1, \Phi)$$

3.3 Rate Limiter Architecture in RNIC

We then introduce integrating Tassel into RNIC architecture, ensuring compliance with RNIC's stringent resource and timing constraints (§3.3.2). Tassel's hierarchical rate limiter simultaneously achieves high performance, scalability, and accuracy at the system level. This is achieved through employing a combination of appropriate data structures, which meets diverse requirements at different stages of the workflow(§3.3.3).

3.3.1 Data Flow in RNIC

RNIC consists of tens of thousands of queue pairs (QPs). The user posts a work queue element (WQE) into a QP to issue a request. Each WQE corresponds to a message that consists of multiple packets, and all messages in the same QP are recognized as a flow. WQEs contain the message size (64 B ~ 2 GB) and the memory address of the data.

RNIC then processes various QPs' WQEs and transmits the packets, and the entire workflow can be divided into three steps:

[4]Dequeueing a packet too early can result in an incorrect scheduling order [46], as when dequeued packets are blocked by the busy link, eligible packets with smaller finish times may appear and should be sent first.

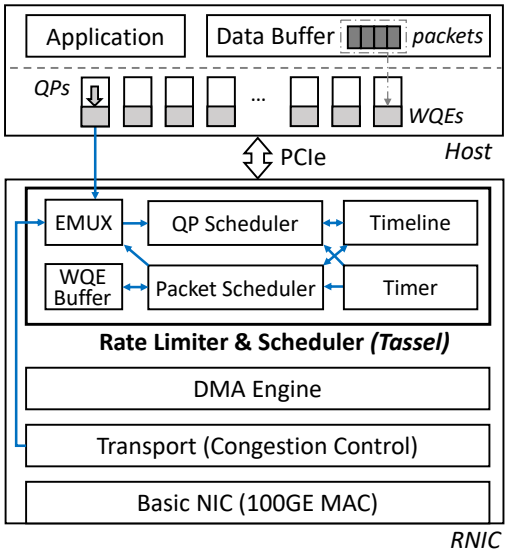


Figure 10: Tassel architecture in RNIC.

(1) Fetch WQEs. RNIC allocates a contiguous ring buffer in host memory for each QP to store WQEs. RNIC schedules tens of thousands of QPs and then fetches WQEs from the host via PCIe.
(2) Fetch packet data. After WQEs arrive, RNIC parses them, gets the memory address of the packet data, and then fetches it from the host via PCIe.
(3) Transmit packets. Packets from various flows are sent to the wire through a single FIFO after assembling the transport header and packet data.

Figure 10 illustrates a typical RNIC architecture, consisting of four parts: QP Scheduler, DMA Engine, Transport, and Basic NIC [51]. QP Scheduler schedules QPs. DMA engine fetches WQEs and data from host. Transport realizes the RDMA transport functionalities and assembles the packets when the payload arrives. The basic NIC sends and receives packets from the network via the MAC interface.

3.3.2 Tassel Architecture

We implement Tassel in RNIC by replacing original QP scheduler with the hierarchical rate limiter, which then schedules tens of thousands of QPs and hundreds of imminent packets. The architecture of RNIC with Tassel integrated is illustrated in Figure 10.

Event MUX (EMUX) gathers all scheduling-related events, including 1) the host doorbell that signals an active QP with new WQEs to process, 2) updates on rates and credits from congestion control, and 3) events that a QP is ready for rescheduling. EMUX passes the active QPs that have data and credits to send to *QP Scheduler* for flow-level rate limiting and scheduling.

The QP scheduler leverages *Timeline* to sort tens of thousands of QPs, and monitors *Timer* to schedule the nearest QP on time. It then applies adaptive batching and notifies *DMA Engine* to fetch multiple WQEs. We set the number of WQEs to fetch as the batch number of packets calculated in adaptive batching. We explain this correlation below. A WQE of a small message contains only one packet, and a WQE of a large message contains multiple packets. Adaptive

