DRAGONKING: A Scalable and High-Throughput Rate Limiter by Enabling WF^{2}Q+ on Programmable Switches for Cloud Networks

Gonglong Chen, *Member, IEEE*, Kejiang Ye, *Senior Member, IEEE*, Kai Chen, *Senior Member, IEEE*, and Chengzhong Xu, *Fellow, IEEE*

***Abstract*—In contemporary cloud architectures, an increasing number of cloud service providers are adopting programmable switches to deliver cloud network services (e.g., load-balancing gateways) for millions of tenants. The rate limiters are essential for cloud networks to execute network policies such as congestion control and traffic isolation on programmable switches. Most existing rate limiters utilize the token bucket algorithm, which suffers from scalability issues and substantial control overhead, impacting bandwidth utilization. The WF^{2}Q+ (Worst-case Fair Weighted Fair Queueing Plus) algorithm, known for its scalability and accuracy, is gaining traction but faces challenges on programmable switches. The hardware limitations hinder key operations of WF^{2}Q+ (sorting and scheduling) in a single switch pipeline. This paper introduces DRAGONKING, a novel rate-limiting system that enables WF^{2}Q+ through a multi-pipeline sorting and scheduling design. DRAGONKING enhances scalability and throughput while maintaining high accuracy. It achieves this by strategically balancing bandwidth across multiple pipelines, allowing for timely packet scheduling. DRAGONKING is implemented and evaluated on Barefoot Tofino switch. Results show that DRAGONKING supports up to two million entries and delivers line-rate throughput, achieving 1.9× improvements compared with token bucket-based limiters. Moreover, it can maintain over 99% accuracy, precisely enforcing rate limits from 10 Gbps to 100 Gbps.**

***Index Terms*—Rate limiter, programmable switches, cloud networks.**

Received 1 May 2025; revised 5 November 2025; accepted 13 December 2025; approved by IEEE TRANSACTIONS ON NETWORKING Editor K. Lee. Date of publication 28 January 2026; date of current version 17 February 2026. This work was supported in part by the National Key Research and Development Program of China under Grant 2025YFE0204100; in part by the Science and Technology Development Fund (FDCT), Macao, SAR, under Grant 0074/2025/AMJ; in part by Guangdong Basic and Applied Basic Research Foundation under Grant 2023B1515130002; in part by the Key Research and Development and Technology Transfer Program of Inner Mongolia Autonomous Region under Grant 2025YFHH0110; and in part by Shenzhen Basic Research Program under Grant JCYJ20250604183035046 and Grant KJZD20230923113800001. *(Corresponding author: Kejiang Ye.)*

Gonglong Chen and Kejiang Ye are with Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen 518055, China, and also with the University of Chinese Academy of Sciences, Beijing 100049, China (e-mail: gl.chen2@siat.ac.cn; kj.ye@siat.ac.cn).

Kai Chen is with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong, China (e-mail: kaichen@cse.ust.hk).

Chengzhong Xu is with the State Key Laboratory of IoTSC, Faculty of Science and Technology, University of Macau, Macau, China (e-mail: czxu@um.edu.mo).

Digital Object Identifier 10.1109/TON.2026.3658037

I. INTRODUCTION

IN MODERN cloud architectures, an increasing number of cloud service providers (CSPs) [1], [2], [3], [4], [5], [6] are adopting programmable switches to deliver cloud network services (e.g., physical-virtual gateways [2], load-balancing gateways [6]). This trend is driven by the ability of programmable switches to provide line-rate forwarding performance (e.g., Tbps [7]) and high cost-effectiveness (e.g., 97.2% cost reduction [8]).[1] Programmable switches thus desire scalable, high-throughput and accurate rate limiters. Specifically, rate limiters are essential for cloud networks to efficiently implement network policies such as congestion control [9], [10] and traffic isolation [11], [12] on programmable switches. The rate limiters enforce specific flow rates to address the challenges of the bursty and unpredictable traffic in data centers [12], [13], [14], [15] and isolate traffic among users to ensure stable application performance [16], [17], [18], which is crucial for managing tens of thousands of flows in cloud networks. Additionally, it is vital that these rate limiters should maintain low forwarding overhead to prevent any reduction in the throughput of programmable switches, as any slowdown could impair overall performance.

There are already several rate limiters implemented on programmable switches, and all of them use the token bucket algorithm [19], [20], [21], [22]. In this algorithm, a packet is forwarded only if there are sufficient tokens in the token bucket; otherwise, it is dropped. These limiters typically use either the switch ASIC's (e.g., Tofino [7]) built-in meter function [19] or manage tokens through a register for stateful tracking [21], [22]. However, meter-based methods [19], [20] struggle with *scalability* due to their high resource consumption. For instance, each table entry requires 32 bytes to record the bucket's capacity and token rate, consuming about 70% of the total entry space. These parameters are hardcoded in the ASIC, limiting flexibility. Register-based methods [21], [22], on the other hand, face *low forwarding throughput* issues due to significant control traffic overhead needed to replenish tokens.

Recently, WF^{2}Q+ (Worst-case Fair Weighted Fair Queueing Plus) [23] has gained significant attention and has been

[1]Decreasing 110/Gbps to 3/Gbps [8] compared with the server-based approach.

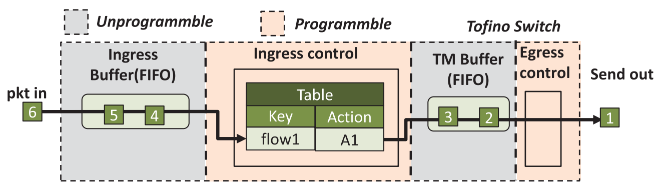


Fig. 1. The pipeline flow of the typical programmable switch ASIC, i.e., Tofino. The hardware constraints (e.g., unprogrammable parts) limit Tofino's ability to support arbitrary packet ordering (e.g., storing) in the two buffers.

utilized in various cloud scenarios [11], [24], [25] to achieve scalable and accurate rate limiting. WF^{2}Q+ is an accurate and classic time-based rate-limiting algorithm [23]. The key idea of WF^{2}Q+ is to compute the Virtual Start Time (VST) and Virtual Finish Time (VFT) for each packet and then *sort* packets based on VFT. The packet with the smallest VFT is then *scheduled* to be forwarded first [23]. Although WF^{2}Q+ has been successfully adapted to RNIC (RDMA NIC) [11] for scalable and accurate rate limiting, it cannot be directly applied on programmable switches. The main challenge is that the key operations of WF^{2}Q+ (i.e., *sorting* and *scheduling*) require *programmable* support of the packet buffer, e.g., the packet reordering and fetching, which are not supported on the switch ASIC (e.g., Tofino [7]). Figure 1 shows that Tofino processes packets in a match-action manner for line-rate forwarding, which is the typical behavior of one of its four switch pipelines [1]. Packets enter and exit the two primary buffers (i.e., Ingress buffer and Traffic manager buffer) in FIFO order [7], which are *unprogrammable*.

To overcome these limitations, we propose a system called DRAGONKING,[2] which achieves the scalable, high-throughput and accurate rate limiter, by enabling WF^{2}Q+ on programmable switches (e.g., Tofino [7]) for cloud networks. Instead of *sorting* and *scheduling* in a *single unprogrammable* pipeline buffer, we propose to implement a *group-level* sorting in the *programmable* control component across *multiple* pipeline buffers. At this core, DRAGONKING renovates the workflow of the rate limiter from the *packet-level* sorting to the *group-level* sorting while maintaining high accuracy by carefully managing the number of packets in each pipeline buffer.

Specifically, **1) to achieve the scalability**, DRAGONKING first significantly reduces the entry size by maintaining only two necessary parameters, i.e., the limited rate and the last finish time of packet (reduced by up to 57% as stated in Section III-B). Additionally, a co-designed table placement approach for both ingress and egress is proposed to further enhance scalability. **2) To achieve the high-throughput**, DRAGONKING strategically balances bandwidth utilization across multiple pipelines to maintain consistent data flow and prevent buffer overflows and packet drops. This careful management of resources ensures that the system can handle high traffic volumes efficiently (Section III-C). **3) To maintain the high accuracy**, DRAGONKING actively manages the timing of packet transmissions to prevent missed deadlines. It adaptively relocates packets from pipelines scheduled for later transmission to earlier ones. This strategy ensures that packets are processed promptly, aligning closely with their scheduled transmission times, thus maintaining strict adherence to rate limits (Section III-D).

We implement and evaluate a DRAGONKING prototype on Barefoot Tofino switch [26], demonstrating its capability to achieve the scalability, high-throughput and high accuracy at the same time (Section V). The evaluation shows that in one Tofino switch, DRAGONKING can support two million limiter entries, and line-rate forwarding throughput, achieving $1.9\times$ improvements at most compared with the token bucket-based approaches [19], [21], [22]. In the meantime, DRAGONKING can preserve high accuracy as 99%+, precisely enforcing rate limits from 10 Gbps to 100 Gbps.

This paper makes the following contributions:

- We analyze and identify performance issues in previous works and reveal that the root cause stems from the design choice of the rate-limiting algorithm (i.e., token bucket). The mechanism of tracking bucket status results in a large entry size, which diminishes scalability. The process of token addition generates excessive control traffic, reducing the forwarding throughput (x II-C).
- We design DRAGONKING, a WF^{2}Q+ enabled rate limiter for programmable switches. DRAGONKING addresses the challenges of implementing the core components of WF^{2}Q+ (e.g., packet sorting) on programmable switches by proposing a multi-pipeline-based sorting and scheduling approach. It overcomes the hardware constraint that prevents packets in the buffers of a single pipeline from being directly reordered (see Section III).
- We implement a DRAGONKING prototype on the Tofino switch [26]. Experiments demonstrate that DRAGONKING achieves our design goals of scalability, high-throughput and high accuracy (x IV and x V).

This work does not raise any ethical issues.

II. BACKGROUND AND MOTIVATIONS

In this section, we first introduce the role of rate limiters in programmable switches used within cloud networks and outline their desired key features. Next, we examine the limitations of existing approaches. Finally, we present our insight and key ideas.

A. The Importance of Rate Limiting in Programmable Switches for Cloud Networks

In cloud networks, rate limiters are essential for cloud service providers (CSPs) to manage data flow efficiently [11], [25], [27]. Programmable switches, known for their high forwarding performance and cost-effectiveness, are increasingly favored for deploying cloud services [1], [2], [3], [4], [5], [6], [8], [28], [29], [30]. By integrating rate limiters into these programmable switches, CSPs can maintain this cost-effectiveness while enhancing service quality.

Rate limiting on programmable switches aids in: 1) Upholding Service Level Agreements (SLAs) to ensure fair bandwidth

[2]In ancient Chinese mythology, DRAGONKING is the deity responsible for controlling the traffic of rain, similar to the capability of rate limiting in communication traffic.

distribution among tenants and service instances (Section II-A1), 2) Managing congestion (Section II-A2), 3) Mitigating potential network attacks (Section II-A3).

1) SLA Guarantee and Traffic Isolation: In scenarios where multiple tenants subscribe to a cloud gateway, the rate-limiting becomes paramount. This mechanism ensures that each tenant achieves the network performance metrics specified in their respective SLAs. These agreements typically define parameters such as bandwidth, latency, and packet loss rate [10], [11].

By implementing the rate-limiting, CSPs can prevent any single tenant from monopolizing excessive resources, thereby achieving effective traffic isolation. This strategy has been widely adopted by renowned cloud service providers such as Tencent [31], and Alibaba [32].

2) Traffic Shaping and Congestion Control: By limiting the transmission rate of specific traffic, the cloud gateway can smooth traffic bursts, thereby avoiding congestion [9]. For example, in the context of load-balancing gateways (LB), which distribute network traffic across multiple backend servers [33], rate limiters are employed to prevent server overload and moderate traffic surges [6]. The absence of effective rate limiting can lead to prolonged system response times and potential service disruptions, critically compromising user experience [9].

