Nimble: Scalable TCP-Friendly Programmable In-Network Rate-Limiting

Vineeth Sagar Thapeta
University of Illinois at Chicago
vthape2@uic.edu

Komal Shinde*
Embedur Systems
komalmshinde94@gmail.com

Mojtaba Malekpourshahraki
University of Illinois at Chicago
mmalek3@uic.edu

Darius Grassi
University of Illinois at Chicago
dgrassi2@uic.edu

Balajee Vamanan
University of Illinois at Chicago
bvamanan@uic.edu

Brent E. Stephens*
University of Utah
brent@cs.utah.edu

ABSTRACT

There is an emerging need for scalable high-performance *in-network* rate-limiting because rate-limiters can be used to provide performance isolation. However, existing approaches to in-network rate-limiting are not scalable or TCP-friendly.

This paper presents the design of Nimble, a new approach to in-network rate-limiting that is scalable, high performance, and TCP-friendly. Nimble uses meters to scalably provide hardware rate-limiting without any dedicated queuing or buffering resources, and Nimble uses ECN-Shaping for TCP-friendly rate-limit enforcement. Nimble also introduces the first algorithm for configuring in-network rate-limiters to enforce network-wide isolation policies.

Through a P4 implementation and experiments with a 100Gbps Barefoot Tofino switch, we find that Nimble is immediately usable and can operate even with high bandwidth rate-limits without needing to recirculate packets or rely on hardware packet generators to generate token refill packets. This overcomes the scalability limitations of prior approaches. Experiments with Apache and Redis show that Nimble can reduce application-level latency by an order of magnitude when compared to not using in-network rate-limiting, and ns-3 simulations demonstrate that Nimble behaves well in larger clusters. We find that Nimble can scale to 100K rate-limiters per-switch when implemented on a Barefoot Tofino switch, and our new rate allocation algorithm reduces rate-limiter updates by a factor of 10x–24x and improves network utilization by 24%.

CCS CONCEPTS

• **Networks** → **Programmable networks**; **Data center networks**; **Network resources allocation**; **In-network processing**.

ACM Reference Format:
Vineeth Sagar Thapeta, Komal Shinde, Mojtaba Malekpourshahraki, Darius Grassi, Balajee Vamanan, and Brent E. Stephens. 2021. Nimble: Scalable TCP-Friendly Programmable In-Network Rate-Limiting. In *The ACM SIGCOMM Symposium on SDN Research (SOSR) (SOSR '21), October 11–12, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3482898.3483361

*Work done while at University of Illinois at Chicago

SOSR '21, October 11–12, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-9084-2/21/10…$15.00
https://doi.org/10.1145/3482898.3483361

1 INTRODUCTION

Rate-limiting is becoming increasingly important in many different types of networks, including data centers, IXPs, and WAN networks. This is because it enables network operators to create *bandwidth reservations* and subdivide network bandwidth across different competing traffic classes (TCs) to avoid network congestion. The benefits of using rate-limiters like this include lower latency, higher throughput, and improved predictability [10, 35, 36, 38, 47, 49, 50, 61]. For example, rate-limiters can ensure that latency-sensitive applications experience predictably low network latency. Similarly, rate-limiters can ensure that competing tenants cannot gain more than their fair share of network bandwidth by opening more connections or running non-standard TCP stacks.

The focus of this paper is on *in-network* rate-limiting, which is needed in a variety of different types of networks, including IXP, data center, and WAN networks. However, there are limitations to all existing approaches to in-network rate-limiting. To address this, this paper presents the design of Nimble, a new approach to in-network rate-limiting that is high performance, scalable, TCP-friendly. Further, Nimble addresses difficulties associated with updating rate-limiters as flows start and stop, and Nimble is implementable on today's commodity programmable switches.