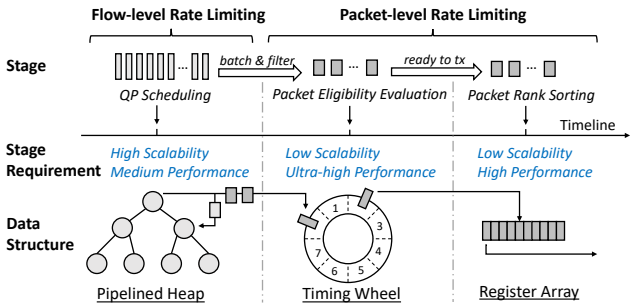


Figure 11: The hardware design of timeline in Tassel. Tassel uses the combination of data structures to meet the requirements of different stages.

batching calculates the maximum possible number of packets to be sent. To ensure the RNIC can send enough packets and avoid situations where packet transmission is allowed but not enough WQEs are fetched, we assume all WQEs are for small messages and fetch the corresponding number of WQEs. For large messages, after sending the imminent packets, some WQEs may remain unused. We drop these unused WQEs and fetch them next time. The wasted PCIe bandwidth is limited, and the RNIC can still reach the line rate (detailed in §3.5).

Fetched WQEs enter *Packet Scheduler*, which computes the transmission time for the packets in these WQEs. It then applies packet filtering and picks the imminent packets, stores their WQE in *WQE Buffer*, and drops the left. After filtering, the packet scheduler computes the rescheduling time of this flow and notifies EMUX.

Then the packet scheduler leverages Timeline to sort hundreds of imminent packets. Meanwhile, the packet scheduler queries the global timer to schedule and transmit the eligible packets with the smallest finish time.

At last, Tassel dequeues the packet, consumes its WQE in WQE Buffer, retrieves the packet data from the host via DMA engine, appends the RDMA header in *Transport*, and then transmits it to the network through *Basic NIC*.

3.3.3 Hardware Design of Timeline

The timeline module is the core of Tassel's hierarchical rate limiter, which sorts tens of thousands of QPs and hundreds of imminent packets, separately.

Sorting tens of thousands of flows requires high scalability but is less performance-sensitive, while scheduling hundreds of packets demands high performance and is less scalability-sensitive. Thus, as depicted in Figure 11, we combine the various data structures, including the pipelined heap (P-heap) [19], timing wheel [50], and register array [37], to match the needs of each stage. This allows us to achieve performance, scalability, and accuracy simultaneously. In brief, we employ an improved P-heap structure to handle the scalability requirements of QP scheduling, while utilizing a fast and high-precision timing wheel and a separate register array to perform packet eligibility evaluation and rank sorting, respectively.

Pipelined heap (P-heap). We leverage P-heap to store and sort QP scheduling time, which is less performance-sensitive but more scalability-sensitive. P-heap maintains a min heap and enables pipelining of the enqueue and dequeue operations, thereby allowing these operations to execute in essentially constant time. Specifically, P-heap can start enqueue or dequeue a packet every four cycles, *i.e.* 62.5 Mpps if running at 250 MHz clock. Also, P-heap's modular and pipelined design makes it friendly to the hardware timing. This allows P-heap to achieve a high clock frequency when supporting massive flows.

Timing Wheel. The timing wheel is a circular array of slots, with each slot representing a distinct time interval. Packets are inserted into corresponding slots based on their transmission time. The timing wheel supports efficient enqueue/dequeue operations. When inserting packets, the timing wheel calculates the slot offset using the packets' timestamps, allowing insertion in O(1) time. For dequeuing, the timer consistently checks the current time slot in the timing wheel; if there are packets, they are dequeued immediately, also in O(1) time. The resource overhead for the timing wheel is low. Considering a 4ns time granularity and a 1us time range (a typical imminent window size in RNIC), only 250 slots are needed.

Register Array. The register array stores and sorts the finish time of eligible packets, using the classic parallel compare-and-shift architecture [37]. The sorted register array supports enqueue, dequeue per two cycles, and can maintain a high clock rate when handling only a few hundred items, thus achieving a high packet rate of several hundred Mpps.

3.4 Design Summary

Performance Analysis. Tassel achieves high packet rate via adaptive batching and packet filtering that improve the performance of flow-level rate limiting and packet-level rate limiting, respectively. By design, the register array is currently the bottleneck of the system. It needs to sort the imminent packets as fast as possible and hence requires the amount to be small. The amount of imminent packets is determined by the scheduling latency multiplied by the system's desired packet rate, which is usually less than a few hundred. At this point, it can achieve a high packet rate of a few hundred million packets per second.