3) Network Attack Mitigation: Rate limiting also plays an important role in cloud gateway attack defense. It can prevent network resources from being exhausted by malicious traffic or overloaded by legitimate traffic [12], [27]. For example, PSP (Proactive Surge Protection) [12] actively manages network traffic through bandwidth isolation and priority control to defend against DDoS attacks. It utilizes rate limiting to dynamically control traffic of different priorities, particularly prioritizing the dropping of low-priority traffic when network capacity is limited. Through this approach, PSP can significantly reduce up to 97.58% of packet loss and 90.36% of traffic loss when an attack occurs.

B. The Desired Features for Rate Limiters on Programmable Switches

In cloud networks, the programmable switches desire a rate limiter to be scalable, high-throughput, and accurate:

- **Scalable.** The rate limiter must support rate limiting for millions of flows at programmable switches, considering the scale of tenants.
- **High-throughput.** The rate limiter should impose minimal impact on forwarding performance while maintaining high transmission throughput.
- **Accurate.** The rate limiter should precisely enforce the given rate on each flow and inject inter-packet gaps to smooth the traffic, preventing bursts.

C. Existing Works and Limitations

1) Token Bucket-Based Rate Limiter: The token bucket algorithm [23], [34], [35] is widely used in programmable switches [20], [21], [22] due to its simplicity and effectiveness in achieving rate limiting. It regulates packet flow based on token availability in two buckets: the Committed Bucket (CB) and either the Excess Bucket (EB) [35] or the Peak Bucket (PB) [34], as defined in RFC-2697 (Single Rate Three Color Marker) [35] and RFC-2698 (Two Rate Three Color Marker) [34]. According to RFC-2697 [35], packets first attempt to obtain tokens from the CB. If sufficient tokens are available, the packet is marked green; otherwise, it is marked yellow, and the algorithm proceeds to draw from the EB. If the EB is also depleted, the packet is marked red. Green packets are forwarded immediately, while yellow or red packets may be dropped depending on the rate limiter's policy, ensuring effective traffic control. The key difference between RFC-2697 [35] and RFC-2698 [34] is that RFC-2697 uses a single token rate CIR (Committed Information Rate) for both CB and EB, while RFC-2698 adds a separate rate PIR (Peak Information Rate) for the PB, allowing more flexible regulation.

2) The Implementation of Commercial Hardware: The programmable switch ASIC Tofino [7] has a built-in implementation of the token bucket function, which complies with the standards of RFC-2698 [21], [34], [36]. The built-in *meter* function provides four parameters that can be modified by the switch control plane to adjust the limited rate, namely CIR, CBS (Committed Burst Size, the size to CB), PIR, and PBS (Peak Burst Size, the size to PB). For each flow that is rate-limited, the *meter* periodically adds tokens to token buckets according to the set rate of CIR/PIR. When the data plane receives a packet, if it hits a rate-limiting entry, it will consume the number of tokens corresponding to the number of bytes of that packet and assign different colors to the packet accordingly. The gateway can decide to drop or forward the packet based on the color.

Performance issues. When employing the *meter* in a rate-limiting table, four token bucket parameters are automatically included with each table entry [7]. The length of these parameters is fixed by the ASIC and is unchangeable, leading to longer table entries. For example, a table entry that uses a five-tuple for the rate-limiting match key, combined with the four token bucket parameters as the action data, occupies 45 bytes.[3] On a standard Tofino chip equipped with four pipelines [1], such a setup restricts support to only a few hundred thousand rate limiters within a typical gateway, like the load balance gateway ConWeave [37] as we evaluated in Section V.

To expand the scale of rate limiters, Wang et al. [21] utilize the *register* functionality of the Tofino chip to implement the token bucket algorithm RFC-2698. While this approach enhances the capacity, it also requires the use of additional control packets to replenish tokens. These packets add tokens but consume the switch's forwarding bandwidth. This consumption substantially reduces the programmable switch's throughput. For example, the smallest token addition TCP packet, measuring 55 bytes,[4] can use up at least 43% of the link bandwidth [21]. SwRL [22] also utilizes *register* to implement the token bucket algorithm RFC-2698. It eliminates the token addition control traffic by utilizing the time intervals

[3]Including 13 bytes for the five-tuple (source address, destination address, source port, destination port, and protocol) and 32 bytes for the token bucket parameters (CIR, CBS, PIR, PBS).

[4]Including a 14-byte Ethernet header, a 20-byte IP header, and a 21-byte TCP header.

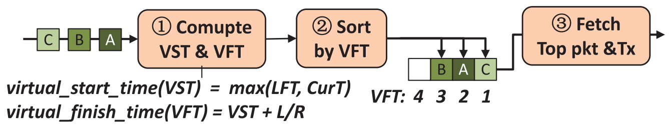

C B A
① Comupte VST & VFT
② Sort by VFT
③ Fetch Top pkt &Tx
B A C
VFT: 4 3 2 1
virtual_start_time(VST) = max(LFT, CurT)
virtual_finish_time(VFT) = VST + L/R

Fig. 2. The procedure of WF^{2}Q+. VST is the virtual start time; LFT denotes the last finish time of a flow, which represents the flow's final virtual finish time; CurT is the current time; VFT is the virtual finish time; L is the length of the packet; R is the limited rate of this packet.

between consecutive packet arrivals to compute token accumulation for enforcing rate limits. However, it still incurs a significant overhead (nearly 37 bytes per entry[5]) to record the necessary information for performing the above transformation.

3) Recent Studies: WF^{2}Q+ (Worst-case Fair Weighted Fair Queueing Plus) [23] is an accurate and classic time-based rate limiting algorithm, widely adopted in rate limiters to achieve scalability and accuracy [24]. Essentially, WF^{2}Q+ computes the Virtual Start Time (VST) and the Virtual Finish Time (VFT) for each packet within a flow. It then schedules the flow whose leading packet has the smallest VFT, and the VST is less than the current time [23] (As shown in Figure 2). Theoretically, WF^{2}Q+ effectively controls the traffic rate per flow and guarantees bounded transmission delays for each packet. Thus, WF^{2}Q+ is widely considered accurate [11], [23], [24].

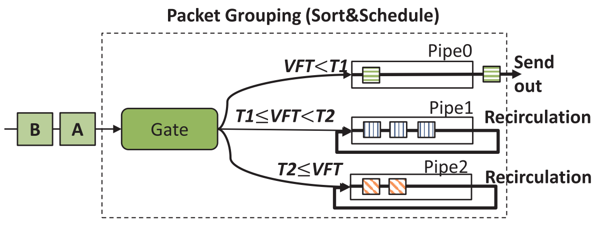
SENIC [24] and PIEO [23] implement WF^{2}Q+ by sorting the VFT of packets, where only the first packet is sent out each time, impacting the throughput. To improve the transmission efficiency, Tassel [11] transmits multiple packets after sorting all flows, thereby maintaining accuracy and scalability.

Limitations. However, the optimizations and implementations of existing works concerning WF^{2}Q+ are primarily based on FPGA or server platforms and do not address the challenges of implementing WF^{2}Q+ on the programmable switch platform, e.g., Tofino ASIC [7]. The core capabilities required for implementing WF^{2}Q+ include: **First,** adjusting the packet order in the buffer as needed (e.g., sorting packets based on their VFT); **Second,** indexing packets by memory address (e.g., retrieving leading packets for transmission). Unfortunately, these functionalities do not apply to Tofino [7].

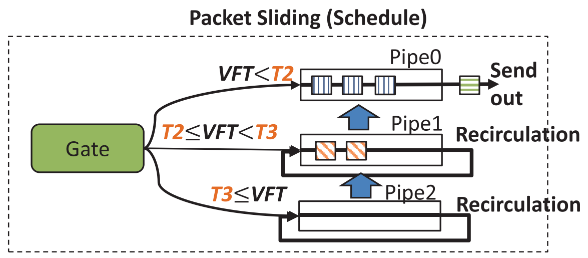
D. Basis of Tofino ASIC

Programmable switch registers: Modern programmable switches, such as the Barefoot Tofino, incorporate stateful storage elements known as registers. These registers function as on-chip memory, enabling the persistent storage of flow-related state information (e.g., counters, flags, or timestamps) across packet processing cycles. Accessible at line rate within the match-action pipeline, registers allow the data plane to perform complex, per-packet computations beyond simple stateless lookups.

[5]This includes 13 bytes for the five-tuple and an additional 24 bytes for the token bucket parameters, specifically an 8-byte CIR and an 8-byte PIR to convert the time intervals into tokens, and an 8-byte variable to record the current amount of accumulated tokens.



(a) Packet grouping (sort and schedule).



(b) Packet sliding (schedule).

Fig. 3. The key idea of packet grouping and packet sliding to achieve the sorting and scheduling..

Cross-pipeline constraints: The Tofino architecture is organized into multiple parallel processing pipelines. Although state information can be shared between pipelines by embedding it in packet headers, strict limitations govern the permissible transit paths. Specifically, a packet can directly move only from the Ingress stage of one pipeline to the Egress stage of another. If a packet reaches the Egress of a pipeline and still requires further processing in a different pipeline, it must first loop back to the Ingress of the same pipeline before it can be forwarded to the target pipeline's Egress. This constraint fundamentally limits the available mechanisms for cross-pipeline state sharing and has significant implications for the design of packet processing and scheduling algorithms.

E. Insight and Key Idea

Instead of sorting and scheduling through modifying the packet order in a *single* pipeline buffer, which is *unprogrammable* in Tofino ASIC, we propose to implement a *group-level* sorting in the *programmable* control component by arranging packets into *multiple* pipeline buffers. At this core, DRAGONKING renovates the workflow of the rate limiter from the *packet-level* sorting to the *group-level* sorting while maintaining high accuracy by carefully managing the number of packets in each pipeline buffer. Specifically, there are two core components (i.e., the packet grouping and the packet sliding) to achieve the above *group-level* sorting.

Packet grouping. As depicted in Figure 3-(a), after the VFT of each incoming packet is calculated based on WF^{2}Q+, the gate module directs packets into different pipelines according to their VFT range (e.g., achieving the sorting). Pipeline zero functions as the head queue, dispatching packets whose VST aligns with the current transmission time (e.g., achieving the scheduling). For instance, as shown in Figure 3-(a), two packets with a VFT greater than $T2$ are buffered in pipeline two. Meanwhile, one packet remains in pipeline one because

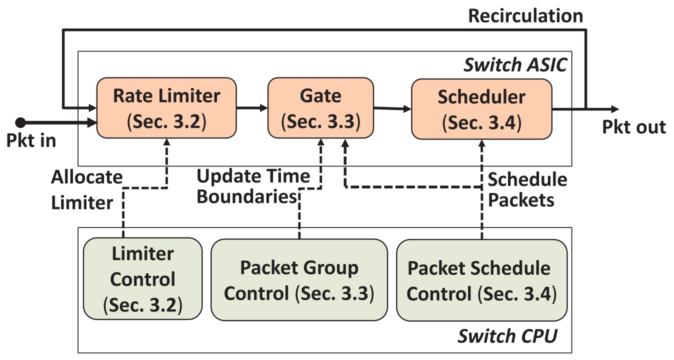


Fig. 4. The overview of DRAGONKING architecture.

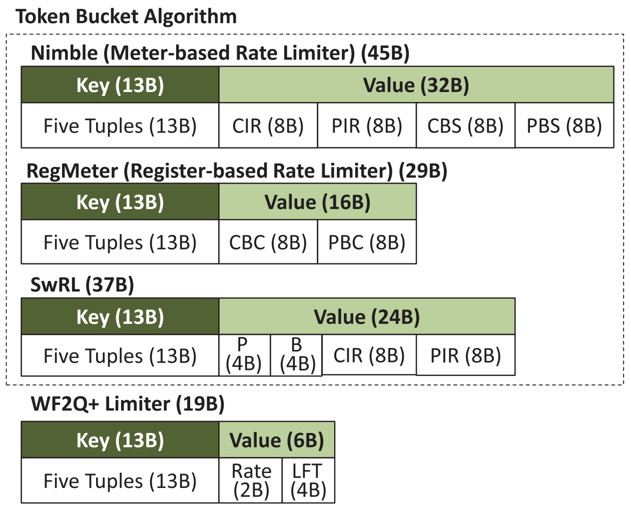


Fig. 5. Rate limiter table design comparison. For the token-bucket-based limiter table, four parameters are required to record the status of buckets, which occupy large memory. By enabling WF^{2}Q+ on programmable switches, DRAGONKING can significantly reduce the entry size of the limiter table by 50%+, achieving $2\times$ scalability.

its VST has not yet been reached, while another eligible packet is transmitted.