Nimble provides in-network rate-limiting because this avoids the security and precision limitations of performing rate-limiting at servers attached to the network edge. In networks like IXPs and transit WANs where hosts are untrusted, edge-based rate-limiting cannot prevent tenants from gaming the system by opening more connections or using non-standard TCP implementations, and in-network rate-limiting overcomes this limitation. Additionally, edge-based rate-limiting may require dynamic coordination across many different rate-limiters as communication patterns shift [38]. In contrast, enforcing rate-limits inside the network instead of at the edge can be simpler and more precise.

Unfortunately, prior approaches to in-network rate-limiting have limitations with respect to performance, scalability, and TCP-friendliness. For example, software rate-limiters struggle to drive 100Gbps line-rates and incur tens of microseconds of latency, which is not acceptable in data center networks [18, 21, 55]. In contrast, hardware in-network rate-limiting that uses queues to perform shaping may also suffer from scalability limitations. For example, most modern data center switches only have a limited number of queues

per port (*e.g.*, 32 or fewer [25, 52]), while production networks run orders of magnitude more applications and flows [41, 48, 55]. Many approaches to in-network rate-limiting drop packets, and this is problematic in data center environments because it hurts TCP performance [8, 67].

To overcome these limitations, Nimble utilizes *meters*, a primitive that assigns a color to a flow when it exceeds a configured rate and burst allowance [29, 30]. Because meters do not perform queuing, they have low resource overheads. However, meters as they are traditionally used are not sufficient to implement a network-wide rate-limiting scheme. Instead, there are challenges that must be overcome with respect to using meters as a primitive to provide rate-limiting, and there is also a need for a network controller that can configure meters appropriately to enforce a high-level isolation policy.

Nimble fundamentally rethinks how meters are implemented, how meters are used to perform rate-limiting, and how network controllers configure meters. We introduce a new design for implementing meters on P4 switches (logical meters). We use *ECN-shaping*, a TCP-friendly approach to enforcing rate-limiters that uses ECN marking to enforce rate-limits without dropping packets or performing per-traffic class packet buffering. Additionally, we introduce a new algorithm for computing in-network rate-limits according to high-level policies that results in fewer rate-limit updates as flows start and stop than edge-based rate-limiting.

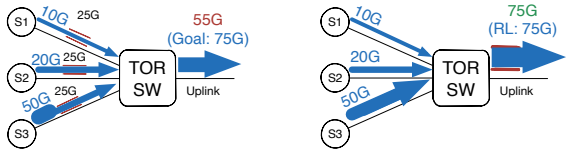
There are three primary contributions of this paper:

First, Nimble introduces *logical meters*. Logical meters are a P4 implementation of meters, and this allows devices that do not have hardware support for meters but do have hardware support for P4 (*e.g.*, PISA switches [12]) to still be able to benefit from Nimble. Switches that support P4 have counters that are used for monitoring, and we show that these counters can be repurposed to implement logical meters with low overheads on switches that support P4.

Second, Nimble keeps track of logical meter occupancy without the need for recirculation, without the need to generate new packets per meter using hardware clocks, and without the need for physical queueing resources. This overcomes the limitations of previous approaches that struggle keep up with the line rates or scalability requirements [16, 19, 66]. Nimble achieves this by introducing a novel way to perform approximate multiplication on PISA switches.

Third, Nimble introduces *a new algorithm* for computing in-network rate-limits (meter configurations) for a network of switches that enforce network-wide isolation policies. Nimble supports all of the same types of policies as the DCB standard provides for individual devices [19] except that Nimble supports network-wide policies with far many more traffic classes, priorities, and weights (100K). Because of the potential asymmetries in topologies, routing, and placement, naively configuring individual rate-limiters according to the same policy is not sufficient to ensure the policy will be enforced. This algorithm overcomes this by computing asymmetric rate-limiter configurations that provide global policy enforcement.

Through experiments with real world applications, we find that Nimble is scalable and can effectively enforce rate-limits with high throughput and low latency. Experiments with a P4 implementation of Nimble that runs on a Barefoot Tofino switch [11] and provides ECN-Shaping, logical meters, and meters demonstrate that scalable TCP-friendly hardware in-network rate-limiting is possible today.



