SwitchRL: Rate Limiters on Programmable Switches in the Cloud

Anonymous Author(s)

ABSTRACT

We argue that implementing rate limiter on switches in the cloud would bring several benefits in availability, management, and cost saving, and we target to implement *a rate limiter on programmable switches for cloud networks*. (1) We address that with the consideration of switch limitations and cloud network characteristics, the rate limiter had better be implemented using token bucket algorithm with traffic policing, and the token accumulation should be triggered by packet arrival events. (2) To make the rate limiter scalable in clouds, we optimize its memory utilization by eliminating and sharing variables in the algorithm. (3) To make the rate limiter providing flexible rate control, we propose approximate operation table to complement missing computation primitives (i.e., × and /). Finally we synthesize all the design, optimization, and complement to achieve a single-rate-Three-Color-Marking (srTCM) rate limiter named SwitchRL, and the evaluation shows that SwitchRL has the features of high bandwidth saturation (>91%), low TCP throughput oscillation (<9%), memory efficient (saving 50% memory), flexible rate control (arbitrary reasonable rate), and low latency (<1us).

1 INTRODUCTION

In modern cloud networks, rate limiters play an important role in bandwidth allocation for tenants, and several solutions have been proposed and applied yet. These solutions include software rate limiters in the end hosts [16, 20, 21, 24, 28, 31] and hardware ones on end-host NICs [29]. We argue that implementing such rate limiters on *network switches* instead of end hosts would bring several benefits, including supporting more cloud business scenarios, simplifying distributed rate control, and saving server resources (§5).

While the QoS management in conventional switches usually support rate limiting, their rigid packet processing pipeline makes such rate limiters hard to scale to the cloud networks (in terms of the number of rate limiters supported in each switch). A recent trend of programmable switches provides stateful memory and customizable packet processing logic, which shows the potential to implement such rate limiters. However, their packet processing pipeline also has its own limitations (e.g., limited computation primitives and memory), and thus our goal in this paper is to *design rate limiters for cloud networks under the constraints of current programmable switches.*

Over the years, a family of rate limiting algorithms are proposed and well studied (e.g., leaky bucket algorithm, token bucket algorithm) [3, 4, 24, 31], we still face and need to overcome several challenges in making design choices and refining their implementation. First, the design space of a rate limiter has several options, and the design choice must suit cloud networks and programmable switches. The design options include algorithmic choice (leaky bucket v.s. token bucket), excessive traffic policy (traffic policing v.s. traffic shaping, i.e., dropping or buffering packets), and token accumulation method (timer-triggered v.s. packet-triggered). By analyzing the limitation of switch's programmability, we exclude leaky bucket algorithm and traffic shaping. By experiments and analysis, we found that TCP flows would likely experience throughput oscillation in timer-triggered rate limiters. The essential reason is that programmable switches cannot implement a timer with the same fine granularity as cloud network RTT (>1ms v.s. 100us). As a result, we claim that a cloud rate limiter should be designed using *token bucket* algorithm with *traffic policing*, and the *token accumulation should be triggered by per-packet events.*

Second, programmable switches have limited memory, but they must support a cloud-scale number of rate limiters. We categorize the variables (they use the memory) and spare two efforts to improve the memory efficiency of the rate limiters. (1) We refine the algorithm to save state variables, (2) we make common parameter variables to be shared among multiple rate limiters. Theoretically, for n rate limiters, the original algorithm costs at least $4n$ units of memory, and our optimization reduce the cost to be $2n + 1$.

Third, programmable switches have limited computation primitives, and the typical arithmetic operations × and / need to be complemented. Without such complement, the rate limiter can only be configured with the rate of 2^n, losing the flexibility to configure arbitrary rate. We trade storage for computation. We propose Approximate Operation Table (AOT) which precomputes the results of multiplication or division and store them in tables. Specifically, we design and analyze Approximate Division Table and Approximate Multiplication Table. In both of them, there is a tunable parameter r which can adjust the tradeoff between memory cost and the computation accuracy. As a result, we are able to design ADT with less than 2% error (smaller than the disturbance from the network) and 10X KB memory cost (negligible compared with the switch memory).

In the implementation we synthesize the three design decision, optimization, and complement. We design a single-rate-Three-Color-Marking [5] rate limiter, named SwitchRL. According to the degree by which the traffic exceeds committed rate and burst size (i.e., non-excessive, slightly excessive, seriously excessive), SwitchRL could perform pass, random drop, and drop to packets. SwitchRL is implemented in P4 language, and deployed on a switch with the Barefoot Tofino [2] chip. Our evaluation shows that the final SwitchRL could achieve flexible rate control, stable and high saturation for TCP flows in the cloud, and memory efficiency to support a cloud-scale number of rate limiters. The contributions of this paper are as follows.

(1) We propose the design choice of a rate limiter on programmable switches that suits low-latency cloud networks. It should be implemented using token bucket algorithm algorithm with traffic policing, and the token accumulation should be triggered by packet arrival events.
(2) We propose memory-efficiency optimization for rate limiter, which makes a switch support a cloud-scale number of rate limiters.
(3) We propose an approximation algorithm to complement the computation primitives on programmable switches, which makes the rate control flexible to be an arbitrary value.
(4) We synthesize the design, optimization, and complement to achieve a srTCM rate limiter. And our evaluation shows its feasibility.

2 BACKGROUND

Implementing rate limiters on programmable switch would benefit cloud networks in availability, management, and scalability. However, designing such rate limiters faces challenges because of the hardware limitations and cloud network characteristics.

2.1 Motivation and Opportunity

In existing clouds, rate limiters are usually implemented in software on dedicated servers, and many end-host solutions are proposed [13, 16, 17, 29]. We argue that on-switch rate limiters could provide extra benefits for the following reasons.

On-switch rate limiters are more applicable in certain cloud scenarios. When the network administrator does not have access to the end points (e.g., a cloud provisioning bare mental machine [8]), they cannot configure end-host rate limiting solutions (e.g., software or on-NIC rate limiters) [1, 29, 31]). Currently in such scenarios, the network would either provides sufficient bandwidth to saturate each end host's NIC (i.e., full bi-section bandwidth) or tacitly allows flows to compete at the bottleneck link, losing performance guarantee for specific clients or applications (e.g., video, emergency-serving IoT device [10, 15, 17]). On-switch rate limiter would complement the performance guarantee in this scenario.

On-switch rate limiters reduce the management complexity of distributed rate limiters. In a network where several end points need to share a fix amount of bandwidth, one solution is to coordinate distributed rate limiters at many end points [32], which, however, is non-trivial (involving fairness issues, control plane integration, and dynamic adjustment). While switches can be a location where the traffic from multiple end points aggregates, and deploying a on-switch rate limiter can avoid the management complexity of distributed rate control. For example, for a cloud tenant with multiple VMs and an aggregated bandwidth reservation, deploying a single rate limiter at the gateway switch is much simpler than coordinating multiple ones on VMs.

Deploying on-switch rate limiters is more cost-saving and scalable than that on dedicating servers. Several existing clouds would dedicate racks of servers to deploy software rate limiters, which is hard to be scalable. As each server's total processing rate is limited by its bus bandwidth (e.g., 16X PCIe 3.0 is 128Gbps); with cloud traffic increasing exponentially, more servers would be required, costing the resources which are originally designed to create rental revenue. While a switch usually have a backplane speed of 10X Tbps, which can potentially replace 10X-100X servers.

Conventional switches do not support rate limiters at a cloud scale. While a lot of commercial switches support Quality of Service (QoS) for network traffic, their implementation is usually heavy-weight and rigid — they maintain several hardware queues with different priorities per port, classify traffic to queues, and apply a rate limiter to each queue [7]. This rigid hardware pipeline constrains the total number of rate limiters that can be implemented.