Packet sliding. DRAGONKING actively manages the timing of packet transmissions to prevent missed deadlines. When specific conditions are met (such as a packet nearing its transmission deadline), the timing boundaries for the VFT in each pipeline are updated. Consequently, packets are systematically transferred from later pipelines (i.e., pipelines one to three) to the earlier pipeline (i.e., pipeline zero) that is ready for dispatch. This process facilitates effective scheduling, as illustrated in Figure 3-(b).

III. DRAGONKING DESIGN

Based on the above insight analysis, we design DRAGONKING, a scalable, high-throughput and accurate rate limiter on programmable switches for cloud networks. In this section, we detail the design of DRAGONKING.

A. Overview

Scalability. Scaling rate limiting on programmable switches involves two critical challenges: reducing the per-entry memory footprint and optimizing table allocation across pipeline stages. In traditional token-bucket-based schemes [19], [20], a *meter* entry must hold four parameters (for example, CIR, CBS, PIR, and PBS), consuming 45 bytes per entry. In contrast, our WF^{2}Q+-based approach distills the necessary state into just two parameters (i.e., the limited rate and the last finish time), thereby reducing the entry length of the *rate_limiter_table* to 19 bytes. This total consists of a 13-byte key (the 5-tuple) combined with a 6-byte value representing the two state parameters, which saves over 57% of the memory cost and nearly doubles the number of rate limiter entries on the same hardware. The entries are updated by the Limiter Control module as shown in Figure 4. Furthermore, we carefully position the key tables (i.e., *rate_limiter_table*, *gate_table*, and *scheduler_table*) in the programmable switch's pipeline to reduce wasted memory caused by table dependencies, ensuring high SRAM/MAPRAM utilization and minimal bandwidth overhead (details are provided in Section III-B).

High throughput. In DRAGONKING, the switch ASIC's *gate_table* performs group-level packet sorting while the CPU-based Packet Group Control module dynamically adjusts time windows across pipelines (as shown in Figure 4). By recording the expected transmission times of packets instead of relying on extra control packets, DRAGONKING eliminates unnecessary overhead. At the same time, its bandwidth-aware mechanism continuously monitors pipeline utilization and intelligently redistributes packets to balance load and prevent buffer overflows. These strategies enable DRAGONKING to boost throughput by at least 43% over the existing method [21], as detailed in Section III-C.

Accurate. The *scheduler_table* is designed to determine the actions for received packets, for example, whether to send them out, recirculate them, or drop them. It is crucial to carefully design the scheduling strategy to maintain high rate limiting accuracy. By default, pipeline zero, referred to as the *front-pipeline*, acts as the external physical port of the switch, forwarding packets that are approaching their expected transmission time. Packets directed to the remaining pipelines (termed *backend-pipelines*) are set for recirculation. To achieve high accuracy, packets buffered in the *backend-pipelines* that are nearing their scheduled transmission time (VST) must be transferred to the *front-pipeline* in a timely manner. As detailed in Section III-D, we formulate this scheduling task as a queuing problem to minimize the discrepancy between actual and expected transmission times. By carefully adjusting the duration T_0 of the *front-pipeline* to account for the VST, we achieve a rate limiting accuracy of 99% (as evaluated in Section V-C).

B. Scalability: Efficient Table Design

1) Table Layout: Figure 5 compares the *rate_limiter_table* layout between the token-bucket-based approach and the WF^{2}Q+-based approach implemented in DRAGONKING. Existing works utilize the function *meter* (provided by the switch ASIC Tofino [7]) to achieve the token-bucket-based rate limiting, which supports the Two-Rate-Three-Color-Marker

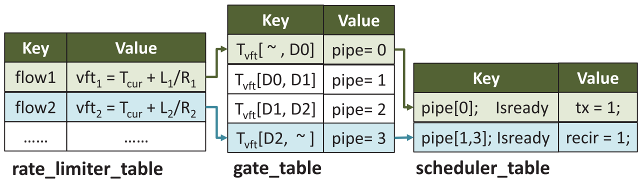


Fig. 6. Relationship between three tables, i.e., rate_limiter_table, gate_table and scheduler_table.

algorithm [34]. However, *meter* consumes 32 bytes to record the status of buckets (i.e., CIR, CBS, PIR, and PBS) following the standard definition [34], resulting in 45 bytes for each limiting entry. Note that for RegMeter [21] and SwRL [22], we let all of the flows share the same CBS/PBS to save the memory. The CBC (Committed Bucket Counter) and PBC (Peak Bucket Counter) are the two counters to record the current number of accumulated tokens in RegMeter [21].

As a comparison, by enabling WF^{2}Q+ on programmable switches, DRAGONKING involves only two parameters in the value field, i.e., the limited rate (2 bytes) and the last finish time (LFT) of the received flow (4 bytes). The LFT indicates the latest VFT of the flow, and the transmission time of subsequent packets should start from the LFT, such that the rate of the flow is limited. The VFT for each packet is calculated by dividing the packet length by the limited rate (note that the Tofino ASIC does not support division operations directly; we accomplish this through a division results matching table, as detailed in the implementation Section IV). The final entry length for the rate_limiter_table has been optimized to 19 bytes, which is 57% shorter than the token-bucket-based approach, leading to 1.9$\times$ improvement of the scalability as evaluated in Section V-A.

2) Efficient Table Allocation: As depicted in Figure 4, DRAGONKING incorporates three key tables to implement the WF^{2}Q+ algorithm, with the logical relationships shown in Figure 6. In the *rate_limiter_table*, when the five-tuple of the received packet matches a limiter entry, its VFT vft_i is updated by taking the maximum of LFT and the current time T_{cur}, then adding the quotient of the packet length L and the limited rate R. Subsequently, in the *gate_table*, the corresponding pipeline is determined based on the matched VFT range. For instance, flow1, which matches a time range where the vft is lower than D_0, will be sent to pipeline zero; while flow2, matching a time range where the vft is greater than $D-2$, will be directed to pipeline three. Finally, the *scheduler_table* determines the transmission behaviors (i.e., send out, recirculation, or drop) based on the targeted pipeline and whether the VST of the packet is reached at the current time.

As a straightforward method, we can put all tables (i.e., *rate_limiter_table*, *gate_table*, *scheduler_table*) in the ingress (as shown in Figure 7(a)). However, these tables have a data dependency (i.e., the action executed relies on the matching result of the previous entry), leading to a separate allocation of the three tables into different stages [4], it results in low SRAM utilizations ($<$10%), harmfully lowering the memory efficiency. As a candidate, we can move the two smaller tables to the egress, as shown in Figure 7(b). Due to the cross-pipeline constraints in Tofino [7], packets that need to be cached on back-end pipelines must be recirculated to the ingress of the front-pipeline.

Considering the above justifications, we propose to place the *gate_table* and the *scheduler_table* to the backend-pipelines' egress, while only reserving the *scheduler_table* in the front-pipeline to determine the packets' transmission behaviors (as shown in Figure 7(c)). Packets are first redirected to the backend-pipelines without consuming front-pipeline bandwidth, which is set to be associated with the switch ports to receive traffic from other servers. In this way, we achieve high scalability without consuming extra bandwidth.

C. High-Throughput: Bandwidth-Aware Packet Redistribution

To achieve high throughput in a rate limiter, it is essential to minimize the use of extra control packets that consume forwarding capacity. Carefully monitoring the packet buffer status across all pipelines is also crucial to prevent buffer overflow, which can lead to packet drops and throughput degradation [23].

Eliminate control traffic overhead: by implementing WF^{2}Q+ on the Tofino ASIC, DRAGONKING eliminates the overhead of control packets used in the token bucket-based approach for adding tokens, thereby preserving the gateway's throughput.

Buffer overflow management: to address potential buffer overflow in pipelines, we propose a bandwidth-aware approach that reallocates packets among pipelines to even out buffer fullness. DRAGONKING first periodically collects the bandwidth of all pipelines and then redistributes packets accordingly, e.g., reducing the time duration in pipelines expected to have high traffic volume and increasing them in pipelines with anticipated lower traffic loads. This strategy helps balance the bandwidth utilization across pipelines and reduces the likelihood of overflow. Packet redistribution is facilitated by adjusting the time boundary in the *gate_table*.

Figure 8 illustrates the process of updating the time boundary in the *gate_table*. When the time-boundary updating timer T_b expires, the control plane on the switch CPU retrieves the current bandwidth B_i from the counters in each pipeline, as recorded in the *statis_table*. The timer T_b should be carefully set to timely detect potential buffer overflows. Because we need to update the entries of *gate_table* to perform the packet redistribution, therefore, T_b is highly restricted by the entry update rate. The table entry update rate can achieve to approximately 2M/s [6], indicating a minimal timer with 5μs. According to existing studies [38], [39], over 90% of inter-burst periods last more than 50μs, therefore, 5μs is sufficiently short to monitor the potential traffic bursts. Given the collected bandwidth B_i, DRAGONKING then estimates the bandwidth variances V_i. The packet redistribution occurs only when a pipeline's bandwidth variance surpasses a threshold to prevent frequent packet movements between pipelines. Finally, the control plane updates the time boundary in the *gate_table* accordingly.

Algorithm 1 outlines the key procedure for bandwidth-aware packet redistribution. Let B_0, B_1, B_2, and B_3 denote

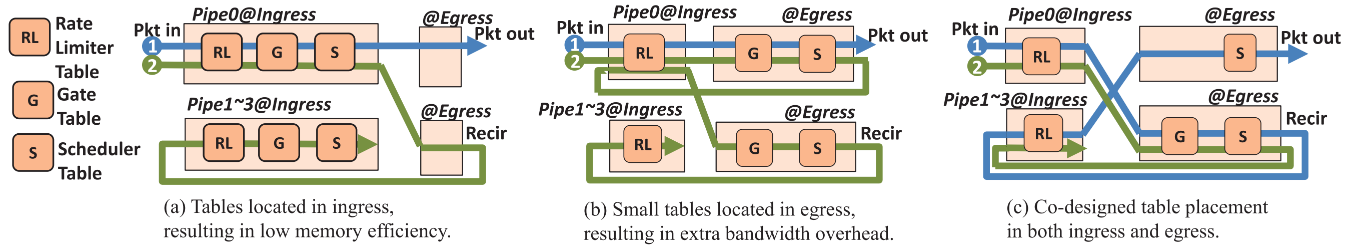

Rate Limiter Table
Gate Table
Scheduler Table
Pkt in
Pipe0@Ingress
@Egress
Pkt out
Pipe1~3@Ingress
Recir
(a) Tables located in ingress, resulting in low memory efficiency.
(b) Small tables located in egress, resulting in extra bandwidth overhead.
(c) Co-designed table placement in both ingress and egress.

Fig. 7. Table allocation strategy comparisons. DRAGONKING utilizes the strategy (c) of ingress&egress co-designed table placement approach to achieve a high memory efficiency and low bandwidth overhead. The packet flow for two typical scenarios is illustrated using the three strategies above: 1) packets are directly forwarded through the front-pipeline zero; 2) packets are redirected to backend-pipelines for temporary caching.