Scalability Analysis. Tassel is resource-efficient and achieves high scalability. It fetches packets only when the right time comes and stores only the WQEs of the imminent packets in RNIC for saving memory resources. Quantitatively, Tassel's memory consumption is negligible compared to 10 Mb total memory size or 375 B QPC [28]. It occupies 17 bytes for each QP, which includes 5 B for the scheduling states, 2 B in the schedule queue, and 10 B in the timeline module. Besides, Tassel consumes as much as 480 B to store WQEs, a constant value unrelated to the number of QPs, thanks to the fetch-and-drop policy and the bounded number of imminent packets. When supporting 10 K QPs, Tassel consumes theoretically 166.5 KB on-chip SRAM in total. In terms of computing resources, Tassel employs a combination of practical and scalable data structures to fulfill algorithm demands, save computing resources, and reduce circuit complexity. Finally, Tassel adopts shared instead of dedicated processing logic for QPs, which therefore limits the consumption of computing resources while making the logic irrelative to the number of QPs.

	ALM (K)	Register (K)	BRAM
P-Heap	3.7	3.4	41
Timing Wheel	24.1	1.8	0
Register Array	6.2	4.5	0
WQE Buffer	0.2	0.4	5
Total	34.2	10.1	46

Table 1: Resource usage of the Tassel prototype on the Intel Agilex FPGA. Tassel consumes 4.4% ALMs, 0.65% registers, and 0.44% BRAMs.

Accuracy Analysis. Tassel ensures accuracy through applying accurate packet-level rate limiting just before packet transmission, which adopts the time-based WF^{2}Q+ and strictly dispatches packets based on the calculated transmission time.

3.5 Discussion

PFC Handling. When the rate limiter's downstream processing logic is blocked, *e.g.*, receiving a PFC frame [24], Tassel will freeze the system by stopping the timer until the system resumes. This approach prevents packet accumulation and potential bursts while ensuring accuracy.

Extra PCIe Overhead. Since adaptive batching accurately regulates the number of packets prefetched, PCIe bandwidth is wasted only when transmitting large messages. In such cases, the wasted bandwidth resulting from discarding WQEs constitutes only a small fraction of the PCIe capacity (*e.g.* 2.2% extra PCIe overhead considering the worst case: single flow, 100 Gbps rate, 2 GB messages and 1024 B MTU) Hence, RNIC can still saturate the link bandwidth for large messages with this fetch-and-drop policy while achieving a high packet rate for small messages.

4 IMPLEMENTATION

We prototyped Tassel on an Intel Agilex FPGA board [6], running at a clock frequency of 250 MHz. Our FPGA board comprises 782 K ALMs, 1565 K registers, and 10464 BRAMs (26.16 MB SRAM), with a PCIe Gen3x16 interface and a 100Gbps Ethernet port. We use 24-bit timestamps, *i.e.* rank and predicate fields.

We realize 16 K QPs in Tassel and the resource consumption is broken down in Table 1. Tassel's implementation consumes 34.2 K ALMs, 10.1 K registers, and 115 KB on-chip SRAM in total, which occupies 4.4%, 0.65%, and 0.44% available resources of our FPGA and is quite low. Memory resources are mainly consumed by P-Heap and WQE Buffer for storing QP-related scheduling metadata and packet-related WQEs. Computing resources are primarily consumed by Timing Wheel and Register Array to calculate packet transmission time and implement high-performance sorting.

The resource consumption of Tassel is considerably low, and it can achieve high clock frequency, thereby making it easy to get integrated into various architectures. For example, the storage resources of Tassel account for only 2.6% of RNIC's [51], while it can achieve a comparable clock frequency of 250 MHz.

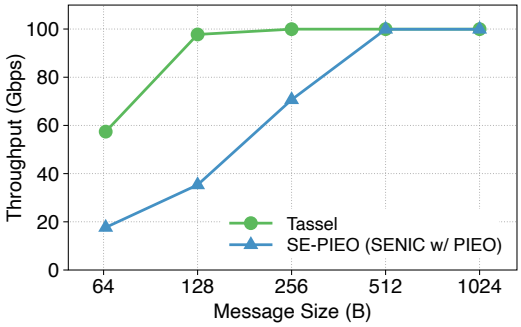


Figure 12: Performance. Tassel achieves a high packet rate, which aligns with the performance requirements of RNIC.