Programmable switches give an opportunity to design cloud-scale rate limiters. In a recent trend of programmable switches (e.g., Barefoot Tofino [2], Cavium XPliant [6], and Intel Flexpipe [9]), packets would traverse through a pipeline of match-action tables (in stages). The pipeline also provides two features: (1) memory (e.g., SRAM) which can store stateful variables (i.e., their lifetime spans multiple packets), and (2) instructions to operate on packets and variables. As a packet traverses a programmable pipeline, computation can be performed and updates can be applied to packets and variables. The packet processing programs are written in languages like P4 [12]. Several network functions (e.g., cache, load balancer, blacklisting [18, 25, 26]) are implemented on such programmable switches, and our goal to design *a cloud-scale rate limiting system on programmable switches.*

Requirements. The rate limiters in cloud networks should satisfy the following requirements. (1) The rate limiter itself should have nice performance properties, including high rate saturation, low throughput oscillation, and low latency. (2) The rate limiter should be deployed at a scale of the number of cloud tenants. (3) Cloud traffic has the characteristics of low-latency, high-volume, and TCP dominating, the rate limiter should interact with such traffic in a correct way, i.e., maintaining the performance properties when handling such traffic.

2.2 Design Space of Rate Limiters

Over the years, rate limiting algorithms have been studied a lot [14, 19, 31]. We start from a simple rate limiter (single-rate-Double-Color-Marking) to articulate our design (from §4 to §8). More complicated one such as single-rate-Three-Color-Marking (srTCM) [5] rate limiters are designed, implemented, and evaluated in §7 and §8.

A simple rate limiter would be configured with a committed information rate (CIR, i.e., the target rate) and a committed burst size (CBS, i.e., the short-term burst that can be tolerated). And the design choices are in the following three aspects.

(1) Algorithmic Choice. There are two rate limiting algorithms — the *token bucket* algorithm and the *leaky bucket* algorithm. In the leaky bucket algorithm, a queue is maintained, "When a packet arrives, if there is room on the queue (i.e., CBS) it is appended to the queue; otherwise it is discarded. At every clock tick one packet (i.e., CIR) is transmitted unless the queue is empty [3]." The leaky bucket algorithm requires a queue and a timer in the implementation.

In the token bucket algorithm, a variable *token* is maintained to record the total token available to send packets, the token accumulates with time (i.e., CIR), and it is constrained by a burst size (i.e., CBS). For each arrival packet, if there are sufficient tokens in the bucket, the packet is sent and consumes tokens of its size (deduced from the token variable); otherwise, the packet is not sent.

(2) Excessive Traffic Policy. While in the token bucket algorithm, once a packet is decided not to be sent, it can be either dropped or buffered for later sending. The two decisions are named *traffic policing* (i.e., dropping) and *traffic shaping* (i.e., buffering) respectively [14, 19, 31]. Traffic shaping could smooth traffic (more burst tolerance), but introduce queuing delay; and traffic policing is the opposite.

(3) Token Accumulation Method. In the token bucket algorithm, "the token accumulates with time" can be implemented in two ways — timer-triggered and packet-triggered. In a timer-triggered approach, a timer periodically triggers the calculation of token accumulation, which is added to the token variable; this approach requires concurrency control on the "token" variable (e.g., locks), as it can be accessed by both the packet processing logic and the timer event logic.

In the packet-triggered approach, the packet arrival event would trigger the token accumulation. This approach does not involve concurrency control, but the token update will cost extra time in each packet's processing.

2.3 Challenges

Despite the potentials, the programmability of such switches also posts several limitations. These limitations do not mean that the corresponding features are not implementable on the hardware chips, but rather a sacrifice to avoid degrading the backplane bandwidth (which is supposed to be most primary goal of a switch). We list the following limitations and the consequent challenges in implementing rate limiters.

Challenge 1: the control flow in switch programs is different from that in a CPU program, and the rate limiter algorithm needs to be carefully designed in the whole design space. On a CPU platform, data has fixed locations in the memory and the control flow execute instructions sequentially to operate on the data; while in the hardware pipeline, the instructions has fix locations and the data (i.e., packet) flows through the pipeline to get processed.

Specifically in the programmable switch, its programming model has the following characteristics. (1) When a packet traverses the pipeline in one pass, it goes in unidirection, and cannot go back to previous instructions (or stages) [1]. (2) In the whole pipeline (i.e., `input port queues`→ `switching circuit` → `output port queues`), the programmability is limited at the switching circuit, excluding the input/output queues. These two differences would influence the choices in the rate limiter design space.

Challenge 2: to support rate limiters at a large scale, a memory efficient optimization is needed. Since the on-switch rate limiters are used to replace those in distributed servers, each switch should support (at least) a comparable number of rate limiters than that in servers. In this paper, our goal is to support rate limiters at a cloud scale, which is estimated to be $O(10^6)$. While current typical programmable switches have limited memory (e.g., ≈50 MB in Tofino), and the memory needs to be shared with other match-action tables (e.g., routing, blacklisting). An optimization to improve memory utilization is required in implementing rate limiters.

Challenge 3: the limited computation primitives constrain the implementation of a fully functional rate limiter. The current programmable switches only support

[1] A packet can be recirculated to the pipeline again in another pass, but we do not suggest such implementations. Because (1) such recirculation costs backplane processing resources, making the performance (throughput and latency) unpredictable, and (2) there is no guarantee to keep the states (i.e., variables in pipeline) consistent between two passes, causing possible logical errors.

limited operations in each Match/Action stage. For example, among the four common arithmetic operators +, −, ×, and /, only + and − are supported. Although × and / are possible to be implemented using the other two, that would introduce non-trivial implementation using P4 language. For another example, programmable switches are hard to provide fine-grained timers (<1ms), which influences the performance of timer-related rate limiters.

3 DESIGN OVERVIEW

We give an overview of designing on-switch rate limiters and overcoming the three challenges above. We elaborate each design choice, optimization, and complement in the following three sections. And finally we synthesize the solutions into a srTCM rate limiter named SwitchRL in §7.

(1) We adopt token bucket algorithm, traffic policing scheme, and packet-triggered token accumulation. According to the description in §2.3, in the switch pipeline (`input port queues→ switching circuit → output port queues`), the queues are not programmable, and thus the leaky bucket algorithm is excluded (i.e., not able to dequeue packet at a constant rate); as a packet goes in unidirection in the pipeline, once it is decided not to be sent, it can neither be buffered in the circuit nor sent back to the previous input queue, and thus the traffic shaping is excluded. Therefore, the rate limiter should adopt token bucket algorithm with traffic policing.

For the remaining choice of token accumulation method, both timer-triggered and packet triggered are feasible in implementation. However, due to the complexity and overhead, the current switch can only support coarse-grained timers (>1ms), which causes TCP throughput oscillation in low-RTT (<100us) cloud networks. Thus, SwitchRL adopts the packet-triggered token accumulation.

(2) We optimize memory efficiency by refining the token metering method and sharing parameters among rate limiters. The original token bucket algorithm has four parameters and variables (Algorithm 2), and a naive implementation of n rate limiter would need $4n$ units of memory. optPacketRL (Algorithm 3) makes the following optimizations: (1) it refines the metering method, so that the two state variables `Tc` and `prev` can be reduced to be one; (2) it shares the parameter CBS/CTW among all rate limiters. As a result, optPacketRL only requires $2n + 1$ units of memory, which indicates that the optimization enables twice of the number of rate limiters deployed in the same setting.