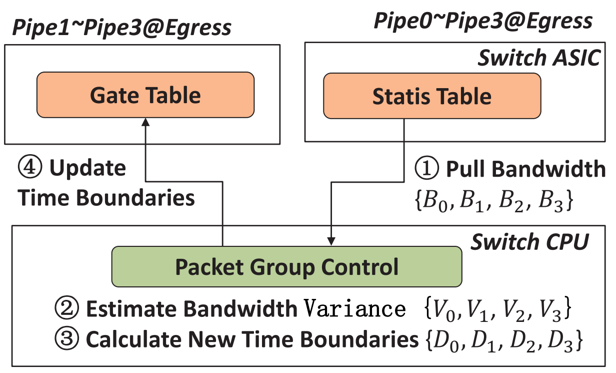

Pipe1~Pipe3@Egress
Pipe0~Pipe3@Egress
Switch ASIC
Gate Table
Statis Table
④ Update Time Boundaries
① Pull Bandwidth $\{B_0, B_1, B_2, B_3\}$
Switch CPU
Packet Group Control
② Estimate Bandwidth Variance $\{V_0, V_1, V_2, V_3\}$
③ Calculate New Time Boundaries $\{D_0, D_1, D_2, D_3\}$

Fig. 8. The procedure of adjusting packet groups through updating time boundaries.

Algorithm 1 Bandwidth-Aware Packet Redistribution

```
Input: $B_0$, $B_1$, $B_2$, $B_3$
Output: $D_0$, $D_1$, $D_2$, $D_3$
1:  is_update_need ← False
2:  [$V_0$, $V_1$, $V_2$, $V_3$] ← est_variance_of_each_pipeline($B_0$, $B_1$, $B_2$, $B_3$)
3:  if $V_i \geq \delta_v, \forall i \in \{0,1,2,3\}$ then
4:  is_update_need ← True
5:  end if
6:  if is_update_need then
7:      $total \leftarrow \sum_{i=0}^{N-1}(1-\hat{B}_i)$
8:      $cumulative \leftarrow 0$
9:      for $i = 0$ to $N-1$ do
10:         $T_i \leftarrow T \cdot \frac{1-\hat{B}_i}{total}$
11:         $cumulative \leftarrow cumulative + T_i$
12:         $D_i \leftarrow T_{beg} + cumulative$
13:     end for
14:     $T_{beg} \leftarrow T_{beg} + T_0$
15: end if
```

the collected bandwidth for each pipeline, let N denote the number of pipelines (which is four in Tofino [4]), and let T denote the total time duration for all pipelines.

For the collected bandwidth B_i, we first perform a normalization for conveniently calculation, i.e., $\hat{B}_i = B_i/B_{\text{cap}}, \hat{B}_i \in [0,1]$. Where B_{cap} is the maximum bandwidth for each pipeline. $\hat{B}_i$ is the normalized bandwidth utilization ratio for the pipeline i. When the variance in bandwidth utilization (i.e., V_0, V_1, V_2, V_3) among the four monitored pipelines exceeds the specified threshold δ_v (line 2), the algorithm is labeled to perform the bandwidth-aware packet redistribution (line 3 to 4). This process evenly redistributes data among the pipelines, reducing the risk of overflow in any single pipeline.

T is the difference between the minimal VFT and the max VFT, which can be directly fetched from the switch ASIC. Initially, T is set to four times T_b, indicating an equal distribution across all pipelines. Initially, T is set to $4 \times 5\mu$s, with each pipeline assigned a 5μs time window to ensure equal distribution. We choose this value to keep the number of packets in the front-pipeline as low as possible while balancing all pipelines. The front-pipeline holds packets closest to their scheduled send time. If too many packets accumulate in the front-pipeline, the switch can only forward them in FIFO order because of hardware constraints. This increases the likelihood of missed deadlines and reduces rate limiting accuracy. Thus, this initial setting improves rate limiting precision and achieves balanced load distribution.

Then the time duration T_i for each pipeline i is inferred as follows:

$$T_i = T\frac{1-\hat{B}_i}{\sum_{j=0}^{N} 1-\hat{B}_j}, \text{if } 0 \leq i < \text{N}-1, \tag{1}$$

Based on the above inferred time duration, the time-boundary is computed as detailed in lines 6 to 14 of Algorithm 1. T_{beg} serves as an earliest virtual start time (VST) point of all packets cached in the front-pipeline. Initially, T_{beg} is set to the current time T_{cur}, and is updated to $T_{beg} + T_0$ following the packet slide time T_{slide}. Since the packet slide action relies on entry updates, we set T_{slide} to the minimal update period of 5μs to ensure that cached packets do not experience excessive delays before being sent. The packet redistribution will be performed on-demand after the packet slide. D_i and D_{i+1} denote the start and end time points of the corresponding time duration for pipeline i, respectively. These values are employed as keys in a range-type *gate_table* to specify the VFT time range required for matching packets (line 12).

D. Accurate: Virtual-Start-Time-Aware Scheduling

To achieve high accuracy, it is crucial to let the permitted packets (e.g., the current time is larger or equal to the VST of packets) be transmitted out timely. Especially to avoid leaving packets cached in the backend-pipelines for an extended

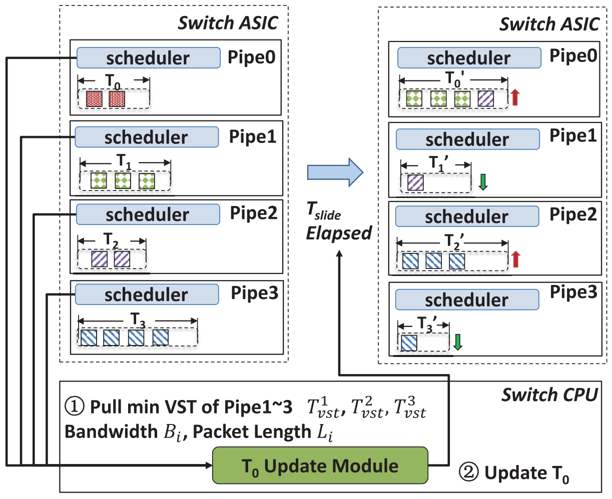


Fig. 9. The procedure of adaptively scheduling packets based on the observed minimal VST of each pipeline, the bandwidth utilization ratio and the packet length.

period. A straightforward approach is to enlarge the time duration T_0 of the front-pipeline, such that the transmission time of packets in the backend-pipelines can be expedited. However, when too many packets are cached in the front-pipeline, it may degrade the rate-limiting accuracy due to the potentially increased queuing delay in the front-pipeline. To determine an appropriate time duration T_0, we formulate the above problem using a queuing model.

1) Problem Formulation: Given the probability that packets are distributed into the front-pipeline $P_0(T_0)$, the expected packet forwarding delay in the front-pipeline W_0, and the average waiting delay d when packets are distributed into the backend-pipelines, then the optimization goal can be formulated as minimizing the average queuing delay A:

$$\begin{aligned} &\min \quad A = P_0(T_0) \cdot W_0 + (1 - P_0(T_0)) \cdot d \\ &\text{subject to} \quad B_i < B_{\text{cap}}, \forall i \in \{0, 1, 2, 3\}. \end{aligned} \tag{2}$$

$P_0(T_0)$ is estimated as the ratio of T_{slide} to the total time duration T, i.e., $P_0(T_0) = \frac{T_{slide}}{T}$. The value of W_0 is approximated using the classic Pollaczek–Khinchine equation [40]. Given the queuing utilization ratio ρ, W_0 is estimated as $\alpha \cdot \frac{\rho}{1-\rho}$. The utilization ratio ρ is calculated as the product of the arrival rate λ and the average service time s. Both λ and s can be obtained online from the switch ASIC. The parameter α represents the coefficient of variation for service time. This parameter is fitted in our simulation environment using real-world data center traffic traces [38].

2) Parameter Acquisition: In addition to T_0 and the coefficient of variation α, all other parameters can be obtained from the statis_table of the switch ASIC, as shown in Figure 9. The arrival rate λ is calculated as the ratio of the front-pipeline's bandwidth B_0 to the total bandwidth of all four pipelines. This is expressed as $B_0 / \sum_{j=0}^{N} B_j$. The average service time s is computed as the ratio of the average packet length l_{ave} to the accumulated limiting rate r_{accu}. The average packet length l_{ave} is obtained directly from the counter function of the Tofino ASIC, with the PACKETS_AND_BYTES option enabled [7]. The accumulated limiting rate r_{accu} is the sum of the rates of all limiters stored in the control plane. The value of d is inferred from the minimal VST of the backend pipelines, $T^{min}_{vst} = \min(T^i_{vst}, \forall i \in \{1, 2, 3\})$. The details are shown below:

$$T_d = T_{cur} + T_{slide} - T^{min}_{vst}, \tag{3}$$

$$d = \begin{cases} T_d & \text{if } T_d > 0, \\ 0 & \text{else.} \end{cases} \tag{4}$$

3) Packet Scheduling Strategy: Once all the above parameters are obtained from the switch ASIC, the minimization goal in equation (2) can be simplified as follows while the constraints stay the same.

$$\min \quad A' = \beta \cdot T_0 + d \tag{5}$$

Here, β is defined as the difference between W_0 and d. When $\beta > 0$, it indicates that too many packets are cached in the front-pipeline, and the time duration T_0 should be reduced. Conversely, when $\beta \le 0$, there may be more potential deadline-missing packets cached in the backend-pipelines. In this case, T_0 should be increased to allow more packets to move forward to the front pipeline. To prevent excessive jitter of T_0, updates are made only when $|\beta|$ exceeds a certain threshold h. We also use $|\beta|$ to control the step size δ when updating T_0. This approach helps to better adapt to the current scheduling strategy. Both the step size δ and the threshold h are estimated empirically. Specifically, T_0 is updated as follows:

$$T'_0 = \begin{cases} T_0 + \delta \cdot \beta & \text{if } |\beta| > h, \\ T_0 & \text{else.} \end{cases} \tag{6}$$

Therefore, in the Algorithm 1, we replace estimation of the time duration T_0 with T'_0 with the above approach. While the rest time duration still follows the procedure of Algorithm 1 in lines 6 to 14.

IV. Implementation

In this section, we explain how the design principles from Section III are implemented on the Barefoot Tofino switch. Our approach converts the efficient table design, multi-pipeline grouping with a sliding mechanism, and VST-aware scheduling algorithm into a P4 data plane program and corresponding CPU control plane logic. The P4 program organizes table entries and manages metadata for multi-pipeline sorting and scheduling, while the CPU control plane updates rate limiter entries, coordinates packet group control, and executes VST-aware scheduling.

We built a P4 prototype of DragonKing and compiled it on a programmable switch ASIC. We also integrate DragonKing into a production programmable gateway ConWeave [37] to evaluate the end-to-end performance. ConWeave [37] implements a typical load balance gateway for the RDMA networks in approximately 3,500+ lines of P4 code. We added about 500 lines of P4 code to implement all tables and metadata required by DragonKing.

Data Plane on Switch ASIC. We implement the *rate_limiter_table* as an exact-match table that stores the rate limit parameter (*Rate*) and the last finish time (*LFT*) for each flow. Since Tofino does not support division, we calculate each packet's expected sending time in the control plane using a small table, *pkt_epased_time_table*, whose key consists of *Rate* and packet length (*Len*). The *gate_table* is implemented as a range-match table with the timing boundary as its key, while the *scheduler_table* is an exact-match table that determines whether a packet should be recirculated or transmitted. Multiple pipelines can be employed for packet buffering; in the Tofino architecture, packet headers are processed in the pipeline and payloads are stored in an internal buffer. During recirculation, the payload is not replicated but is referenced by a pointer that identifies its location in the buffer, and when the scheduled transmission time arrives, the pipeline retrieves the packet header and its associated payload from the buffer, merges them, and transmits the complete packet.

To ensure atomic updates on the data plane, our system relies on the Tofino chip that guarantees the atomicity of single table entry updates. For example, updates to the *rate_limiter_table* and *scheduler_table* are performed without inconsistencies. In cases where the slide operation triggers simultaneous updates of multiple range table entries in the *gate_table*, we update entries sequentially from the lower time interval (e.g., the entry covering [$\sim$, D_0]) to the higher time interval (e.g., the entry covering [D_2, $\sim$]). Newly updated entries are assigned a higher `MATCH_PRIORITY` value so that when time intervals overlap, the most recent entry is matched. This procedure guarantees that every packet finds a matching entry and is not dropped due to a mismatch.