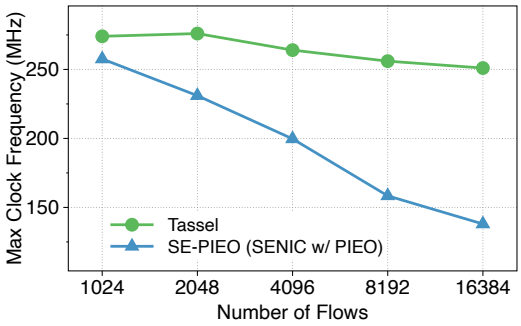


Figure 13: Performance. Tassel can achieve high clock frequency.

5 EVALUATION

In this section, we evaluate Tassel prototype across three metrics: performance, scalability, and accuracy. To serve as a baseline, we implement SE-PIEO [1, 2] (§2.3) on top of our FPGA. Our results reveal that:

- Tassel achieves high performance: it attains 125 Mpps high packet rate when supporting several thousand flows. Consequently, Tassel can transmit a packet every 8 ns, making it sufficient to schedule 100B packets at a 100 Gbps line rate, outperforming SE-PIEO by 3.6×.
- Tassel achieves high scalability: it supports tens of thousands of flows with very low resource usage, 7.5% to 25.6% as compared to SE-PIEO.
- Tassel achieves high accuracy: it precisely enforces rate limits ranging from 100 Kbps to 100 Gbps for tens of thousands of concurrent flows. Furthermore, it proportionally shares bandwidth when the link is oversubscribed.

5.1 Performance

We compare Tassel with SE-PIEO in terms of performance. We measure the aggregate throughput of multiple flows while increasing the message size from 64 B to 1024 B, as shown in Figure 12. Tassel can saturate the 100Gbps link bandwidth with a 128 B message size, achieving 125 Mpps high packet rate. In contrast, SE-PIEO achieves 34.5 Mpps, 27.6% as compared to Tassel. Tassel meets RNIC's desired high performance of 110 Mpps, allowing for seamless integration without performance penalty.

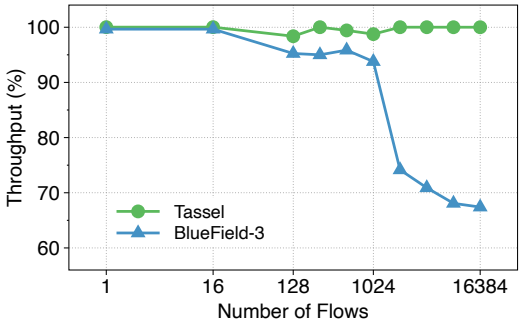


Figure 14: Performance. Tassel consistently maintains a high throughput as the number of flows increases.

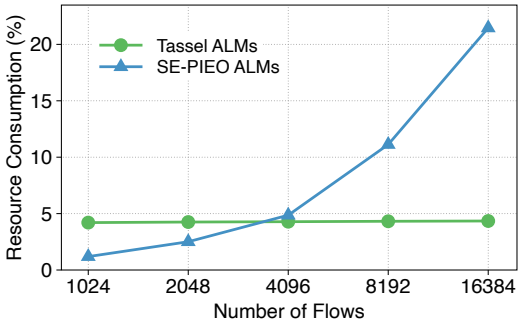


Figure 15: Scalability. Percentage of computing resource consumed (out of 782 K ALMs and 1565K registers).

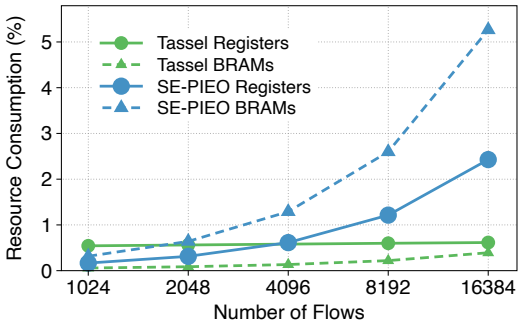


Figure 16: Scalability. Percentage of memory resource consumed (out of 26.16 MB SRAM).