(3) We design Approximate Operation Table to complement missing computation primitives. Our methodology to support the missing operations is to trade storage for computation, i.e., pre-computing the results, storing the results in a table, and looking up the table during the runtime instead of computation. We design operation tables for multiplication (×) and division (/) for rate limiters. Furthermore, we refine the design to make the table tunable between

Algorithm 1: MeterRL: timer-triggered token accumulation

```
1  Parameter: CIR // Committed Information Rate
2  Parameter: CBS // Committed Burst Size
3  Parameter: Interval // token refreshment interval
4  Variable: Tc // token, initialize to CBS
5  Thread 1
6  while True do
7      pkt := getPacket()
8      size = pkt.size()
9      if size ≤ Tc then
10         send(pkt)
11         Tc -= size
12     else
13         drop(pkt)
14     end
15 end
16 Thread 2
17 while True do
18     sleep(Interval)
19     Tc = min(CBS, CIR × Interval + Tc)
20 end
```

Algorithm 2: PacketRL: packet-triggered token accumulation

```
1  Parameter: CIR // Committed Information Rate
2  Parameter: CBS // Committed Burst Size
3  Variable: Tc // token, initialize to CBS
4  Variable: prev // time of previous successful packet
     sending
5  while True do
6      pkt := getPacket()
7      size = pkt.size()
8      curr = Now()
9      Tc = min(CBS, Tc + CIR × (curr-prev))
10     if size ≤ Tc then
11         send(pkt)
12         Tc -= size
13     else
14         drop(pkt)
15     end
16 end
```

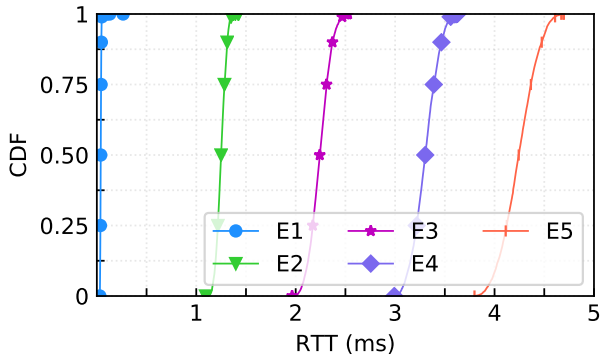

Figure 1: CDF of the RTT in the five test environments

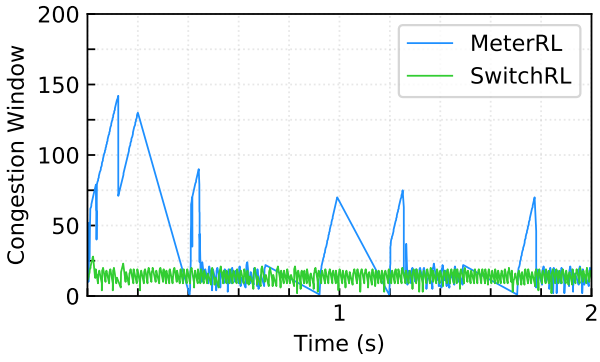

Figure 2: Congestion window of MeterRL and PacketRL change over time in E2 when CIR=128 Mbps

the computation accuracy (i.e., the approximation) and the storage space.

4 TOKEN ACCUMULATION METHOD

We implement the timer-triggered and packet-triggered token bucket algorithms, and conduct experiments to show that the packet-triggered approach is a better choice for low-latency cloud networks.

4.1 Implement and Test Two Approaches

In token bucket algorithms, a variable "Tc" is maintained to record the available tokens, and each packet sending would consume tokens of its size. The timer-triggered and packet-triggered token bucket algorithms are different in the token accumulation method. Algorithm 1 uses timer to periodically trigger the accumulation (line 16-20); since our following implementation reuses an existing P4 implementation named Meter, we call this rate limiter MeterRL. Algorithm 2 accumulates tokens for each packet arrival (line 9); since it is packet-triggered, we name it PacketRL.

We implement MeterRL and PacketRL in P4 switches and deploy them on a programmable switch with Barefoot Tofino switching ASIC. We set up a star topology with the programmable switch in the middle and several servers at the edge. Each server has 16 1.2GHz cores, 64GBmemory and a 10Gbps NIC. We use `iperf` to transmit traffic from sender to receiver and use `sockperf` to measure RTT between client and server. We observe that the primary RTT is less than 100us in our testbed, and use `tc netem` to add extra latency to emulate different network environments.

In the whole experiments, there are following parameters.

(1) **Network Latency.** We set up five environments. As shown in Figure 1. Environment "E1" represents the primary environment where no extra latency is configured, where the median RTT is 38 us. In environments "E2" ~"E5", we use `tc netem` to add an extra latency both at the sender side and receiver side, and the median RTT is 1.25 ms, 2.24 ms, 3.32ms and 4.24ms respectively.

(2) **Committed Information Rate.** We configure CIR to be 32Mbps, 64Mbps, 128Mbps and 256Mbps for both rate limiters.

(3) **Committed Burst Size.** In MeterRL, CBS varies from 2MB to 16MB; in packetRL, CBS varies from 0.1MB to 1MB.

(4) **Timer Refreshment Interval.** In MeterRL, we use the default (also minimum) value 4ms. The interval can be configured from 4ms to 4000s, but cannot be smaller due to the switch limitation.

We measure two performance metrics.

- **Average Throughput.** We send and receive traffic for 15 seconds, and compute the average value of the per-second throughput (provided by iperf).
- **Throughput Oscillation.** We compute the standard deviation of the per-second throughput to quantify the throughput oscillation.

4.2 Observations and Analysis

The performance of MeterRL and PacketRL in E1 and E2 are shown in Figure 3 and 4 respectively. Each figure shows the average throughput varying with the CBS, and the vertical line is the throughput oscillation. And we get the following observations.

Observation 1: TCP flows in MeterRL would experience throughput oscillation when token refreshment interval is much smaller than RTT; while TCP flows in PacketRL do not have oscillation. For example, when the CIR is 32Mbps, the oscillation of a MeterRL (16MB CBS) is 7.93Mbps (24.8%), while that in a PacketRL (1.2MB CBS) is 2.58Mbps (8.1%).

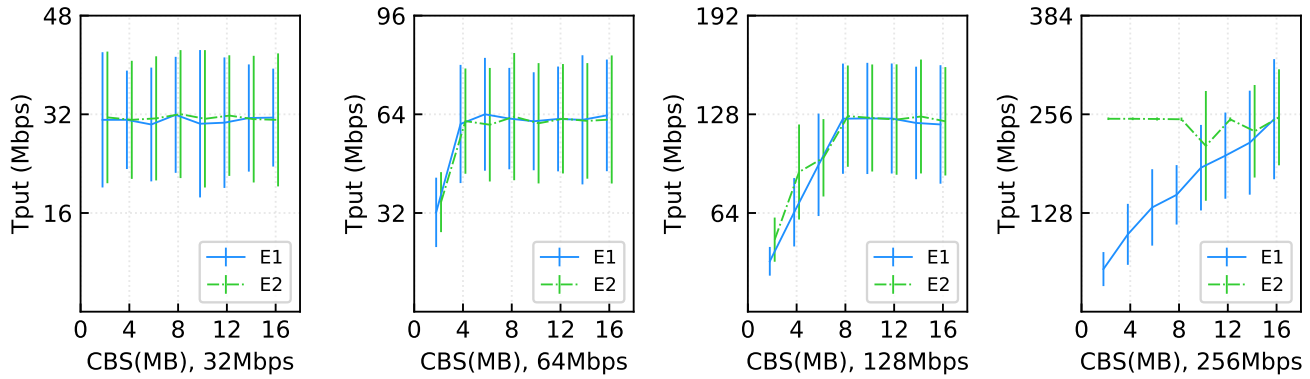

Figure 3: Average throughput of MeterRL at different rate-limiting

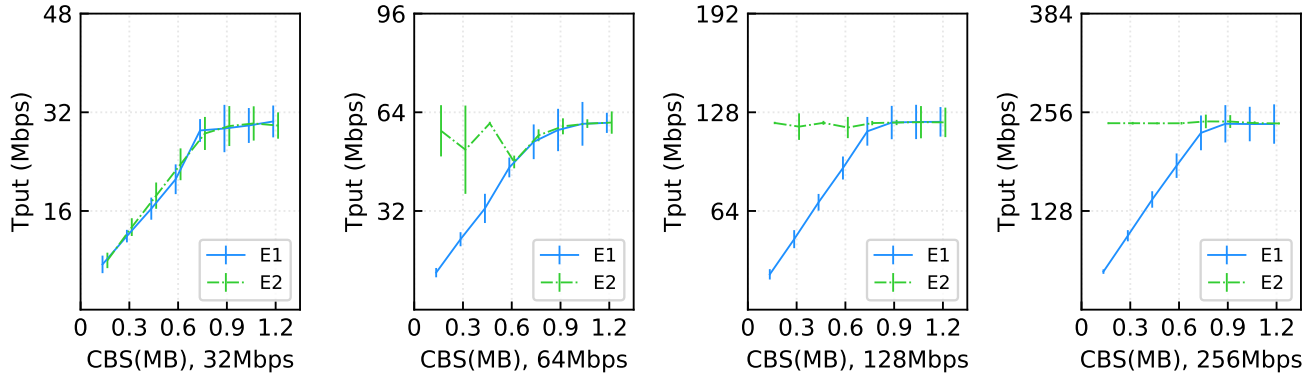

Figure 4: Average throughput of PacketRL at different rate-limiting

Token accumulation in a MeterRL is periodically bursty with the token refreshment interval; but the TCP window adjustment is performed every RTT. In meterRL with E1 and E2, the interval is larger than RTT (4ms v.s. 38us and 1.25ms), whenever tokens are refreshed a TCP flow would increase its congestion window dramatically, TCP would send packets with a large window, and when the tokens are exhausted between refreshments, TCP would drop packets and reduce window size. While in a PacketRL, the token is updated every packet, whose arrival interval is smaller than RTT, avoiding bursty token accumulation. Figure 2 shows the window size variation in MeterRL and PacketRL with CIR to be 128Mbps; the TCP flows' congestion window in MeterRL shows a more dramatic adjustment than that in PacketRL.