Control Plane on Switch CPU. We implement a control plane in the switch software for the Limiter Control, the Packet Group Control and the Packet Scheduler Control. The Limiter Control module targets the add/del/modify/query of the *rate_limiter_table*. The Packet Group Control module targets at adaptively balancing the bandwidth utilization across backend-pipelines, while the Packet Schedule Control model targets at updating the front-pipeline window T_0 to achieve high limiting accuracy. We implement the entry update action using the Runtime API rather than gRPC, to achieve an approximately 2M/s table entries update rate [6].

The control plane employs a multi-threaded design that avoids resource contention by not sharing data structures among threads. The Limiter Control module runs in a dedicated thread to exclusively process user rate limiting requests and, for each specified five-tuple flow, installs the corresponding rate limiting entry into the data plane's *rate_limiter_table*. This module is operationally isolated from the other functionalities. The Packet Group Control and Packet Schedule Control modules execute sequentially within a single thread. At the end of each bandwidth monitoring window, the Packet Schedule Control module first performs the VST-aware T0 window optimization, and the Packet Group Control module subsequently determines whether to perform backend pipeline bandwidth load balancing. If required, the update is executed according to Algorithm 1; otherwise the T0 window size is adjusted, and the *gate_table* is updated accordingly.

Parameter α fitness. In this study, we adopt the previously derived formula for the expected packet forwarding delay in the front-pipeline: $W_0 = \alpha \cdot \frac{\rho}{1-\rho}$, where the utilization ρ is defined as $\rho = \lambda \cdot s$ and the average service time is $s = \frac{T_{ast}}{P_{\text{out}}^{ast}}$. Here, T_{ast} denotes the period during which the control plane periodically retrieves the number of packets forwarded (P_{out}^{ast}) from the data plane, and λ is determined by the number of packets received (P_{in}^{ast}) in that period. To capture these periodic metrics, we deploy two statistical tables at the front-pipeline: `Ingress_statis_table` at the Ingress and `Egress_statis_table` at the Egress. Both tables use the designated port ID as the key and a DirectCounter (configured in PACKETS_AND_BYTES mode) as the value to concurrently count packet numbers and forwarded bytes.

To measure the actual per-packet delay W_0 in the switch, we further deploy two timestamp recording tables at the front-pipeline: `Ingress_time_table` and `Egress_time_table`. Their keys are composed of the L4 source port (`4l.srcport`) and the sequence number (`4l.seq_no`), which together form a unique identifier for each packet. These tables enable us to track the ingress and egress timestamps for every packet, thereby obtaining an accurate ground truth of W_0 in a controlled environment. Note that while the statistical tables are used in DragonKing for collecting port-wise bandwidth and packet rate data in real deployments, the timestamp tables are solely for offline data collection to acquire true delay measurements for fitting α.

By replaying real data center traffic traces from [38], we collect numerous offline samples $\{W_0[i], \rho_i\}$ $(i = 1, 2, \ldots, N)$. We then define $X_i = \frac{\rho_i}{1-\rho_i}$ so that the model becomes linear ($W_0[i] = \alpha \cdot X_i$). A least-squares fit minimizes $E(\alpha) = \sum_{i=1}^{N} (W_0[i] - \alpha \cdot X_i)^2$, yielding $\alpha = \frac{\sum_{i=1}^{N} X_i W_0[i]}{\sum_{i=1}^{N} X_i^2}$. We deduce α using nine out of ten samples, with the estimated value being 0.001. The remaining samples are used to evaluate the accuracy of α in estimating W_0, and the results show that the predicted W_0 has a relative error below 5%, confirming the robustness and accuracy of our approach.

V. Evaluation

We evaluate DragonKing using testbed experiments, and compare it with three most related works that implement token bucket-based rate limiting on the Barefoot Tofino programmable switch that equips with switch ASIC Tofino T1 [7]: Nimble [19], RegMeter [21], and SwRL [22].

Testbed Deployment. To evaluate DragonKing's performance, we established a testbed with a high-performance, multi-functional hardware traffic generator, Spirent testers [41], directly connected to our system. This setup enables precise measurements of throughput, latency, and packet loss.

Parameter Settings. *Nimble* [19] utilizes meter to implement a token bucket algorithm. It composes the rate limiter table based on the five-tuple information and the four inherent parameters (*i.e*, CB, PB, CIR, PIR). Therefore, each entry occupies 45 bytes. *RegMeter* [21] utilizes the register to implement the token bucket algorithm. The key field is the same with MeRL, while the value field is composed with 16 bytes total to record the current accumulated tokens of the two buckets (CB and PB). The length of token-addition packet is set to the minimal TCP packet as 55 bytes as described in [21]. For *SwRL* [22], there are total 37 bytes needed for each rate limiter entry as stated in [22].

Bursty Settings. Existing measurement studies [38], [39] show that approximately 99% of traffic burstiness occurs within the range of 50μs to 0.5ms, with the duration of 1ms to 20ms, with bursty rates varying between 50% and 100% of link utilization (e.g., 50Gbps to 100Gbps). We generate the bursty traffic following the traffic patterns described in [38]. The CBS/PBS parameters must be carefully configured to balance accommodating bursts without overloading servers or causing unnecessary packet drops. Then given a maximum 100Gbps link, we set the CBS as 10Mb. The PB of RegMeter is configured to match the CB, also set to 10Mb, as described in [21]. For Nimble [19] and SwRL [22], which do not specify PB settings, we configure PB as 1.6$\times$ the CB, based on typical production settings [42], i.e., 16Mb. DRAGONKING mainly utilizes the backend-pipelines to cache bursty traffic, with a capacity of around 120Mb [7]. In addition, we vary the time interval (i.e., from 500μs to 50μs) between the occurrences of burst traffic to create scenarios with different degrees of burstiness. This setup is utilized to evaluate the ablation study of the core component of bandwidth-aware packet redistribution model of DRAGONKING (as detailed in Section V-G).

Production Environment Settings. We have integrated DRAGONKING into a production gateway, ConWeave [37], to evaluate its end-to-end performance. It is a load-balancing gateway designed to support high-performance RDMA transmissions. We deployed ConWeave [37] on a Barefoot switch equipped with Tofino1, with the reorder feature disabled. We employed two servers equipped with Mellanox CX5 RNICs, which are connected via 100Gbps links to the programmable switch running ConWeave [37].

MicroBenchmarks. To evaluate the micro-behaviors of DRAGONKING, (i.e., bandwidth utilization and packet scheduling performance), we utilize the mirroring capability of Tofino [7] to replicate the normalized utilization data to an external server for analysis. Packet scheduling timestamps are embedded in the IP header's option field and redirected to the external server.

Results reveal that:

- DRAGONKING achieves high scalability: it supports 2 millions of rate limiters, outperforming the Nimble [19], RegMeter [21] and SwRL [22] by 1.9$\times$, 1.6$\times$ and 1.9$\times$ respectively (Section V-A).
- DRAGONKING achieves high throughput, 15.7%, 44.5%, and 14.1% compared with Nimble [19], RegMeter [21], and SwRL [22] under bursty scenarios (Section V-B).
- DRAGONKING can consistently achieve a high limiting accuracy of over 99% when limited rates vary from 10Gbps to 100Gbps. Specifically, with a limited rate of 100Gbps, DRAGONKING can improve the accuracy by 75.6% compared with RegMeter [21] (Section V-C).
- In the production scenarios of integrating DRAGONKING with a typical load balance gateway ConWeave, DRAGONKING can improve goodput by 106.7%, 434.4%, and 84.5% compared to Nimble [19], RegMeter [21], and SwRL [22], respectively (Section V-E).

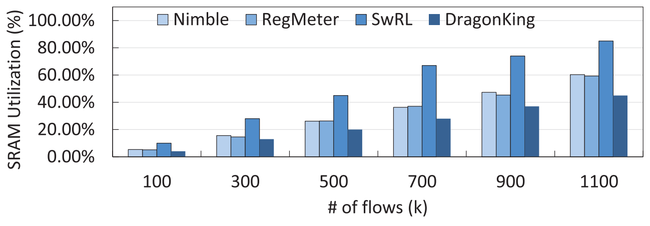


Fig. 10. Scalability. The SRAM utilization ratio.

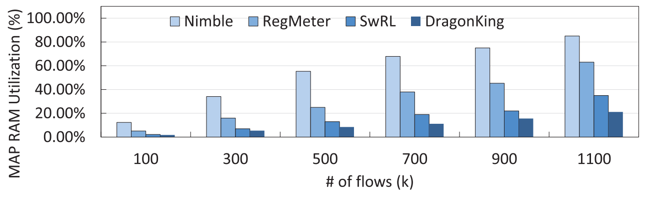


Fig. 11. Scalability. The MAP RAM utilization ratio.

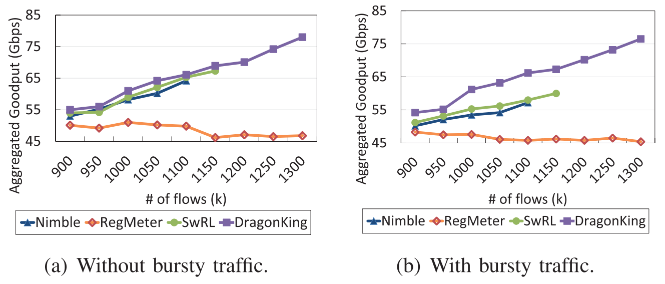
A. Scalability

Among the hardware resources, SRAM and Map RAM are the most intensively used, while the utilization of other resources remains low and does not impact scalability. As depicted in Figures 10 and 11, DRAGONKING consistently maintains the lowest memory usage as the number of flows increases from 100k to 1100k. Results show that with 1100k flows, Nimble [19], SwRL [22] and RegMeter [21] consume 1.34$\times$, 1.31$\times$, 1.88$\times$ more SRAM and 4.04$\times$, 2.97$\times$, 1.65$\times$ more MAP RAM compared with DRAGONKING.

The bottlenecks affecting scalability are related to the types of memory utilized, such as SRAM and MAP RAM. Typically, SRAM is employed to store the key and stateless parameters in the value field (e.g., the five-tuples, CIR, and PIR). While MAP RAM is used to store stateful data, including meters, registers, and statistical counters. For instance, the bottleneck resource for Nimble [19] is MAP RAM, where 71% of the bytes in each entry are stored as stateful data. It is important to note that DRAGONKING maintains low utilization levels of both SRAM and MAP RAM to achieve high scalability, particularly by reducing MAP RAM usage by 87.5% compared with Nimble [19]. In summary, with efficient resource management, DRAGONKING can support two million rate limiters on a single switch, which is 1.9$\times$, 1.6$\times$ and 1.9$\times$ greater than the capacities of Nimble [19], SwRL [22] and RegMeter [21], respectively, thus achieving the goal of scalability.

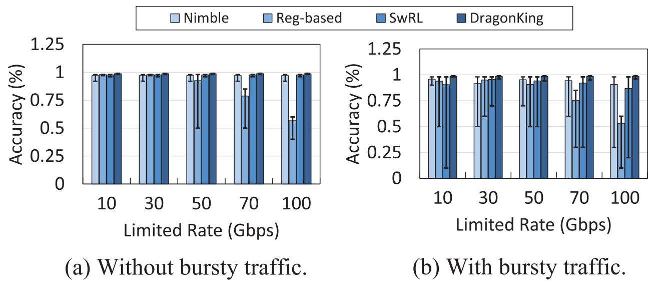
B. High Throughput

We utilize the metric of goodput to evaluate the throughput performance of DRAGONKING. Goodput is defined as the ratio of useful bytes to transmitted bytes within a given time window. Each flow is set with a fixed limiting rate of 60kbps, and all flows share a 100Gbps port link. We collect data on the aggregated goodput of active flows to examine the effectiveness of rate limiters.