We then measure the maximum clock frequency that the rate limiter can achieve varying the number of supported flows. Figure 13 shows that Tassel achieves a high clock frequency and exhibits a slower decline in frequency as the number of flows increases. In contrast, SE-PIEO's clock frequency declines from 257.6 MHz to 138 MHz, a reduction of 46.4%, when the flow number increases from 1 K to 16 K, due to the complex combinational logic.

We also compare Tassel with NVIDIA BlueField-3 in terms of the aggregate throughput while increasing the number of QPs from 1 to 16 K. Each QP is assigned the same rate limit, calculated as the total link bandwidth divided by the number of QPs. We generate the traffic using the standard *perftest* with default settings and measure the aggregate throughput via *mlnx_perf*. As Figure 14 shows, Tassel consistently achieves line rate as the number of QPs increases, while BlueField-3's throughput decreases significantly when the number of QPs exceeds 1024.

Tassel's high performance stems from the algorithm's low time complexity and the practical hierarchical design. Tassel's algorithm supports enqueuing/dequeuing a packet in constant cycles. And the hierarchical design enables Tassel to combine the scalable P-heap, fast timing wheel, and register array to achieve high clock frequency. On the one hand, P-heap utilizes a modular and pipelined design. It is friendly to the hardware timing when supporting massive flows and helps Tassel achieve high clock frequency. On the other hand, timing wheel and register array enable rapid packet transmission within two cycles. In contrast, the clock frequency of SE-PIEO is limited to only 138 MHz due to the high circuit complexity of storing and accessing the ordered list stored in SRAM with one level of indirection, and it consumes 4 cycles per operation.

We believe Tassel's design can scale to higher bandwidth scenarios, such as 400 or 800 Gbps. On one hand, our prototype achieves a packet rate of 125 Mpps, and given a larger MAC capacity, it can achieve a throughput of nearly 1.0 Tbps with a 1024 B MTU. On the other hand, our FPGA prototype can achieve higher packet rates by upgrading to more advanced FPGA chips or ASIC platforms, allowing the design to operate at higher clock frequency.

5.2 Scalability

In this section, we evaluate the scalability of Tassel by examining how the consumption of the computing and memory resources scales with the number of supported flows.

As shown in Figure 15 and Figure 16, Tassel exhibits very low resource usage. Even when supporting 16k flows, the computing resources utilized amount to less than 5% of our FPGA, while memory resources account for less than 1%. Moreover, as the flow number increases, the growth in the number of ALMs and registers used is not significant, with only the BRAMs for storing the flows' scheduling states exhibiting a linear increase. In comparison, with 16k flows, SE-PIEO consumes 4.9× more ALMs, 3.9× more registers, and 13.4× more BRAMs. Additionally, as the flow number increases, SE-PIEO's resource consumption grows at a square root rate due to the requirement of log(N) comparators.

5.3 Accuracy

We evaluate Tassel's accuracy by configuring rate limits ranging from 100 Kbps to 100 Gbps for different flows and timestamping transmitted packets with a 4 ns clock resolution to compute the rate achieved. Figure 17 displays the rate limiting accuracy results for a single flow. Tassel accurately enforces any given rate limit, including both small and large rates. These results highlight the effectiveness of Tassel's timeline design, which can support large inter-packet gaps for small rates and high time precision required by large rates. In comparison, BlueField-3 supports rate limits ranging from 200 Kbps to the line rate with a precision of 200 Kbps [8]. However, we observed that the actual minimum rate limit supported by BlueField-3 is 400 Kbps. When users set the rate limit to 200 Kbps, we measured an actual throughput of 376.25 Kbps, as shown in Figure 17. Regarding precision, our measurements confirmed an actual precision of 200 Kbps, consistent with the datasheet.

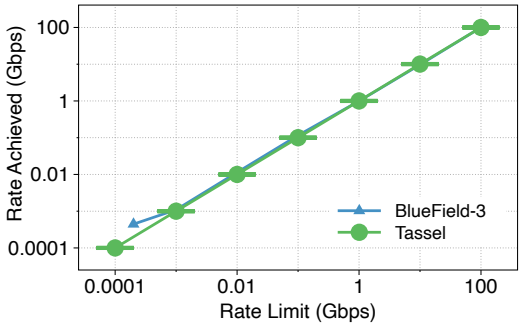


Figure 17: Accuracy. Tassel supports accurate rate limiting for a single flow, ranging from 100 Kbps to 100 Gbps.