Table 1 in the Appendix shows the complete results of the standard deviation of MeterRL with varying CBS, RTT, and CIR; and the refreshment interval is the minimum value 4ms. When the RTT approximates to the interval (e.g., E3 with median RTT to be 3.32ms), the oscillation would be less obvious.

In current programmable switches, the timer interval cannot be finer-grained than 1ms. The on-chip timer is coarse-grained (interval > 1ms) to avoid the overhead (i.e., overhead to execute timer and contend for memory access with the pipeline) and its programmability is not provided either. Another way to implement the timer logic is to let the switch OS to trigger the event, which would traverse the whole switch software and system stack, and cannot be finer-grained either. While in typical data centers, the RTT is usually smaller than 100us [33]. And thus, MeterRL is not a good option for data center networks.

Observation 2: in both rate limiters, CBS needs to reach a threshold for TCP flows to saturate the CIR, but its value does not affect the oscillation obviously. In experiments with MeterRL, E1, and CIR = 128Mbps, when CBS is 4MB, the mean throughput is only about 95.2Mbps, when CBS is 8MB, the mean throghput reaches 125.3Mbps,

and when CBS is 12MB and 16MB, the oscillation does not get better. In addition, we get other observations: (3) A larger CIR can alleviate the throughput oscillation, but the alleviation is not significantly. (4) UDP packets can always saturate the CIR.

Summary: In low-latency cloud networks, PacketRL is more friendly to TCP flows than MeterRL. And CBS needs to be configured for flows to saturate CIR and does not affect throughput oscillation.

5 OPTIMIZE MEMORY EFFICIENCY

Memory is scarce resource on switch chips and can potential limit the scalability of rate limiters (in terms of the number of rate limiters per switch). Thus, we refine the token bucket algorithm to use less memory.

The original token bucket algorithm (PacketRL) needs the following memory space: (1) two parameters CBS and CIR, which are configured during the initialization and does not vary during execution, (2) two state variables `Tc` and `prev`, whose lifetime is the same with the whole program spanning multiple packets, and (3) temporary variables `size` and `curr`, whose lifetime is only the current packet. The temporary variables does not cost memory (they are in temporary registers), and there remains four parameters and state variables. Thus, a naive implementation of n rate limiters would cost at least $4n$ units of memory.

Our intuition to optimize memory utilization is to share parameters among rate limiters and to refine algorithm to eliminate state variables. And we finally achieve an algorithm whose memory cost is $2n + 1$ units for n rate limiters.

Algorithm 3: optPacketRL: Memory utilization is optimized

```
1  Paremeter : CIR // Committed Information Rate
2  Paremeter : CTW // Committed Time Window
3  Variable : accuCons // Accumulated consumption,
     starting from init_time
4  while True do
5      pkt := getPacket()
6      size = pkt.size()
7      curr = Now()
8      cons = size / CIR
9      if accuCons + cons ≤ curr then
10         accuCons = max(curr - CTW, accuCons) + cons
11         send(pkt)
12     else
13         drop(pkt)
14     end
15 end
```

Eliminate State Variables The semantics of the token bucket algorithm (PacketRL in Algorithm 2) includes: (1) Tokens "Tc" (or budget) accumulates with time (line 9, CIR×(curr-prev)), (2) Tc is consumed by successful packet sending (line 12), (3) consumption cannot exceeds the Tc accumulation (line 10), and (4) Tc accumulation cannot exceeds burst size (line 9, Tc = min(CBS, Tc+...)). The unit of Tc and consumption is byte.

While we change the budget (Tc) and consumption unit to be the same with time (e.g., 1us), and rephrase the semantics in time domain (Algorithm 3). (1) Budget accumulates naturally with time elapse, which can be denoted as curr (starting from init_time), (2) budget is consumed by successful packet sending, each packet consumes packet_size/CIR (line 8), and the total consumption is accumulated (line 9, "+consp"), (3) consumption cannot exceeds the budget (line 9), and (4) equivalently to bounding budget accumulation (by CBS), we choose another way: once the budget accumulation exceeds the consumption by CBS/CIR, the consumption is forced to be accumulated (line 10, max(curr - CBS/CIR, accuCons)).

The Algorithm 3 is an optimization of PacketRL, we name it optPacketRL. In the algorithm, the variable "accuCons" takes the place of "Tc" and "prev" in packetRL. As in optPacketRL, burst size is expressed in CBS/CIR, we use a new parameter CTW (meaning committed time window) to replace the quotient.

Share Parameters. For the two parameters CIR and CBS in packetRL (or CTW in optPacketRL), CIR is the unique configuration in individual rate limiters, while CBS can be shared among rate limiters. According to the observation (2) in §4.2, CBS (or CTW) should be sufficiently large to allow TCP flows saturating the CIR, but once CBS/CTW reaches a threshold it does not affect oscillation. Thus, we use the same CBS (or CTW) across all rate limiters.

Summary: The original token bucket algorithm consumes $4n$ units memory for n rate limiters; our optimization reduce that to be $2n + 1$ units.

6 COMPLEMENT COMPUTATION PRIMITIVES

Current programmable switches usually only support limited computation primitives. The essential reason is that switch design has the backplane bandwidth as the most primary goal, and would sacrifice costly operations to avoid degrading packet processing speed. As an example, Barefoot Tofino could support bit shifting operations (<< and >>), limited arithmetic operations (+ and −), and compound operations such as hash functions, but it does not support some other common operations such as × and /. In the near future, backplane bandwidth is still the most primary goal

for switches, and adding new computation hardware to the switch chips would involve non-trivial demonstration, design, manufacturing, and testing processes; thus, we turn to consider complementing the missing primitives under the constraints of the current switches.

The rate limiter algorithm requires several missing computation primitives in current switches, including CIR × Interval in MeterRL(Algorithm 1), CIR × (curr-prev) in PacketRL (Algorithm 2), and size/CIR in optPacketRL (Algorithm 3). In the current programmable switch, we can only use $>> n$ and $<< n$ to compute $\times 2^n$ and $/2^n$, and thus the timer interval in MeterRL and the CIR in PacketRL and optPacketRL can only be 2^n units, which constrains the rate control flexibility.

Pre-compute operations with fixed operands. To compute an operation $f(x_1, x_2, ..., x_n)$, if the operands' value is fixed, the result can be pre-computed. During the compilation, the result can replace the appearance of the operation in the program. In the rate limiter examples, once the program and parameters are fixed (e.g., CBS, CIR, and Interval), several operations can be pre-computed and replaced, including CIR × Interval in MeterRL and size/CIR in optPacketRL.

Pre-compute operations with range operands. For operations whose operands are in a range with finite elements, we can enumerate all possible operand combinations in the range and pre-compute the result of each enumeration, and then transform the operation into a lookup table (operands as the key and the corresponding computation result as the value).

Take a binary operation $f(x, y)$ as an example. If the operands are in a range with finite elements, i.e., $x \in \{X_1, X_2, ..., X_n\}$ and $y \in \{Y_1, Y_2, ..., Y_m\}$. Then the result of each operand combination x_i and y_j ($1 \le i \le n$ and $1 \le j \le m$) can be pre-computed $f(x_i, y_j)$, and the table can be stored with $< x_i, y_j >$ or $< i, j >$ as the key and $f(x_i, y_j)$ as the value. The lookup table can provide precise computation results.