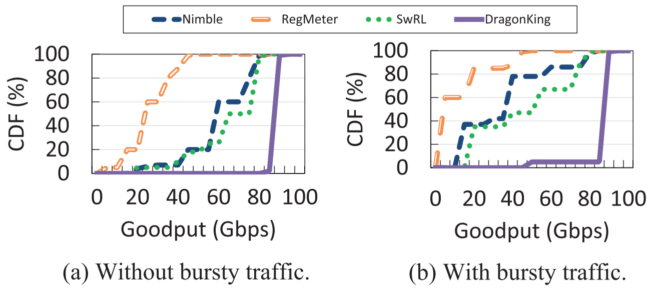
(a) Without bursty traffic. (b) With bursty traffic.

Fig. 12. Aggregated goodput. DRAGONKING consistently maintains high goodput. RegMeter performs the worst due to significant control traffic.



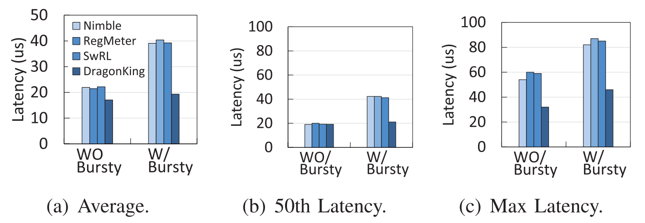
(a) Without bursty traffic. (b) With bursty traffic.

Fig. 13. Accuracy. DRAGONKING supports accurate rate-limiting, ranging from 10 Gbps to 100 Gbps.



(a) Without bursty traffic. (b) With bursty traffic.

Fig. 14. Production. Goodput distribution when rate limit=90Gbps.



(a) Average. (b) 50th Latency. (c) Max Latency.

Fig. 15. Production. Latency distribution when rate limit=90Gbps.

Figure 12 shows that DRAGONKING consistently achieves the highest aggregated goodput as the number of flows increases from 900k to 1300k. Specifically, DRAGONKING improves aggregated goodput by 15.7%, 44.5%, and 14.1% compared with Nimble [19], RegMeter [21], and SwRL [22] under bursty scenarios. Thanks to the efficient queuing mechanism, DRAGONKING can retain and forward packets during bursty traffic periods, unlike the token bucket approach. The latter drops packets when bursty traffic exceeds bucket capacity, leading to timeouts, retransmissions, and ultimately, goodput degradation. RegMeter [21] performs the worst due to significant token-adding traffic, which wastes forwarding bandwidth. Note that when the number of flows exceeds 1100k, Nimble [19] and SwRL [22] lack data due to their limited scalability, whereas DRAGONKING continues to achieve high goodput.

C. Accuracy

In this experiment, similar to the throughput testing scenario, each flow is limited to 60kbps. All flows share a single 100Gbps port for data forwarding. By varying the number of flows, we create rate limiting scenarios that span aggregate rates from 10Gbps to 100Gbps.

Figure 13 presents that DRAGONKING can consistently achieve a high limiting accuracy with 99%+ when limited rates vary from 10Gbps to 100Gbps. Note that due to extra control packets needed by RegMeter [21], given a 100Gbps port, the maximum achieved rate is around 57Gbps, therefore the accuracy of RegMeter [21] is the lowest when the limited rate is larger than 57Gbps. Therefore, with a limited rate of 100Gbps, DRAGONKING can improve the limiting accuracy by 75.6% compared with RegMeter [21]. Note that under bursty scenarios, results show that token-bucket-based approaches exhibit significant variance due to the suddenly permitted bursty traffic, while DRAGONKING maintains stable rate limiting thanks to the efficient packet caching.

D. In Production Environments

Figure 14 presents the end-to-end goodput of RDMA communication traffic. Results show that under no bursty traffic scenarios, RegMeter [21] exhibits the lowest performance due to its substantial control traffic. Under bursty traffic scenarios, all token bucket-based rate limiters experience significant goodput degradation. Results further indicate that DRAGONKING can improve goodput by 106.7%, 434.4%, and 84.5% compared to Nimble [19], RegMeter [21], and SwRL [22], respectively, in bursty conditions. This improvement occurs because when bursty traffic exceeds the bucket size, packets are dropped according to the token bucket algorithm's scheduling strategy [19], [20], [21], [22], [34].

Similarly, Figure 15 shows the distribution of end-to-end latency in RDMA applications. Results demonstrate that DRAGONKING can reduce average latency by 50.6%, 52.2%, and 50.9% compared to Nimble [19], RegMeter [21], and SwRL [22] under bursty scenarios. The default go-back-N link loss recovery mechanism in RDMA NICs [37] means that a single packet loss can lead to numerous packet retransmissions, thereby incurring significant transmission overhead and increased latency. In contrast, DRAGONKING buffers bursty packets in the pipeline, waiting for an opportune moment to transmit them. Consequently, DRAGONKING consistently maintains high goodput and low tail latency.

E. Memory Efficiency

Table I shows the hardware resources used by Nimble [19], RegMeter [21], and SwRL [22] with 700k rate limiters. In overall, DRAGONKING can save 62.3%, 47.9% and 54.4%

TABLE I
MEMORY EFFICIENCY WHEN SUPPORT 700K RATE LIMITERS BY NIMBLE, REGMETER, SWRL AND DRAGONKING

Resource	Nimble	RegMeter	SwRL	DRAGONKING
SRAM	36.34%	37.11%	67.0%	28.0%
MAP RAM	67.89%	38.0%	19.0%	11.2%
TCAM	0%	0.61%	0%	1.04%
Match Cxbar	2.57%	2.71%	4.20%	5.52%
Hash Bits	3.14%	2.83%	10.21%	3.09%
VLIW Action	1.82%	3.64%	1.75%	2.86%

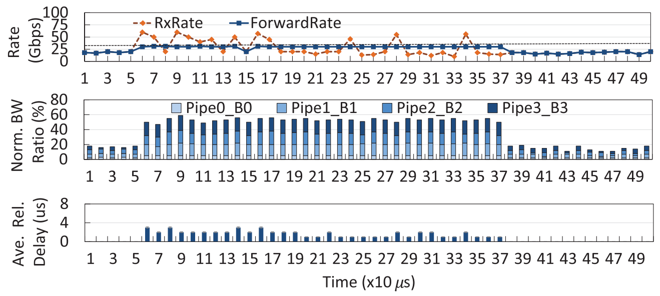
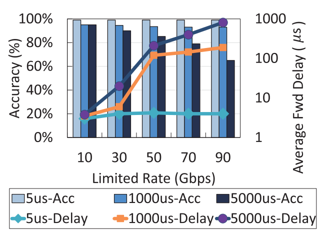
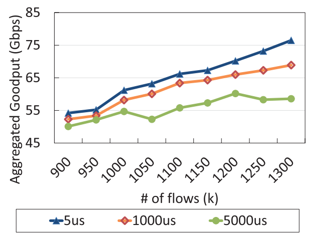


Fig. 16. Microbenchmarks. The bandwidth utilization and packet scheduling performance under dynamic traffics (rate limit=30Gbps).

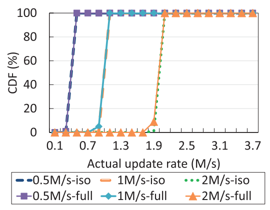


(a) Bandwidth occupancy.

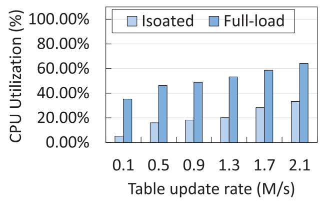


(b) Packets forwarding delay.

Fig. 17. Microbenchmarks. The sensitivity of bandwidth monitor interval T_b.



(a) Actual update rate.



(b) CPU utilization.

Fig. 18. Microbenchmarks. Table update rate and CPU utilization.

memory overhead (the average of SRAM and MAP RAM) compared with Nimble [19], RegMeter [21], and SwRL [22]. For DRAGONKING, we see that the resource usage is less than 30% for all types of hardware resources, which is a very small proportion. While the other approaches consume a large amount of overhead especially in SRAM and MAP RAM (e.g., over 60% SRAM and MAP RAM for Nimble [19] and SwRL [22] respectively).

F. Microbenchmarks

Bandwidth utilization and scheduling performance under dynamic traffic scenarios. Figure 16 illustrates microbenchmark results under dynamic traffic with a 30Gbps rate limit. The top panel shows that both received and forwarding rates consistently remain within the preset limit, demonstrating stable rate control despite significant traffic fluctuations. The middle panel confirms that the bandwidth-aware redistribution mechanism achieves balanced load allocation across the four pipelines (Pipe0–Pipe3), effectively preventing single-pipeline overloads. The bottom panel indicates that the average relative delay remains low, verifying that the Virtual Start Time (VST) based algorithm effectively regulates scheduling to keep actual transmission times close to their expected values.

In summary, by judiciously selecting the bandwidth monitoring and update interval T_b, the system is able to promptly capture traffic bursts and trigger immediate packet redistribution, ensuring that under dynamic traffic conditions the forwarding rate strictly remains within constraints while maintaining balanced load distribution across the pipelines and low scheduling delay.

Parameter T_b sensitivity. Figure 17 illustrates the impact of the parameter T_b on performance under various rate-limiting rates and concurrency scales. Results indicate that setting the bandwidth monitoring period T_b to 5μs achieves the best rate-limiting accuracy, the lowest forwarding delay, and the highest aggregated throughput. Specifically, the configuration of 5μs results in an improvement of up to 7.2% and 28.3% in rate-limiting accuracy compared to the settings of 1000μs and 5000μs, respectively. Furthermore, the aggregated throughput is enhanced by up to 11.03% and 30.5%, and the forwarding delay is reduced by 97.8% and 99.5% relative to these alternative configurations.

These performance gains can be attributed to the fact that a bandwidth monitoring period of 1000μs is near the threshold of the characteristic burst appearance period (e.g., recall that over 90% burst duration is within 1ms and 20ms [38], [39]). This proximity increases the probability of missing burst events, which result in some pipelines exceeding their forwarding bandwidth and subsequently suffering from packet loss and retransmission. Similarly, a 5000μs monitoring period leads to a higher likelihood of undetected burst events. This delay in observation prevents the timely execution of load balancing across pipelines, which in turn increases the probability of traffic overflow and packet loss and ultimately degrades performance.

Control plane table update performance. This evaluation examines scenarios with different update rates under two conditions: one where the data plane forwards no packets (0.5M/s-iso) and one where it is fully loaded with 100Gbps traffic (0.5M/s-full). Figure 18(a) shows that the Runtime API-based control plane achieves nearly identical update performance in both scenarios, demonstrating a clear separation between data and control plane operations. Figure 18(b) indicates that under full-load, overall CPU utilization remains below 60% while that for control plane entry updates stays under 35%. Moreover, under an aggregate 80Gbps bandwidth scenario, Figure 19 confirms that control plane entry updates do not adversely affect packet forwarding on the data plane.

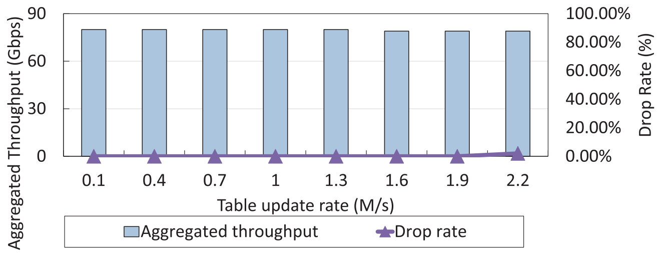
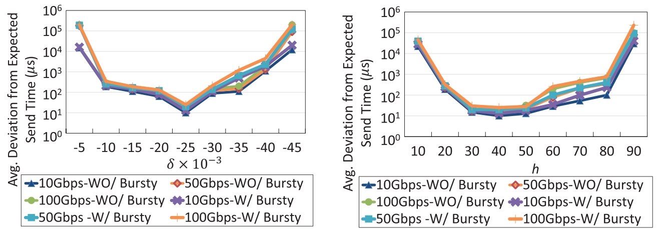


Fig. 19. Microbenchmarks. Impact of table update rate on the throughput and drop rate.



(a) Step size δ evaluation (h=40). (b) Scheduling threshold h evaluation (δ=-0.025).