We then demonstrate that Tassel can precisely enforce configured rate limits for tens of thousands of concurrent flows. We divide several thousand flows into six groups, configuring distinct rate limits for flows in different groups. The aggregate rate is less than the link bandwidth, ensuring the link is not oversubscribed. As shown in Figure 18a, Tassel's accurate rate limiting enables each flow to achieve the configured rate limit precisely.

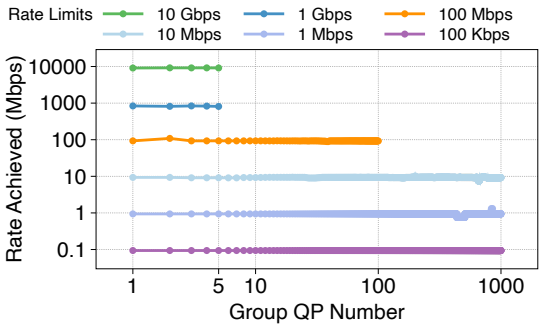
When the aggregate rate exceeds the link rate, resulting in link oversubscription, Tassel can proportionally adjust the rate across flows according to the original rate limit and the over-subscription factor Φ to share the bandwidth proportionally. As shown in Figure 18b, when Φ equals two, the rate achieved by each flow is precisely half of the configured rate limits.

Tassel's high accuracy stems from its adoption of time-based WF^{2}Q+ algorithm as well as its efficient and comprehensive hardware design.

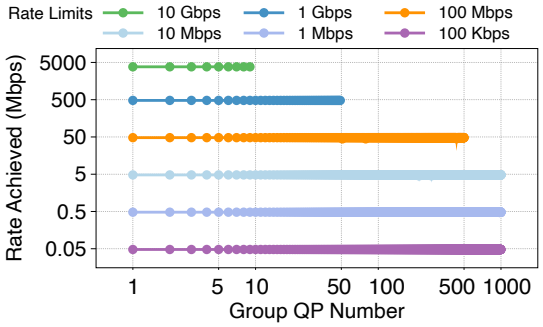
6 RELATED WORK

Rate limiting and packet scheduling. Rate limiting has long been a classic research problem [14, 41, 43] and is used in current congestion control and bandwidth allocation systems [17, 26, 29, 30, 40, 55]. Software solution Carousel [43] improves the efficiency and accuracy of rate limiting in software. Hardware solutions like SENIC [41] offload this task to improve accuracy and reduce the CPU overhead. Via the hierarchical design, Tassel improves the packet rate of rate limiter in hardware. In addition, rate limiting is a specific task of packet scheduling and thus can be supported by programmable packet schedulers [12, 16, 20, 34, 44, 45, 49, 52–54] such as PIFO [48] and PIPO [54]. These approaches also follow the one-packet-per-sorting transmission and our hierarchical idea could apply to them to improve the performance.

Priority queues in hardware. Research on priority queues [19, 46, 48, 52, 53] can be applied to sort flows and packets. There are various types of priority queues, each with distinct characteristics: some offer high performance [48], while others have a very large capacity [19]. These studies are orthogonal to our work, allowing us to select suitable data structures from them to meet the diverse needs of different stages in Tassel's hierarchical rate limiter. For instance, we could use the BMW Tree [52], an improved version of P-heap, to replace the original P-heap for flow-level rate limiting.



(a) No oversubscription.



(b) Oversubscription. Φ=2.

Figure 18: Accuracy. (a) Tassel maintains the accuracy of different configured rate limits with numerous flows. (b) Tassel supports weighted fair sharing when the link is oversubscribed.