(a) Edge-based Rate-Limiting **(b) In-network Rate-Limiting**

Figure 1: Example of how in-network rate-limiters can provide more precise rate-limiting and achieve higher network utilization than edge-based rate-limiters.

We find that Nimble can scale to a total of 100K meters *per-switch*. Using ns-3 simulations, we show that Nimble is scalable and beneficial across different topologies and protocols. Our experiments also show that Nimble reduces rate-limiter updates by 10x–24x when compared to edge-based rate-limiting, and our algorithm correctly enforces network-wide isolation policies while *improving network utilization by 24%* when compared to dynamic local policy enforcement.

2 MOTIVATION

Across a variety of different types of networks, including Internet eXchange Points (IXPs), data centers, Content Distribution Networks (CDNs), and Wide-Area Networks (WANs), there is a need for *in-network* rate-limiting. However, all existing approaches to in-network rate-limiting suffer from key limitations, and it is difficult to correctly configure rate-limiters in a multi-hop network.

2.1 The Need for In-Network Rate-Limiting

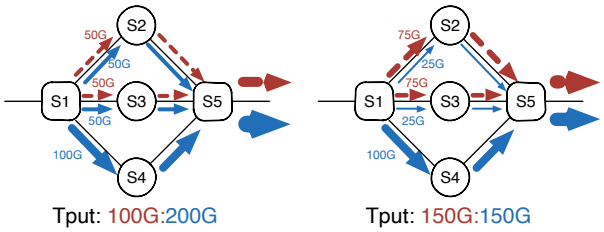
In-network rate-limiting can be used to enforce network-wide (global) performance isolation policies, and it is more secure, more precise, and results in fewer updates when compared with edge-based rate-limiting.

Ideally, it would be possible to ensure that competing tenants and applications are isolated and share network resources like bandwidth and buffer space fairly. Unfortunately, congestion control algorithms do not provide performance isolation because they converge to per-flow fairness. Similarly, relying on TCP for isolation is not secure. If one application opens more connections than another or uses a non-standard TCP implementation, it will consume more than its fair share of switch buffer space and bandwidth, causing packet losses, reduced throughput, and increased latency for other applications [10, 35, 36, 41, 46, 53, 61, 67, 68].

In-network rate-limiting versus edge-based rate-limiting: Although prior work has established the benefits of edge-based rate-limiting [10, 35, 36, 38, 47, 49, 50, 53, 61], there is also a need for rate-limiters inside the network to enable performance isolation in scenarios where edge-based rate-limiting is either not possible or is inefficient.

For example, in an IXP, the end-hosts attached to the network are untrusted because they are owned and operated by the clients of the IXP. Because the edge is untrusted, these networks need in-network rate-limiters.

However, even in networks where the edge is also controlled by the network operators, which includes networks like data center networks, private WAN networks, and CDNs [31, 32, 34, 38, 49, 50, 63], in-network rate-limiting can enable more precise rate-limiting



(a) Local Policy Enforcement **(b) Global Policy Enforcement**

Figure 2: Example of how strictly local enforcement can fail to enforce a global network sharing policy that the red and blue TCs should have a 1:1 share of total network bandwidth.

and more efficient utilization of available network bandwidth. For example, it is often better to use a single in-network rate-limiter than many edge-based rate-limiters.

Figure 1 illustrates this by comparing using multiple edge-based rate-limiters and using a single in-network rate-limiter to implement a policy where the traffic from three servers traversing an uplink must not exceed 75Gbps and the demands of servers 1, 2, and 3 are all different. As Figure 1a shows, naively using edge-based rate-limiters to enforce a 25Gbps rate limit at each server leads to underutilization as only 55Gbps of traffic is sent. In contrast, Figure 1b shows that in-network rate-limiting can precisely enforce this policy with a single rate-limiter placed on the output port of the shared uplink. Although it is possible to overcome the limitations of edge-based rate-limiting by measuring network demand and dynamically adjusting rate-limiters, this approach is worse than in-network rate-limiting given dynamically changing traffic because edge-based rate-limiting would potentially require more number of rate-limiter updates than in-network rate-limiter would. In Section 4 we elaborate on further reducing the number of updates in our network controller.