Fig. 20. Microbenchmarks. Sensitivity testing of the step size δ and the scheduling threshold h.

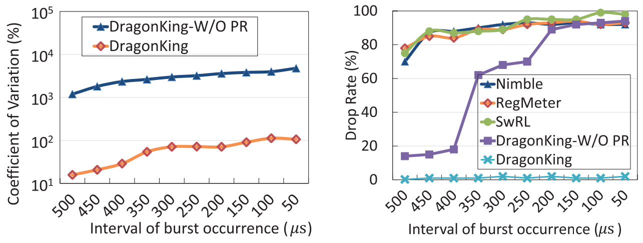
Sensitivity Testing of δ and h: As shown in Figure 20, we assess how the step size δ and the scheduling threshold h affect rate limiting accuracy under varied limiting rates and burst traffic scenarios. We use the average deviation from the expected sending time as our metric. Experimental results indicate that setting δ to -0.025 and h to 40 minimizes this deviation, achieving optimal performance. Since these parameters primarily influence the data scheduling module, our evaluation focuses on their impact on aligning actual sending times with expectations.

Figure 20(a) shows that an inappropriate step size δ degrades rate-limiting accuracy. When δ is too large (e.g., set to -0.005), the front-pipeline's T_0 adjusts too slowly, causing packets near their deadlines to remain unscheduled in the back-end pipelines or accumulate excessively in the front-pipeline. This results in scheduling delays up to 10,000 times longer than those observed with $\delta = -0.025$. Conversely, an excessively small δ (e.g., set to -0.045) makes T_0 change too rapidly, which may drive T_0 into negative (i.e., invalid) values or trigger sudden packet bursts that exceed the front-pipeline's capacity and lead to packet loss. Both extremes impair the alignment of actual transmission with the expected send times.

The scheduling threshold h also plays a critical role in the timing precision of data packets. As shown in Figure 20(b), When h is set too low (e.g., set to 10), T_0 fluctuates excessively, causing frequent preemption of front-pipeline packets by those arriving from the back-end pipelines and increasing scheduling delays. Conversely, if h is set too high (e.g., set to 90), the system fails to timely capture packets nearing their scheduling deadlines in the back-end pipelines, which also leads to increased delay.

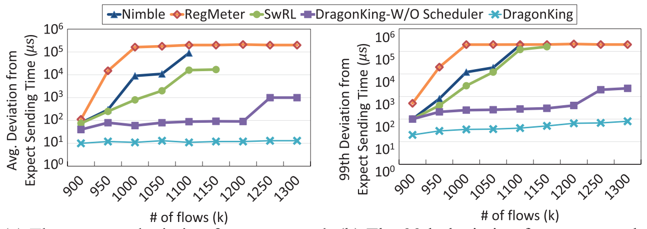
G. Ablation Studies

Figure 21 and Figure 22 show the ablation results for two core modules of DragonKing: bandwidth-aware packet redistribution and VST-aware scheduling. DragonKing-W/O PR disables packet redistribution, while DragonKing-W/O Scheduler deactivates VST-aware scheduling.



(a) Variances in bandwidth allocation. (b) Drop rate at switchs.

Fig. 21. Ablation studies on bandwidth-aware packet redistribution.



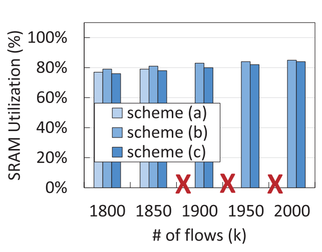
(a) The average deviation from expected sending time. (b) The 99th deviation from expected sending time.

Fig. 22. Ablation studies on VST-aware scheduling.

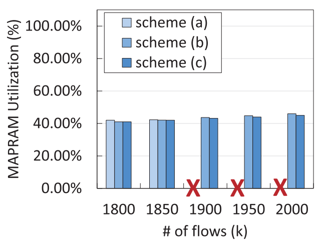
Bandwidth utilization. Figure 21(a) compares the bandwidth distribution among pipelines under various burst frequency scenarios for DragonKing and DragonKing-W/O PR. We use the coefficient of variation (CV) to measure bandwidth disparities. CV is calculated by dividing the variance of the bandwidth fractions by their mean. A lower CV indicates a more uniform distribution and lowers the risk of overloading any pipeline. This analysis involves only DragonKing and DragonKing-W/O PR because only DragonKing employs packet caching to handle burst traffic. In contrast, schemes such as Nimble [19], RegMeter [21], and SwRL [22] use token bucket rate limiting that discards packets when tokens are insufficient. As burst frequency increases, bandwidth allocation becomes more uneven. Nonetheless, DragonKing consistently achieves a more balanced distribution than DragonKing-W/O PR, with the CV reduced by up to 98.8% under extreme conditions.

Packet drops at the switch. Figure 21(b) shows that DragonKing achieves the lowest packet drop rates at the switch under various burst frequency scenarios. This performance results from DragonKing's ability to allocate bandwidth evenly among pipelines during burst traffic. In contrast, Nimble [19], RegMeter [21], and SwRL [22] experience significant packet drops due to token bucket exhaustion, with drop rates increasing as burst frequency rises.

Scheduling delay deviation. Figure 22 presents the mean and 99th deviation of packet scheduling times from their expected sending times over different concurrency levels. A larger deviation indicates that more packets are delayed, reducing the precision of rate limiting. The experiments show that DragonKing achieves the smallest scheduling delay

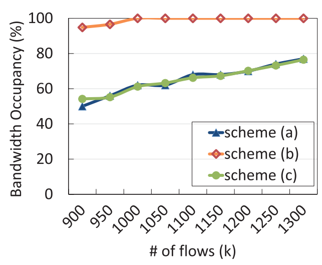


(a) SRAM.

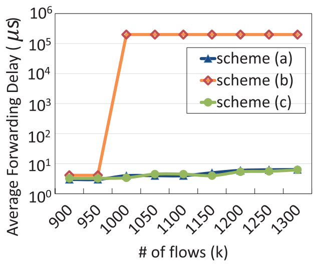


(b) MAP RAM.

Fig. 23. Table allocation scheme comparison in term of SRAM and MAP RAM.



(a) Bandwidth occupancy.



(b) Packets forwarding delay.

Fig. 24. Table allocation scheme comparison in terms of bandwidth occupancy and forwarding delay.

deviation at all concurrency levels. This result is due to DRAGONKING's packet scheduling mechanism with VST, which promptly detects packets nearing their transmission deadlines and accelerates scheduling accordingly. In contrast, Nimble [19], RegMeter [21], and SwRL [22] do not employ a packet caching strategy or a scheduling mechanism that considers the expected dispatch time. Under high concurrency or burst traffic, these schemes drop packets, leading to retransmissions that further delay dispatch. At a concurrency level of 1100K, DRAGONKING reduces the average scheduling deviation by 99.98% compared to Nimble, 99.97% compared to RegMeter, 99.93% compared to SwRL, and 87.6% compared to DRAGONKING-W/O Scheduler.

H. Table Allocation Scheme Comparisons

Figure 23 compares the SRAM and MAP RAM usage of three schemes under varying concurrency scales. Scheme (c) uses the least memory, improving scalability by 10.8% over scheme (a) and by 2.1% over scheme (b). In contrast, scheme (a) fails when the scale exceeds 1,900k because it deploys all three tables (the *rate_limiter_table*, *gate_table*, and *scheduler_table*) on the ingress. Since the *gate_table* and *scheduler_table* require few entries but must each occupy two separate stages (with less than 10% SRAM usage that cannot be reallocated), this design results in significant SRAM wastage. Meanwhile, scheme (b) reserves a few *gate_table* entries in the front pipeline, leading to slightly higher SRAM and MAP RAM usage compared to scheme (c).

Figure 24(a) evaluates the three schemes by measuring bandwidth occupancy, defined as the ratio of effective application traffic bandwidth to total port bandwidth. A lower ratio indicates less redundant traffic and higher forwarding efficiency. The occupancy ratios for scheme (c) and scheme (a) are nearly identical and are 40.1% and 41.3% lower, respectively, than that for scheme (b). This disparity is because scheme (b) recirculates traffic destined for the back-end pipelines in the front pipeline, thereby increasing bandwidth usage. When concurrency exceeds 1,000k flows, scheme (b)'s occupancy ratio reaches 100%, obstructing further traffic forwarding.

Figure 24(b) shows the packet forwarding latency on the switch under varying concurrency levels. When concurrency exceeds 1,000k, scheme (b) suffers from packet loss and retransmission due to its lower forwarding efficiency, leading to latency increases of up to 40,000 times compared to scheme (a) and scheme (c). Meanwhile, scheme (c) exhibits 5.25% higher latency than scheme (a) because traffic in the front pipeline must be rerouted to the back-end pipelines for *gate_table* matching. This minor additional delay is acceptable.

In summary, scheme (c) offers a 10% improvement in concurrent traffic capacity compared to scheme (a), a 40.1% reduction in bandwidth occupancy relative to scheme (b), and a 99.9% reduction in forwarding latency. Based on these performance benefits, scheme (c) has been selected as the final table entry allocation strategy for DRAGONKING.

VI. RELATED WORK

Rate Limiters on Programmable Switches. MeRL [20] and Swish [43] rely on the native meter function for rate limiting, a simple approach that suffers from poor scalability due to limited hardware entries. While RegMeter [21] and SwRL [22] attempt to improve scalability using registers or timestamp-based interval calculations, they introduce significant bandwidth or processing overheads. Consequently, existing switch-based solutions fail to balance performance metrics. In contrast, DRAGONKING simultaneously achieves high scalability, throughput, and accuracy.

Rate Limiters on FPGAs. FPGA-based systems like SENIC [24] and PIEO [23] leverage the accurate WF^{2}Q+ algorithm, with Tassel [11] further optimizing transmission efficiency via batching. Harmonic [44] employs backpressure for RDMA but risks congestion spread. However, these FPGA-specific designs do not address the architectural constraints of programmable switches. DRAGONKING bridges this gap by effectively implementing WF^{2}Q+ on commodity switches through a novel packet redistribution and scheduling design.

Rate Limiter Algorithms. Various algorithms aim to enhance efficiency but face hardware compatibility issues. BC-PQP [45] optimizes token buckets dynamically. PIFO [46] and vPIFO [47] offer programmable scheduling but require hardware features (e.g., programmable buffers) unavailable in standard Tofino switches. Although SP-PIFO [48] approximates priority scheduling on standard hardware, it lacks the precise timing control necessary for strict rate enforcement. Finally, CMDRL [25] focuses on distributed rate-limiting, which is orthogonal to our approach.

VII. DISCUSSION

Queue Pausing and Resuming Features in Tofino2. The ASIC Tofino2 [49] supports fundamental buffer actions like queue pausing and resuming, which ConWeave [37] leverages for packet reordering within the data plane. However, the limited number of queues capable of these actions poses scalability challenges on programmable switches. Additionally, Tofino2's higher cost compared to Tofino1 makes the latter more attractive for many cloud providers. ConWeave also notes that the pause and resume functionality primarily aids reordering when data is mostly sequential with few out-of-sequence packets. In highly disordered scenarios, such as sorting by packets' VFT in WF^{2}Q+, this method may not effectively sort packets, potentially leading to increased delays and buffering overhead.

Algorithm Generality. The proposed packet grouping and sliding approach can be adapted for use on FPGAs and servers. For FPGAs, leveraging the platform's parallel processing and custom hardware logic is crucial. Internal memory like BRAM can be organized into smaller front-buffers for ready-to-dispatch packets and larger back-end buffers for others. Adjustments to the packet sliding timer are necessary to match the platform's minimum processing times. On servers, utilizing multi-threading and multi-core processing enables concurrent handling of multiple data streams. Implementing dynamic buffer sizing through a traffic-aware packet redistribution strategy helps adapt to varying network loads, ensuring optimal performance during high traffic conditions.