7 CONCLUSION

We present the design and implementation of Tassel, a fast, scalable, and accurate rate limiter for RDMA NICs. Tassel's hierarchical rate limiter combines scalable flow-level rate limiting with fast and accurate packet-level rate limiting, thereby achieving high performance, scalability, and accuracy at the system level. Tassel enables RNICs to precisely allocate bandwidth resources at end hosts. We hope this can inspire more innovative protocols in high-speed networks and domain-specific networks, aiming at a congestion-free datacenter. Specifically, general datacenter networks can divide link bandwidth into multiple portions, allowing the sender to request and reserve bandwidth before transmission, thereby avoiding resource contention and incast. Furthermore, AI training clusters can leverage the predictability of the training traffic to assign the sending rate among GPU pairs and avoid link congestion.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd Masoud Moshref for their constructive comments. We thank Layong Luo for all the discussions and suggestions to this paper. This work is supported in part by the Hong Kong RGC TRS T41-603/20R, GRF 16213621, ITF ACCESS, NSFC 62062005, Key-Area R&D Program of Guangdong Province (2021B0101400001), and a joint HKUST-ByteDance research project. Kai Chen is the corresponding author.

REFERENCES

[1] 2019. PIEO's Open-Source Implementation. https://github.com/vishal1303/PIEO-Scheduler. (2019).

[2] 2019. SENIC's Open-Source Implementation. https://github.com/gengyl08/SENIC. (2019).

[3] 2020. Mellanox ConnectX-6 Product Brief. https://network.nvidia.com/sites/default/files/doc-2020/pb-connectx-6-en-card.pdf. (2020).

[4] 2021. Mellanox ConnectX-7 Product Brief. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/connectx-7-datasheet-Final.pdf. (2021).

[5] 2021. OFED Perftest. https://github.com/linux-rdma/perftest/. (2021).

[6] 2023. Intel Agilex FPGA. https://www.intel.com/content/www/us/en/products/sku/225389/intel-agilex-7-fpga-fseries-023-r25a/specifications.html. (2023).

[7] 2023. NVIDIA BLUEFIELD-3 DPU Data Sheet. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf. (2023).

[8] 2023. NVIDIA DOCA Programmable Congestion Control Programming Guide. https://docs.nvidia.com/sdk-v2.2.0/pdf/pcc-programming-guide.pdf. (2023).

[9] 2024. Mellanox userland tools and scripts. https://github.com/Mellanox/mlnx-tools. (2024).

[10] Saksham Agarwal, Qizhe Cai, Rachit Agarwal, David Shmoys, and Amin Vahdat. 2024. Harmony: A Congestion-free Datacenter Architecture. In *Proc. NSDI*.

[11] Saksham Agarwal, Arvind Krishnamurthy, and Rachit Agarwal. 2023. Host Congestion Control. In *Proc. SIGCOMM*.

[12] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. 2020. SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues. In *Proc. NSDI*.

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

[14] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling programmable transport protocols in high-speed NICs. In *Proc. NSDI*.

[15] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. 2010. A view of cloud computing. *Commun. ACM* (2010).

[16] Nirav Atre, Hugo Sadok, and Justine Sherry. 2024. BBQ: A Fast and Scalable Integer Priority Queue for Hardware Packet Scheduling. In *Proc. NSDI*.

[17] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. 2011. Towards predictable datacenter networks. In *Proc. SIGCOMM*.

[18] Jon CR Bennett and Hui Zhang. 1997. Hierarchical packet fair queueing algorithms. *IEEE/ACM Transactions on networking* (1997).

[19] Ranjita Bhagwan and Bill Lin. 2000. Fast and scalable priority queue architecture for high-speed network switches. In *Proc. INFOCOM*.

[20] Peixuan Gao, Anthony Dalleggio, Jiajin Liu, Chen Peng, Yang Xu, and H Jonathan Chao. 2024. Sifter: An Inversion-Free and Large-Capacity Programmable Packet Scheduler. In *Proc. NSDI*.

[21] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, et al. 2021. When Cloud Storage Meets RDMA. In *Proc. NSDI*.

[22] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. 2013. Achieving high utilization with software-driven WAN. In *Proc. SIGCOMM*.

[23] Shuihai Hu, Wei Bai, Kai Chen, Chen Tian, Ying Zhang, and Haitao Wu. 2018. Providing bandwidth guarantees, work conservation and low latency simultaneously in the cloud. *IEEE Transactions on Cloud Computing* (2018).

[24] IEEE. 2008. 802.1 Qbb—Priority-based Flow Control. (2008).

[25] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. 2015. Silo: Predictable message latency in the cloud. In *Proc. SIGCOMM*.

[26] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. 2013. EyeQ: Practical Network Performance Isolation at the Edge. In *Proc. NSDI*.