If the operands are in a range with infinite elements, then the algorithm would be an *approximate* algorithm. For example, assume $x \in [X_L, X_H]$ and $y \in [Y_L, Y_H]$, the whole input space can be divide into subspaces by dividing each operand's range separately. The x dimension can be divided as follows.

$$[X_L, X_H] \subset \bigcup_{i=0}^{n-1} [x_i, x_{i+1})$$

$$[x_i, x_{i+1}) \bigcap [x_j, x_{j+1}) = \varnothing, i \neq j$$

Then each subspace can use an approximate value to represent the results of all operand combinations in that subspace. For all $x \in [x_i, x_{i+1})$ and $y \in [y_j, y_{j+1})$, an approximate value $approxVal$ can be chosen as the computation result of $f(x, y)$

$$approxVal \in f_{x \in [x_i, x_{i+1}), y \in [y_j, y_{j+1})}(x, y).$$

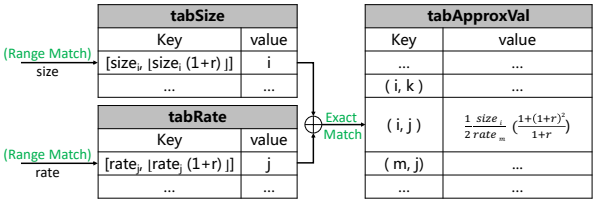

Figure 5: Approximate Operation Table (Division)

The lookup table can have $< i, j >$ as the key and $approxVal$ as the value.

In rate limiter algorithms, the inputs are finite positive integers, a precise lookup table can be constructed. But to make a tradeoff between the storage (i.e., table entries) and the accuracy (result precision), we still design an approximate lookup table. We elaborate the construction of an Approximate Division Table (ADT) and analyze its accuracy and space complexity, and the similar construction and analysis of multiplication is in the Appendix.

To compute x/y, we assume $x, y \in \mathbb{N}^+, x \in [X_L, X_H]$ and $y \in [Y_L, Y_H]$. We introduce a parameter $r \in (0, 1)$, which is used to divide value space of x, y and guarantee the result precision. $[X_L, X_H]$ is divided for $\forall i, j = 0 \cdots N, i \neq j$ as follows:

$$\begin{cases} [X_L, X_H] \subset \bigcup_{i=0}^{N} [x_i, \lfloor x_i(1+r) \rfloor] \\ [x_i, \lfloor x_i(1+r) \rfloor] \bigcap [x_j, \lfloor x_j(1+r) \rfloor] = \varnothing \end{cases} \quad (1)$$

where $x_0 = X_L, x_N \le X_H$ and $\lfloor x_i(1+r) \rfloor + 1 = x_{i+1}$. And $[Y_L, Y_H]$ is divided in the same way. For a subspace where $x \in [x_i, \lfloor x_i(1+ratio) \rfloor]$ and $y \in [y_j, \lfloor y_j(1+ratio) \rfloor]$, as division is a monotonic function, we can compute the lower bound of the result to be

$$lower = \frac{x_i}{\lfloor y_j(1+r) \rfloor},$$

the upper bound to be

$$upper = \frac{\lfloor x_i(1+r) \rfloor}{y_j}.$$

We choose $approxVal$ to be

$$approxVal = \frac{lower + upper}{2}$$

The computation error is defined as

$$\frac{\frac{x}{y}}{approxVal} - 1,$$

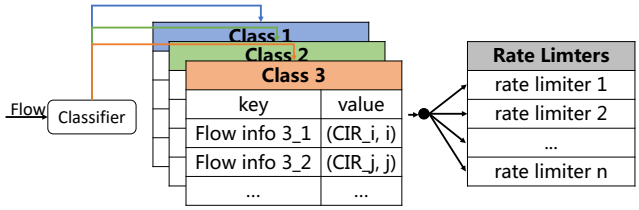

Figure 6: A rate limiter system with switchRL

which is in the range

$$[\frac{lower}{approxVal} - 1, \frac{upper}{approxVal} - 1]$$
$$=[-\frac{(1+r)^2-1}{(1+r)^2}, \frac{(1+r)^2-1}{(1+r)^2}].$$

We use $error = \frac{(1+r)^2-1}{(1+r)^2+1}$ to represent the maximum computation error.

In the ADT, the total number of entries is

$$V_1 = \lceil log_{1+r}(X_H/X_L) \rceil \times \lceil log_{1+r}(Y_H/Y_L) \rceil.$$

If the division is implemented in a precise lookup table, there would be $V_2 = (X_H - X_L + 1) \times (Y_H - Y_L + 1)$ entries. Thus, ADT trades computation precision with memory spaces.

In Algorithm 3, if the CIR is between 1Mbps and 10000Mbps and the packet size is between 1byte and 10000bytes, then we have $x, y \in [1, 10000]$ and $r = 0.02$. The ADT would have $error = 0.0198$, $V_1 = 87025$ and $V_1/V_2 = 0.00087$. Compared with the typical data center rate limiter saturation (e.g., 93.8%~96% in HTB [19]), the precision error is within the variation caused by other factors and the space cost is negligible.

Summary: We design approximate operation tables to complement the missing computation primitives in programmable switches; for rate limiters, we design approximate tables for division and multiplication, where the tradeoff between storage space and computation is tunable.

7 SYNTHETIC IMPLEMENTATION

We synthesize the design choice, optimization, and complement in §4, §5, and §6, and implement SwitchRL as a single-rate-Three-Color-Marking[2] rate limiter (srTCM [5]). Algorithm 4 shows its logic. Compared with the simple rate limiter in previous sections (Algorithm 2 and 3), SwitchRL supports two burst size CBS (committed burst size) and EBS (excess burst size). In the SwitchRL, they are transformed into CTW and ETW, where CTW = CBS / CIR and ETW = EBS / CIR. At the arrival of new packets, the time corresponding to the token consumption is obtained from ADT

[2]The algorithm would mark packets in three colors: GREEN means pass, YELLOW means random drop, and RED means drop.

Algorithm 4: SwitchRL: srTCM with three optimizations

```
1  Paremeter : CIR // Committed Information Rate
2  Paremeter : CTW // Committed Time Window
3  Paremeter : ETW // Excess Time Window
4  Variable : accuConsp // Accumulated consumption,
     initialized to init_time
5  while True do
6      pkt := getPacket()
7      size = pkt.size()
8      curr = Now()
9      idxSize = tabSize.rangeMatch(size)
10     idxRate = tabRate.rangeMatch(CIR)
11     consp = tabApproxVal.exactMatch(idxSize,idxRate)
12     if accuConsp + consp ≤ curr then
13         eTime = curr - ETW
14         cTime = eTime - CTW
15         if accuConsp > eTime then
16             randomDrop(pkt)
17         end
18         accuConsp = max(cTime, accuConsp) + consp
19         send(pkt)
20     else
21         drop(pkt)
22     end
23 end
```

(line 9 — 11). As illustrated in Figure 5, there are two tables, tabSize and tabRate, for range matching of packet size and CIR to determine to which two sub-intervals packet size and CIR belong, and then get the approximation consp of packet_size / CIR from tabApproxVal. Packets would be forwarded (line 19) if accuConsp is $\in [-\infty, eTime]$, which means tokens are adequate; once CBS tokens are exhausted (accuConsp *in* $(eTime, curr - consp]$), EBS tokens would be consumed and packets are sent with random drop (line 15 — 17); otherwise, packets would be dropped (line 21).