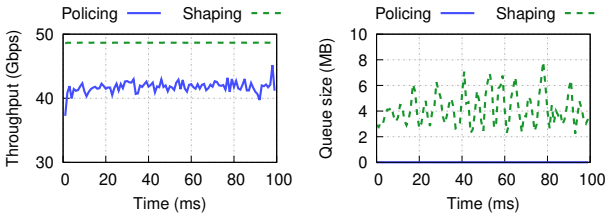
Local versus global enforcement: DCB allows operators to configure per-port rate-limiters for eight total TCs [19]. This is too few TCs for many scenarios. Additionally, this requires network operators to configure rate-limiters by hand, and this can become arduous error-prone in large networks.

However, it is not immediately straightforward to automate the process of configuring in-network rate-limiters. Given a global network isolation policy, naively configuring every port at every switch to locally enforce this policy may not lead to accurate enforcement. To help this, Figure 2 shows a scenario where local enforcement of a policy that a red and blue TC should equally share the total capacity of the network is not accurate. Because routing is asymmetric, the blue TC gets more throughput with strictly local enforcement. Instead, to globally enforce the policy, it is necessary to install asymmetric rate-limits. As such, there is a need for creating a network controller that can enforce network-wide (global) isolation policies by dynamically computing asymmetric in-network rate-limits for a network of switches.

2.2 Limitations of In-Network Rate-Limiting

In-network rate-limiting can be implemented with software or hardware, and both approaches have key limitations.

2.2.1 Software Rate-Limiters. Through the use of software middleboxes, it is possible to perform in-network rate-limiting [18, 24]. However, this approach is often not viable because of limitations



(a) Aggregate throughput for policing and shaping. **(b) Queue size evolution for policing and shaping.**

Figure 3: Results from an experiment that shows that policing leads to poor TCP throughput because of packet loss and that shaping requires an excessive amount of switch buffering capacity to avoid dropping packets.

related to low throughput, low precision, high latency, high CPU overhead, poor locality, and wasted network bandwidth. While the first four limitations are obvious, poor locality and wasted bandwidth result from routing traffic from inside the network back to the edge (server). For example, prior work has found that software-based rate-limiters struggle or fail to drive 100Gbps line-rates, incur tens of microseconds of latency, and do not provide enough precision over inter-packet spacing to avoid microbursts [18, 21, 35, 49, 50].

2.2.2 Hardware Rate-Limiters. Hardware rate-limiters are high performance and typically operate at line-rate without incurring additional processing latency. However, today's hardware rate-limiters are not scalable, and they do not interact well with TCP.

Scalability: Switches typically use *virtual queues* to perform rate-limiting [16, 19, 47] by dedicated queuing and buffering resources to each TC. Switches that implement the Data Center Bridging (DCB) standard only support 8 different TCs per port [19, 62], and prior work has reported that currently available commodity Ethernet switches support at most 32 virtual queues per-port [25]. Further, even the state-of-the-art approaches that are not yet available in practice only provide thousands of different TCs [54, 58].

Today's networks need many more traffic classes. For example, Google reports that their data center network hosts thousands of different services [55], and Microsoft reports that over 90K servers are used to host both Bing and batch analytics [33]. Kumar *et al.* report that a link in Google's WAN can be used by 400K different job-level flow groups [38].

TCP-Friendliness: Existing approaches to in-network rate-limiting are not TCP-friendly. There are two ways to react when an in-network rate-limit is exceeded: policing, which drops packets, and shaping, which buffers packets then releases packets at the appropriate time.

Policing is harmful to TCP performance, especially when considered at the tail [16, 22, 27, 64, 67]. This is because dropping a single packet can cause TCP to suffer from a retransmission timeout (RTO). Additionally, Flach *et al.* recently found that traffic policing is particularly harmful to video streaming playback quality of experience [22].