The importance of switch-level rate limiting for cloud gateway deployments. Switch-level rate limiting is critical for cloud gateways due to its ability to handle Tbps-scale traffic with minimal impact on normal flows [28], a capability that server-based solutions using 100Gbps NICs struggle to match without complex state synchronization and increased jitter. While hybrid approaches combining coarse-grained switch filtering with fine-grained server control offer potential benefits for specific scenarios like DDoS defense [12], this study focuses primarily on the inherent advantages of switch-level implementation to ensure high-performance traffic management at the network edge.

Strategic decision efficiency as link bandwidth increases. DRAGONKING employs a 5μs decision window that effectively manages congestion even as link speeds scale. Extrapolating from prior studies [38], [39], a 3.2Tbps link would exhibit burst durations of $\approx 31.25\mu$s, still allowing the 5μs window at least six measurement opportunities for prompt mitigation. Given current industry upgrade cycles [50], [51], this ensures viability for the next 6–8 years. Furthermore, emerging switch architectures [4], [52] that replace PCIe with high-speed Ethernet channels promise to reduce the detection window to 200ns, extending DRAGONKING's practicality to 50Tbps networks and securing its relevance for nearly two decades.

VIII. CONCLUSION AND FUTURE WORKS

This paper introduces DRAGONKING, a novel rate-limiting system that utilizes a multi-pipeline sorting and scheduling approach to implement WF^{2}Q+. DRAGONKING enhances scalability and throughput while maintaining high precision by effectively distributing bandwidth across various pipelines, which enables prompt packet scheduling. DRAGONKING has been deployed and tested on the Barefoot Tofino switch. Results show that DRAGONKING can manage up to two million entries and achieve throughput at line rate, doubling the performance of traditional token-bucket systems. Additionally, DRAGONKING maintains over 99% accuracy.

In the future, there are multiple directions to be explored. First, we aim to expand DRAGONKING beyond active/standby deployment scenarios to include multi-active deployments. Second, we plan to enhance DRAGONKING with more sophisticated and diverse rate-limiting features, e.g., enabling preemption priorities for tenants.

REFERENCES

[1] X. Li et al., "Triton: A flexible hardware offloading architecture for accelerating apsara vSwitch in Alibaba cloud," in *Proc. ACM SIGCOMM Conf.*, Aug. 2024, pp. 750–763.

[2] M. Arumugam et al., "Bluebird: High-performance SDN for bare-metal cloud services," in *Proc. USENIX NSDI*, 2022, pp. 355–370.

[3] K. Daehyeok, N. Jacob, P. R. K. Dan, S. Vyas, and S. Srinivasan, "Redplane: Enabling fault-tolerant stateful in-switch applications," in *Proc. ACM SIGCOMM*, 2021.

[4] Y. Li, J. Gao, E. Zhai, M. Liu, K. Liu, and H. H. Liu, "CETUS: Releasing p4 programmers from the chore of trial and error compiling," in *Proc. USENIX NSDI*, 2022, pp. 371–385.

[5] Z. Chen et al., "OptimusPrime: Unleash dataplane programmability through a transformable architecture," in *Proc. ACM SIGCOMM Conf.*, Aug. 2024, pp. 904–920.

[6] C. Zeng et al., "Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing," in *Proc. USENIX NSDI*, 2022, pp. 1345–1358.

[7] (2024). *Intel Intelligent Fabric Processors*. [Online]. Available: https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors.html

[8] M. Zhang et al., "Poseidon: Mitigating volumetric DDoS attacks with programmable switches," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020, pp. 1–18.

[9] A. G. Alcoz, M. Strohmeier, V. Lenders, and L. Vanbever, "Aggregate-based congestion control for pulse-wave DDoS defense," in *Proc. ACM SIGCOMM Conf.*, Aug. 2022, pp. 693–706.

[10] X. Zeng et al., "SLA management for big data analytical applications in clouds: A taxonomy study," *ACM Comput. Surv.*, vol. 53, no. 3, pp. 1–40, Jun. 2020.

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

[12] J. C.-Y. Chou, B. Lin, S. Sen, and O. Spatscheck, "Proactive surge protection: A defense mechanism for bandwidth-based attacks," *IEEE/ACM Trans. Netw.*, vol. 17, no. 6, pp. 1711–1723, Dec. 2009.

[13] Y. Zhang et al., "Aequitas: Admission control for performance-critical RPCs in datacenters," in *Proc. ACM SIGCOMM Conf.*, Aug. 2022, pp. 1–18.

[14] W. W. Song, T. Um, S. Elnikety, M. Jeon, and B.-G. Chun, "Sponge: Fast reactive scaling for stream processing with serverless frameworks," in *Proc. USENIX ATC*, 2023, pp. 301–314.

[15] 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.

[16] D. Kim et al., "FreeFlow: Software-based virtual RDMA networking for containerized clouds," in *Proc. USENIX NSDI*, 2019, pp. 113–126.

[17] C.-Y. Hong et al., "Achieving high utilization with software-driven WAN," in *Proc. ACM SIGCOMM Conf. SIGCOMM*, Aug. 2013, pp. 15–26.

[18] 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.

[19] 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.

[20] Y. He and W. Wu, "Fully functional rate limiter design on programmable hardware switches," in *Proc. ACM SIGCOMM Conf. Posters Demos*, Aug. 2019, pp. 159–160.

[21] 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.
[22] Y. He, W. Wu, X. Wen, H. Li, and Y. Yang, “Scalable on-switch rate limiters for the cloud,” in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, May 2021, pp. 1–10.
[23] V. Shrivastav, “Fast, scalable, and programmable packet scheduler in hardware,” in *Proc. ACM Special Interest Group Data Commun.*, Aug. 2019, pp. 367–379.
[24] 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.
[25] L. Chen et al., “CMDRL: A Markovian distributed rate limiting algorithm in cloud networks,” in *Proc. 8th Asia–Pacific Workshop Netw.*, Aug. 2024, pp. 59–66.
[26] (2024). *Edgecore WEDGE100BF-32x*. [Online]. Available: https://www.edge-core.com/wp-content/uploads/2023/08/DCS800-Wedge100BF-32X-R11.pdf
[27] K. He et al., “Low latency software rate limiters for cloud networks,” in *Proc. 1st Asia–Pacific Workshop Netw.*, Aug. 2017, pp. 78–84.
[28] Y. Yang, L. He, J. Zhou, X. Shi, J. Cao, and Y. Liu, “P4runpro: Enabling runtime programmability for RMT programmable switches,” in *Proc. ACM SIGCOMM Conf.*, Aug. 2024, pp. 921–937.
[29] J. Xing et al., “Runtime programmable switches,” in *Proc. USENIX NSDI*, 2022, pp. 651–665.
[30] Y. Feng et al., “Enabling in-situ programmability in network data plane: From architecture to language,” in *Proc. USENIX NSDI*, 2022, pp. 635–649.
[31] (2024). *Custom Rate Limiting Rules*. [Online]. Available: https://www.tencentcloud.com/document/product/1145/55943
[32] (2024). *Configure Rate Limiting*. [Online]. Available: https://www.alibabacloud.com/help/en/cdn/configure-rate-limiting
[33] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, “SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching ASICs,” in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 15–28.
[34] J. Heinanen and R. Guerin, *A Two Rate Three Color Marker*, document RFC 2698, Sep. 1999. [Online]. Available: https://www.rfc-editor.org/info/rfc2698
[35] D. J. Heinanen and D. R. Guerin, *A Single Rate Three Color Marker*, document RFC 2697, Sep. 1999. [Online]. Available: https://www.rfc-editor.org/info/rfc2697
[36] D. Kim et al., “TEA: Enabling state-intensive network functions on programmable switches,” in *Proc. ACM SIGCOMM*, Jul. 2020, pp. 90–106.
[37] C. H. Song, X. Z. Khooi, R. Joshi, I. Choi, J. Li, and M. C. Chan, “Network load balancing with in-network reordering support for RDMA,” in *Proc. ACM SIGCOMM Conf.*, Sep. 2023, pp. 816–831.
[38] Z. Qiao, L. Vincent, Z. Hongyi, and K. Arvind, “High-resolution measurement of data center microbursts,” in *Proc. ACM IMC*, 2017, pp. 78–85.
[39] D. Shan, F. Ren, P. Cheng, R. Shu, and C. Guo, “Micro-burst in data centers: Observations, analysis, and mitigations,” in *Proc. IEEE 26th Int. Conf. Netw. Protocols (ICNP)*, Sep. 2018, pp. 88–98.
[40] L. Huang and T. T. Lee, “Generalized pollaczek-khinchin formula for Markov channels,” *IEEE Trans. Commun.*, vol. 61, no. 8, pp. 3530–3540, Aug. 2013.
[41] (2023). *Unlock the Potential of New Technology*. [Online]. Available: https://www.spirent.com
[42] (2024). *S7700 V200r021c00, C01 Configuration Guide - Qos*. [Online]. Available: https://support.huawei.com/enterprise/en/doc/EDOC1100213127/f6e567c8?idPath=24030814—21782164—21782186—259602655
[43] L. Zeno et al., “SwiSh: Distributed shared state abstractions for programmable switches,” in *Proc. USENIX NSDI*, 2022, pp. 171–191.
[44] J. Lou, X. Kong, J. Huang, W. Bai, N. S. Kim, and D. Zhuo, “Harmonic: Hardware-assisted RDMA performance isolation for public clouds,” in *Proc. USENIX NSDI*, 2024, pp. 1479–1496.
[45] 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.
[46] A. Sivaraman et al., “Programmable packet scheduling at line rate,” in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 44–57.
[47] Z. Zhang et al., “VPIFO: Virtualized packet scheduler for programmable hierarchical scheduling in high-speed networks,” in *Proc. ACM SIGCOMM Conf.*, Aug. 2024, pp. 983–999.
[48] A. G. Alcoz, A. Dietmüller, and L. Vanbever, “SP-PIFO: Approximating push-in first-out behaviors using strict-priority queues,” in *Proc. USENIX NSDI*, 2020, pp. 59–76.
[49] (2024). *Intel Tofino 2*. [Online]. Available: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html
[50] (2025). *Celestica, NVIDIA, and Arista Lead the Ethernet Pack, Accounting for Nearly Two-Thirds of Sales in the Market*. [Online]. Available: https://www.delloro.com/news/infiniband-switch-sales-surged-in-2q-2025-while-ethernet-maintains-market-lead-for-ai-back-end-networks/
[51] (2025). *Market Research Reports on Data Center Switch—Front-End Networks*. [Online]. Available: https://www.delloro.com/market-research/data-center-infrastructure/data-center-switch-front-end-networks/
[52] (2022). *Tencent Implements Gateway Functionality in P4 Switching Appliance*. [Online]. Available: https://p4.org/p4-programmable-switch-appliance-in-tencent-data-center-network/

Gonglong Chen (Member, IEEE) received the Ph.D. degree from Zhejiang University in 2020. He is currently an Associate Professor with Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences. Before that, he was a Cloud Network Architect with Tencent from 2021 to 2024. His research interests include cloud networking, machine learning systems, and mobile computing.

Kejiang Ye (Senior Member, IEEE) received the B.S. and Ph.D. degrees from Zhejiang University. He was a Post-Doctoral Research Associate with Carnegie Mellon University (CMU). He is currently a Professor and the Director of the Research Center for Cloud Computing, Shenzhen Institutes of Advanced Technology and Chinese Academy of Sciences. His research interests include digital technology and systems (e.g., cloud computing, big data and industrial internet). He is a Distinguished Member of China Computer Federation (CCF).

Kai Chen (Senior Member, IEEE) received the Ph.D. degree in computer science from Northwestern University, Evanston, IL, USA, in 2012. He is currently a Professor with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong. His current research interests include data center networking, AI-centric networking, and machine learning systems.

Chengzhong Xu (Fellow, IEEE) received the Ph.D. degree from The University of Hong Kong in 1993. He is currently the Dean with the Faculty of Science and Technology and the Interim Director with the Institute of Collaborative Innovation, University of Macau, and the Chair Professor of computer and information science. His research interests include parallel and distributed computing, with an emphasis on resource management for performance, reliability, availability, power efficiency, and security.