In the implementation and deployment, we assume that different priorities and burst sizes are guaranteed for different flows by ISP. We categorize flows with the same priority and burst size into one class table with each entry representing a flow, and one dedicated SwitchRL rate limiter is associated with one or more entries within one class table. These rate limiters are also maintained as a table, of which each entry only records a stateful variable accuConsp. Figure 6 depicts the architecture, which implements in several tables chained together in the programmable switch. Each packet would go through the following match action tables: (1) a classification table that maps the packet to a class table;

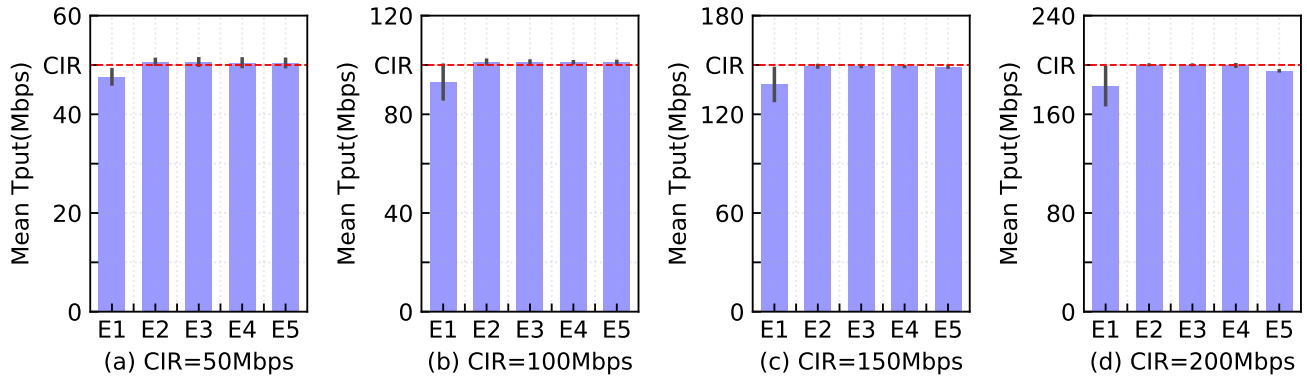

Figure 7: Mean throughput of switchRL in different test environments, varing Committed Information Rate(CIR)

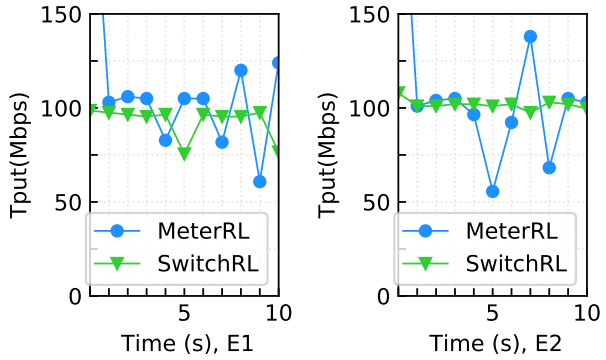

Figure 8: Mean throughput of MeterRL and SwitchRL

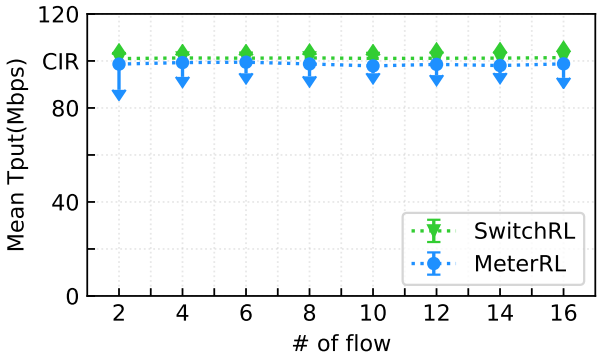

Figure 9: Mean throughput of meterRL and switchRL in E1 when CIR=100Mbps, varing flow number

(2) from the class table, the index (assume k) of the flow's rate limiter as well as its parameters (CIR, CTW, and ETW) are obtained; (3) finally the Algorithm 4 is executed with the parameters obtained from (2) on the k-th rate limiter.

8 EVALUATION

We measure the performance and overhead of SwitchRL to show its feasibility. The experiments are conducted in the same setting with §4. In all experiments, we configure the CTW and ETW in SwitchRL to be 17.5ms and 25ms respectively; and all end hosts use RENO for TCP flows.

8.1 Functional Validation

CIR saturation and throughput oscillation. We configure the switchRL with different CIR (50Mbps, 100Mbps, 150Mbps and 200Mbps) and evaluate it in different environments (E1, E2, E3, E4 and E4). Figure 7 depicts the mean throughput for all different configurations, and the vertical lines show the throughput oscillation (i.e., the standard deviation of the throughput). We get the following observations.

(1) SwitchRL can achieve high saturation in most cases. In E2 ~E5, the saturation of CIR is perfect between 99.23% and 101.19%; in E1 the saturation is in [91.4%, 95.2%], which is still acceptable. In practice, higher CIR can be configure to complement the insufficient saturation.

(2) SwitchRL has stable TCP throughput without dramatic oscillation. In E2 ~E5, the throughput oscillation is between 0.58% and 2.24%; in E1 the oscillation is a bit larger in [3.6%, 8.2%], but still acceptable. While the performance in E1 is slightly worse than E2~E5, the following experiments show that it is still better than other current on-switch implementation.

meterRL v.s. SwitchRL. We compare MeterRL with SwitchRL as the former is an existing implementation in the current P4 library. We visualize their throughput in Figure 8 with two settings <E1, CIR=100Mbps> and <E2, CIR=100Mbps>. We

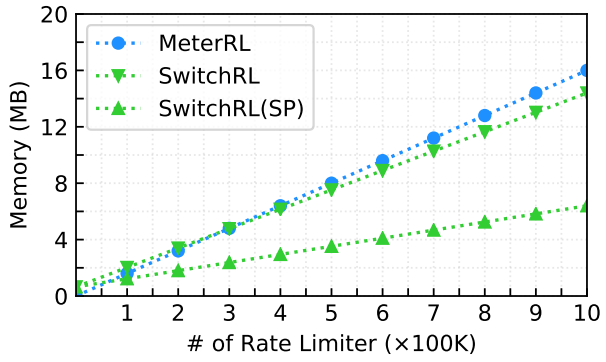

Figure 10: Memory usage of meterRL, switchRL and switchRL (SP, share parameters), varying the number of rate limiters

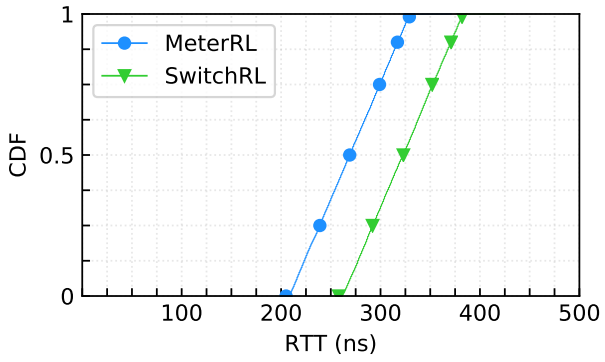

Figure 11: Latency caused by meterRL and switchRL

find that both rate limiters has high saturation (>93%), but SwitchRL has significantly smaller oscillation than meterRL (7.52% v.s. 22.56% in E1 and 1.41% v.s. 23.94% in E2).

One rate limiter for aggregated flows. In practice, a tenant may be configured with one aggregated rate for its multiple flows, thus we examine the performance of SwitchRL when controling multiple TCP flows. We configure SwitchRL and MeterRL, and vary the TCP flow number from 1 to 16 by changing the number of sender and receiver pairs. The result of experiment with E1 and CIR=100Mbps is illustrated in Figure 9. X-axis is the varying number of TCP flows, and for each point in the figure, the marker shows the mean throughput and the length of the vertical arrow line shows the throughput oscillation of the aggregated throughput. We observe that:

(1) Both rate limiters show high saturation, but SwitchRL still outperform meterRL in throughput oscillation (1.10% ~2.36% v.s. 3.13% ~10.95%).

(2) Increasing the number of TCP flows would help alleviate the throughput oscillation, and this alleviation is more significant for MeterRL.

8.2 Overhead

Memory. We deploy MeterRL, SwitchRL without shared parameters, and SwitchRL (default with shared parameters). We vary the number of rate limiters from 0 to 1 million, and draw the memory usage in Figure 10. We observe that SwitchRL (with and without shared parameter) initially consumes more memory than MeterRL; the reason is that SwitchRL has precomputed ADT, which consumes some memory.

While as the number of rate limiters increases, SwitchRL shows better memory efficiency. When 1 million rate limiters are deployed, MeterRL, SwitchRL without shared parameters, and SwitchRL utilize 16MB, 14.4MB, and 6.4MB memory respectively. With one switch having memory size of about 50MB, SwitchRL could support the rate limiter requirement at the cloud scale ($O(10^6)$).