Although shaping avoids dropping packets, shaping can cause unacceptable increases in network latency because the number of buffered packets can grow large. Congestion can be persistent because this approach does not restrict the rate at which end-hosts inject packets into the network.

To demonstrate the pitfalls of policing and shaping, we ran ns-3 simulations with 10 source servers sending to one sink server, creating a 10-to-1 incast. The servers are connected to one switch via 100 $Gbps$ links, and there is a 50 $Gbps$ rate-limit on the output port that connects to the sink; the servers use DCTCP. Figure 3a shows aggregate throughput over time and Figure 3b shows the queue size on the output port that connects to the sink for both policing and shaping.

First, we observe that policing cannot achieve full throughput. It achieves about ∼41Gbps instead of the full 50Gbps. This is because policing relies on packet drops to enforce the rate limit, and packet drops cause end-hosts to suffer from TCP timeouts and reduced congestion windows. In contrast, while shaping achieves closer to the expected throughput of 50Gbps, it incurs substantial queuing. The queue lengths in this experiment vary in the range of 2MB–8MB, and this amount of queuing can lead to significant latency. For example, 4MB of queuing at a 100Gbps line-rate adds 336μs.

2.2.3 Hardware Meters. Meters are another hardware primitive supported by some switches that can monitor the rates and burst allowances of different packet streams and mark the packets either green, yellow, or red depending on if these allowances are exceeded [29, 30]. Although meters are scalable because they can be implemented with counters, they have problems with respect to isolation and TCP-friendliness.

The recommended use-case for meters is to discard all red packets (policing), forward all yellow packets as best effort, and forward green packets with a low drop probability. In this scenario, dropping all red packets is policing, which is not TCP-friendly. Additionally, because all yellow packets are treated the same with respect to queuing, it is not possible to guarantee performance isolation across competing TCs.

3 NIMBLE OVERVIEW

Nimble is a new system that utilizes scalable hardware in-network rate-limiting to dynamically enforce network-wide performance isolation policies. There are three primary components to Nimble. First, switches provide rate-limiting by using meters with ECN-shaping or policing. Second, network operators specify local and global isolation policies. Third, a network controller monitors traffic patterns and dynamically configures rate-limiters for the active TCs on the appropriate switch ports to enforce the policy. The rest of this section discusses these components in more detail.

3.1 Rate-Limiting Primitives

Nimble utilizes meters to scalably monitor TC rates inside the network, and Nimble utilizes both policing and ECN-shaping to enforce in-network rate-limits. Because meters can be implemented in hardware with only SRAM and ALUs, they are more scalable than virtual queues.

3.1.1 Meters. Switches in Nimble support one rate three color meters [29] and the two rate three color meters [30]. A controller configures a single rate meter by specifying a three tuple of a rate and two burst sizes ((R, B_{yellow}, B_{red})). A controller configures a two rate meter by specifying two rate and burst size tuples ((R_{yellow}, B_{yellow}), and (R_{red}, B_{red})). With both the single and two rate meters the meter internally maintains a token bucket for each colors' rate and burst allowance. If the red bucket is empty, the packet is colored red. If the red bucket is not empty but the yellow is, the packet is colored yellow. Otherwise, the packet is colored green.

Nimble supports two types of meters: native meters (NM), and logical meters (LM). When Nimble is used with switches with native support for meters, NM should be used because it likely has lower overheads. However, LM is useful because it provides support for meters on devices that support P4.

3.1.2 ECN-Shaping. ECN-Shaping performs ECN marking based on the color of a meter. This enables TCP-friendly rate-limiting, avoids packet buffer overflows, and overcomes the limitations of both shaping and policing. DCTCP and DCQCN both use ECN marking to cause end-hosts reduce their transmission rate to avoid buffer overflow. With these algorithms, switch queue buffer occupancy converges to the ECN marking threshold [8, 69, 70]. In effect, ECN-Shaping causes end-hosts to modify their congestion windows to converge to the rate determined by an in-network meter. This makes it possible to enforce rate-limits without dropping packets or using queuing resources.