[27] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. 2024. MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs. In *Proc. NSDI*.

[28] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be general and fast. In *Proc. NSDI*.

[29] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. 2015. BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing. In *Proc. SIGCOMM*.

[30] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, et al. 2020. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proc. SIGCOMM*.

[31] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. 2017. The quic transport protocol: Design and internet-scale deployment. In *Proc. SIGCOMM*.

[32] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. 2019. HPCC: High precision congestion control. In *Proc. SIGCOMM*.

[33] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, et al. 2019. Snap: A microkernel approach to host networking. In *Proc. SOSP*.

[34] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2015. Universal packet scheduling. In *Proc. HotNets*.

[35] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based congestion control for the datacenter. In *Proc. SIGCOMM*.

[36] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. 2018. Revisiting network support for RDMA. In *Proc. SIGCOMM*.

[37] Sung-Whan Moon, Jennifer Rexford, and Kang G Shin. 2000. Scalable hardware priority queue architectures for high-speed packet switches. *IEEE Transactions on computers* (2000).

[38] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. 2018. Restructuring endpoint congestion control. In *Proc. SIGCOMM*.

[39] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. 2014. Fastpass: A centralized" zero-queue" datacenter network. In *Proc. SIGCOMM*.

[40] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C Mogul, Yoshio Turner, and Jose Renato Santos. 2013. Elasticswitch: Practical work-conserving bandwidth guarantees for cloud computing. In *Proc. SIGCOMM*.

[41] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. 2014. SENIC: Scalable NIC for end-host rate limiting. In *Proc. NSDI*.

[42] Waleed Reda, Marco Canini, Dejan Kostić, and Simon Peter. 2022. RDMA is Turing complete, we just did not know it yet!. In *Proc. NSDI*.

[43] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. 2017. Carousel: Scalable traffic shaping at end hosts. In *Proc. SIGCOMM*.

[44] Ahmed Saeed, Yimeng Zhao, Nandita Dukkipati, Ellen Zegura, Mostafa Ammar, Khaled Harras, and Amin Vahdat. 2019. Eiffel: Efficient and flexible software packet scheduling. In *Proc. NSDI*.

[45] Naveen Kr Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. 2020. Programmable calendar queues for high-speed packet scheduling. In *Proc. NSDI*.

[46] Vishal Shrivastav. 2019. Fast, scalable, and programmable packet scheduler in hardware. In *Proc. SIGCOMM*.

[47] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. 2019. Shoal: A network architecture for disaggregated racks. In *Proc. NSDI*.

[48] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. Programmable packet scheduling at line rate. In *Proc. SIGCOMM*.

[49] Brent Stephens, Aditya Akella, and Michael Swift. 2019. Loom: Flexible and efficient NIC packet scheduling. In *Proc. NSDI*.

[50] George Varghese and Tony Lauck. 1987. Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility. In *Proc. SOSP*.

[51] Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, Xinchen Wan, Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, Tianhao Wang, Weicheng Ling, Kejia Huo, Pingbo An, Kui Ji, Shideng Zhang, Bin Xu, Ruiqing Feng, Tao Ding, Kai Chen, and Chuanxiong Guo. 2023. SRNIC: A Scalable Architecture for RDMA NICs. In *Proc. NSDI*.

[52] Ruyi Yao, Zhiyu Zhang, Gaojian Fang, Peixuan Gao, Sen Liu, Yibo Fan, Yang Xu, and H Jonathan Chao. 2023. BMW Tree: Large-scale, High-throughput and Modular PIFO Implementation using Balanced Multi-Way Sorting Tree. In *Proc. SIGCOMM*.

[53] Zhuolong Yu, Chuheng Hu, Jingfeng Wu, Xiao Sun, Vladimir Braverman, Mosharaf Chowdhury, Zhenhua Liu, and Xin Jin. 2021. Programmable packet scheduling with a single queue. In *Proc. SIGCOMM*.

[54] Chuwen Zhang, Zhikang Chen, Haoyu Song, Ruyi Yao, Yang Xu, Yi Wang, Ji Miao, and Bin Liu. 2021. Pipo: Efficient programmable scheduling for time sensitive networking. In *Proc. ICNP*.

[55] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion control for large-scale RDMA deployments. In *Proc. SIGCOMM*.