Latency. We measure whether rate limiters introduce extra latency to traversing packets. We generate 10000 packets whose length is uniformly distributed in [100, 1500], and each packet is assigned with a distinct destination IP address to enforce a distinct rate. We record the timestamps for each packet at the ingress and egress Switch pipelines, and compute the difference of the two timestamps as the latency that a packet experienced in the pipeline. We measure the latency of MeterRL and SwitchRL, and Figure 11 shows the CDF of the 10000 packets' latency.

We observe that SwitchRL show longer latency than MeterRL 53.4ns more (median to be 318.9 v.s. 265.5). The reason is that SwitchRL implement ADT which uses more stages in the switch pipeline than MeterRL, and the extra stages introduce such extra latency. Considering the end-to-end flow latency (e.g., RTT is 38us in E1), the latency introduced by either rate limiter is less than 1us, which are negligible.

9 RELATED WORK

Other rate limiters. Rate limiters are common software provided by operating system; for example, Linux provides HTB, TBF, etc. to perform rate limiting by supporting queue discipline (qdisc) operations. They usually adopt traffic shaping (either leaky bucket or token bucket) as there is more flexible programmability on CPU platforms. However, as argued in §2.1, in the cloud environment, dedicated extra racks of servers are costly and do not scale with the tenants' traffic volume.

Hardware rate limiters. Hardware rate limiters are implemented either in NICs [29] or in switches [7, 30]. In general, the implementation on current NICs and conventional switches are too rigid. The existing solutions are configurable, not programmable; they configure a queue for each traffic

class and deploy a rate limiter for each queue. Consequently, they are limited by the number of queues on the hardware (e.g., 8–128 queues in NICs [29], or 4 queues per port in swithces [7]), which is hard to be scalable in the cloud.

Rate limiting policies. A bunch of work discuss about rate limiting policies, i.e., how to decide the committed rate to network entities (e.g., a flow or a server). And the example scenarios are cloud networks [21, 30], content delivery [15, 17], and IoT [10]. This paper discusses the problem if the committed rate is decided, how to enforce the exact rate control.

Bandwidth sharing. There exist a set of work [13, 24, 28] about weighted bandwidth sharing, which solves the problem of "allocating bandwidth proportionally to weights". And rate limiting is to solve a different problem that "constrains the sending rate at a precise value."

Programmable switch applications. Programmable switches [2, 12] have been used in implementing many network appliances. There are examples such as load balancers [27], caching [22, 23], and monitoring [26]. Our approximate table that overcomes the computation primitive limitation would provide a methodology for other applications meeting with the similar problem.

10 CONCLUSIONS

We point out the benefits in deploying rate limiter on switches in cloud networks and design a rate limiter on programmable switches. We show that under the current programmable model and capability of programmable switches, the rate limiter has better be implemented using the token bucket algorithm with traffic policing, and the token accumulation should be triggered by packet events instead of timer events. We then refine the algorithm to reduce memory usage by eliminating and sharing variables so that the rate limiter can be deployed at a cloud scale ($O(10^6)$). Finally, we propose a approximate table method to complement missing computation primitives so that the rate limiter can have flexible rate control. We synthesize the design, optimization, and complement and implement a srTCM rate limiter named SwitchRL, and our evaluation shows that SwitchRL has the feature of high saturation of CIR, low TCP throughput oscillation in low-latency cloud networks, memory efficiency, fully functional rate control, and low latency.

REFERENCES

[1] [n. d.]. http://man7.org/linux/man-pages/man8/tc-htb.8.html. ([n. d.]).
[2] [n. d.]. https://barefootnetworks.com/products/brief-tofino/. ([n. d.]).
[3] [n. d.]. https://en.wikipedia.org/wiki/Leaky_bucket. ([n. d.]).
[4] [n. d.]. https://en.wikipedia.org/wiki/Token_bucket. ([n. d.]).
[5] [n. d.]. https://tools.ietf.org/html/rfc2697. ([n. d.]).
[6] [n. d.]. https://www.cavium.com/xpliant-packet-trakker-programmable-telemetry-solution.html. ([n. d.]).
[7] [n. d.]. https://www.cisco.com/c/en/us/support/docs/smb/switches/cisco-esw2-series-advanced-switches/smb4079-quality-of-service-qos-queue-settings-on-esw2-350g-switches.html. ([n. d.]).
[8] [n. d.]. https://www.cloudlab.us. ([n. d.]).
[9] [n. d.]. https://www.intel.com/content/www/us/en/trademarks/intel-flexpipe.html. ([n. d.]).
[10] Basim KJ Al-Shammari, Nadia Al-Aboody, and Hamed S Al-Raweshidy. 2018. IoT traffic management and integration in the QoS supported network. *IEEE Internet of Things Journal* 5, 1 (2018), 352–370.
[11] Mohammad Alizadeh, Adel Javanmard, and Balaji Prabhakar. 2011. Analysis of DCTCP: stability, convergence, and fairness. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*. ACM, 73–84.
[12] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.
[13] Li Chen, Yuan Feng, Baochun Li, and Bo Li. 2018. Efficient Performance-Centric Bandwidth Allocation with Fairness Tradeoff. *IEEE Transactions on Parallel and Distributed Systems* 29, 8 (2018), 1693–1706.
[14] Tobias Flach, Pavlos Papageorge, Andreas Terzis, Luis Pedrosa, Yuchung Cheng, Tayeb Karim, Ethan Katz-Bassett, and Ramesh Govindan. 2016. An internet-wide analysis of traffic policing. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 468–482.
[15] Joshua B Gahm, Saamer Akhshabi, Ali C Begen, David R Oran, Biswaranjan Panda, and Frederick Baker. 2016. Stabilization of adaptive streaming video clients through rate limiting. (April 5 2016). US Patent 9,306,994.
[16] Sergio Leon Gaixas, Jordi Perelló, Davide Careglio, Eduard Grasa, and Miquel Tarzán. 2019. End-user traffic policing for QoS assurance in polyservice RINA networks. *Telecommunication Systems* 70, 3 (2019), 365–377.
[17] Mei Guo, Jun Xin, Yeping Su, Chris Y Chung, ZHOU Xiaosong, and Hsi-Jung Wu. 2019. Scene Based Rate Control for Video Compression and Video Streaming. (April 4 2019). US Patent App. 15/724,798.
[18] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 357–371.
[19] Keqiang He, Weite Qin, Qiwei Zhang, Wenfei Wu, Junjie Yang, Tian Pan, Chengchen Hu, Jiao Zhang, Brent Stephens, Aditya Akella, et al. 2017. Low latency software rate limiters for cloud networks. In *Proceedings of the First Asia-Pacific Workshop on Networking*. ACM, 78–84.
[20] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. 2015. Silo: Predictable message latency in the cloud. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 435–448.
[21] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. 2013. EyeQ: Practical network performance isolation at the edge. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*. 297–311.
[22] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. Netchain: Scale-free sub-rtt coordination. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 35–49.
[23] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 121–136.

[24] Murizah Kassim, Aini Azmi, Ruhani Ab Rahman, Mat Ikram Yusof, Roslina Mohamad, and Azlina Idris. 2018. Bandwidth Control Algorithm on YouTube Video Traffic in Broadband Network. *Journal of Telecommunication, Electronic and Computer Engineering (JTEC)* 10, 1-5 (2018), 151–156.

[25] Naga Katta, Aditi Ghag, Mukesh Hira, Isaac Keslassy, Aran Bergman, Changhoon Kim, and Jennifer Rexford. 2017. Clove: Congestion-aware load balancing at the virtual edge. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*. ACM, 323–335.

[26] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. FlowRadar: a better NetFlow for data centers. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*. 311–324.

[27] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 15–28.

[28] Kanthi Nagaraj, Dinesh Bharadia, Hongzi Mao, Sandeep Chinchali, Mohammad Alizadeh, and Sachin Katti. 2016. Numfabric: Fast and flexible bandwidth allocation in datacenters. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 188–201.

[29] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. 2014. {SENIC}: Scalable {NIC} for End-Host Rate Limiting. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*. 475–488.

[30] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C Snoeren. 2007. Cloud control with distributed rate limiting. In *ACM SIGCOMM Computer Communication Review*, Vol. 37. ACM, 337–348.

[31] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Carlo Contavalli, Amin Vahdat, et al. 2017. Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 404–417.

[32] Alan Shieh, Srikanth Kandula, Albert G Greenberg, and Changhoon Kim. 2010. Seawall: Performance Isolation for Cloud Datacenter Networks.. In *HotCloud*.

[33] Haitao Wu, Jiabo Ju, Guohan Lu, Chuanxiong Guo, Yongqiang Xiong, and Yongguang Zhang. 2012. Tuning ECN for data center networks. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. ACM, 25–36.

APPENDIX

A. THE CONSTRUCTION OF AMT

The construction of Approximation Multiplication Table (AMT) is similar with ADT. To compute $x \times y$, we assume $x, y \in \mathbb{N}^+, x \in [X_L, X_H]$ and $y \in [Y_L, Y_H]$. We introduce a parameter $r \in (0, 1)$, which is used to divide value space of x, y and guarantee result precision. $[X_L, X_H]$ is divided for $\forall i, j = 0 \cdots N, i \neq j$ as follows:

$$\begin{cases} [X_L, X_H] \subset \bigcup_{i=0}^{N} [x_i, \lfloor x_i(1+r) \rfloor] \\ [x_i, \lfloor x_i(1+r) \rfloor] \bigcap [x_j, \lfloor x_j(1+r) \rfloor] = \varnothing \end{cases} \tag{2}$$

where $x_0 = X_L$, $x_N \leq X_H$ and $\lfloor x_i(1+r) \rfloor + 1 = x_{i+1}$. And $[Y_L, Y_H]$ is divided in the same way. For a subspace where $x \in [x_i, \lfloor x_i(1+ratio) \rfloor]$ and $y \in [y_j, \lfloor y_j(1+ratio) \rfloor]$, we can compute the lower bound of the result to be

$$lower = x_i \times y_i$$

the upper bound to be

$$upper = \lfloor x_i(1+r) \rfloor \times \lfloor y_j(1+r) \rfloor$$

We choose $approxVal$ to be

$$approxVal = \frac{lower + upper}{2}$$

The computation error is defined as

$$\frac{\frac{x}{y}}{approxVal} - 1,$$

which is in the range

$$\begin{aligned} &[\frac{lower}{approxVal} - 1, \frac{upper}{approxVal} - 1] \\ =&[\frac{1-(1+r)^2}{1+(1+r)^2} - 1, \frac{2\times(1+r)^2}{1+(1+r)^2} - 1]. \end{aligned}$$

Similarly, in the AMT, the total number of entries is

$$V_1 = \lceil log_{1+r}(X_H/X_L) \rceil \times \lceil log_{1+r}(Y_H/Y_L) \rceil.$$

while

$$V_2 = (X_H - X_L + 1) \times (Y_H - Y_L + 1)$$

if the multiplication is implementated in a precise lookup table.

B. THE EFFECT ON THROUGHPUT OF ADT

We argue that the effect on throughput caused by calculation error introduced by Approximation Operation Table is negligible. Next, we take ADT for example in Algorithm 4. When handling the burst of traffic, consider the `CTW` and `ETW` are full at time t, a stream of packets arrive at switchRL in the following period of time Δt, the average instantaneous rate of the flow is

$$insRate = CIR \times \frac{\Delta t^{'} + CTW + ETW \times lr}{\Delta t} \tag{3}$$

where $\Delta t^{'} \in [\frac{\Delta t}{1+error}, \frac{\Delta t}{1-error}]$, $error = \frac{(1+r)^2-1}{(1+r)^2+1}$, and lr is the packet loss rate when burst size is larger than CTW. $insRate$ varing with Δt follows the following equation

$$\lim_{\Delta t} insRate = \begin{cases} +\infty & \Delta t \to 0 \\ meanTput & \Delta t \to +\infty \end{cases} \tag{4}$$

As a result, switchRL allows burst of traffic while guaranteeing the mean throughput in the long run.

In addition, throughput is influenced by many factors and can not be guaranteed absolutely. For instance, the bandwidth saturation is between 91% — 96% for HTB[19], 94% —

100% for DCTCP[11]. Consequently, though ADT may introduce calculation error to the final results, the gap between actual throughput and CIR can be ignored in pratice.

C. RESULTS OF METERRL

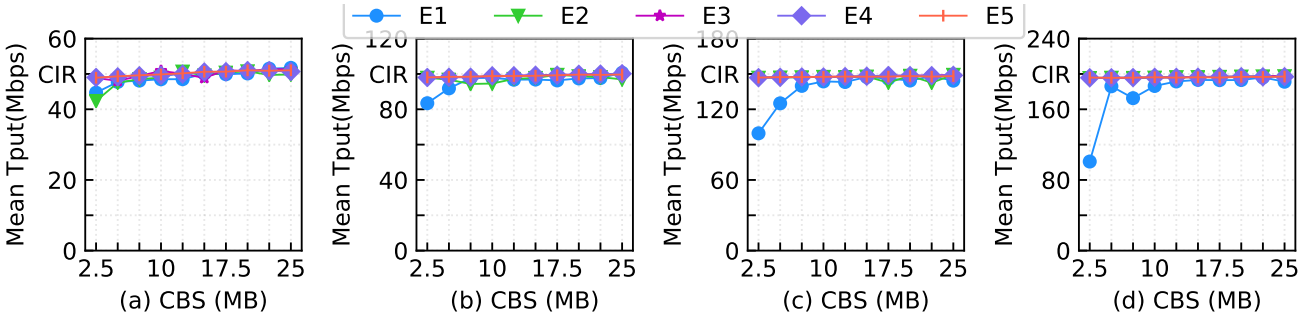

Figure 12: Mean throughput of meterRL in different test environments, varing Committed Information Rate(CIR=50Mbps,100Mbps,150Mbps,200Mbps) and Committed Burst Size

Table 1: Standard deviation(STD) of mean throughput varies with Rate and CBS in different test environments, EBS=$\frac{1}{4}$CBS

STD(%) CBS(MB) / Rate(Mbps)		2.5	5	7.5	10	12.5	15	17.5	20	22.5	25
50	E1	15.6	30.2	28.0	21.1	27.1	21.2	17.1	19.3	12.8	14.1
	E2	21.2	34.8	25.2	26.7	27.9	29.5	27.8	23.9	28.3	25.8
	E3	3.2	23.2	18.8	16.6	5.4	31.2	18.3	7.3	7.1	8.5
	E4	1.8	3.2	4.3	5.1	4.7	6.5	7.2	7.0	7.2	10.6
	E5	1.8	3.5	4.0	4.6	4.9	5.4	7.1	6.9	7.4	8.7
100	E1	20.8	11.8	16.4	26.8	28.9	27.0	21.2	23.1	25.4	19.3
	E2	0.9	5.8	33.5	29.5	28.4	4.2	26.8	24.3	6.9	12.4
	E3	1.3	1.8	2.0	2.8	2.6	3.5	3.2	3.8	4.9	4.9
	E4	1.6	1.8	2.0	2.5	2.3	3.3	3.3	4.1	4.2	4.1
	E5	2.8	3.1	3.0	2.5	2.8	3.1	3.0	4.2	4.3	4.0
150	E1	7.0	17.4	14.8	11.6	27.1	26.1	24.9	26.1	23.6	25.3
	E2	0.6	1.0	1.7	1.8	1.7	2.1	20.9	3.0	18.9	3.0
	E3	0.9	1.3	1.3	1.2	1.7	1.9	2.0	2.5	2.6	2.8
	E4	0.8	1.3	1.6	1.6	1.7	2.3	2.1	2.7	2.9	3.0
	E5	2.0	1.6	1.9	1.6	2.5	2.0	2.3	2.1	2.7	2.5
200	E1	7.4	13.0	18.8	16.6	27.7	14.0	26.4	23.8	24.5	26.2
	E2	0.7	0.8	1.0	1.3	1.3	1.6	1.9	2.0	2.2	2.5
	E3	0.7	0.9	1.2	1.3	1.6	1.9	1.6	2.0	2.2	2.0
	E4	1.2	2.0	1.3	1.3	1.3	1.5	1.7	1.7	2.1	2.3
	E5	1.4	1.3	1.0	1.9	3.3	2.4	2.0	1.9	2.3	3.3