To understand why it is possible to enforce rate-limits without either dedicated queuing resources or packet pacing, it is useful to revisit the equation that is used to update congestion windows in DCTCP: $cwnd \longleftarrow cwnd \times (1 - \alpha/2)$ where α is the fraction of bytes that were ECN marked. Importantly, this equation is independent of the network RTT. As long as the same packets are marked by the switch, the end-hosts will reduce their congestion windows in the same way regardless of how packets are queued or paced.

3.2 Policies

Network operators use Nimble to specify performance isolation policies. Nimble uses a controller to dynamically compute appropriate rate-limits to provide work-conserving policies. Policies in Nimble allow for different TCs to be assigned different priorities, weights, and rate-limits. Because Nimble supports many more traffic classes (100K), it can implement policies not possible with DCB.

Nimble supports both local and global policies. Local policies allow operators to exert expert control over specific links like WAN uplinks. Global policies allow operators to specify policies that are enforced across a network. This ensures that the entire network is shared according to high-level policies without requiring operators to configure rate-limiters for every switch port in the network, which would be arduous and error-prone.

Policies in Nimble are specified at the granularity of the TC. To specify a policy for a TC in Nimble operators specify two things: 1) a packet classifier, and 2) the isolation policy.

Packet classification: To configure Nimble, a network operator starts by providing a mechanism to assign a packet to a traffic class. Since Nimble is designed to be used with P4 switches, any packet header fields can be used. However, for simplicity and compatibility with existing protocols, our current design uses a packet's 7-tuple to find the TC for a packet. The 7-tuple lookup maps `(SMAC, DMAC, SIP, DIP, PROTO, SPORT, DPORT)` → `(TC)`, where SMAC, SIP, and SPORT and DMAC, DIP, DPORT are the source and destination MAC addresses, IP addresses, and ports, respectively, PROTO is the IP protocol field, and TC is the

traffic class. For flexibility, any of the input fields may be a wildcard. This also enables 5-tuple (no MAC addresses) and 2-tuple (either only MAC or IP addresses) lookups. Also, traffic classification can be combined with routing for efficiencies sake so that one lookup table can be used to find both the output port and traffic class for a packet at the same time.

Isolation Policies: For each TC, operators must configure a global priority and weight, and they can also configure a rate-limit. When applied together, these per-TC configurations are used by the controller to dynamically configure rate-limits across the network. Additionally, the operator may also optionally configure local policies with respect to specific input and output ports on switches for a TC, and these policies are applied before global policies.

Global and local policies are specified as follows:

```
GBL_Policy(TC, Weight, Priority, RL, ETYPE)
LCL_Policy(LOC, TC, Weight, Priority, RL, QTYPE,
    ETYPE)
```

`TC`, `Priority`, and `Weight` are all integers. `TC` is a number previously associated with a packet classifier. `Priority` and `Weight` configure how this TC is treated when it is competing for bandwidth with another TC. When allocating the network bandwidth, the controller will allocate bandwidth to the highest priority TCs first, and TCs of the same priority will receive a share of available bandwidth proportional to their relative weights. Because TCs with a higher priority will be strictly allocated bandwidth before lower priorities, this intentionally can lead to starvation. If this is undesirable, rate-limits can be applied to high priority flows.

`RL` is a 3-tuple that defines a rate-limit: `(Rate, Burst, RLType)`, where `Rate` is a rate in bits per second, `Burst` is a burst tolerance in bytes, and `RLType` is either soft, which means that the TC can receive less than the limit or hard, which means that a rate-limit for the TC will be configured exactly at the specified rate. To help avoid overallocation, the controller generates a warning if the sum total of rate-limits exceeds the available bandwidth for any port.

For local policies, `LOC` specifics a location in terms of a switch and either one or both of an input port and output port on that switch. `QTYPE` defines how the rate-limit will be enforced. The options for `QTYPE` are virtual queue (VQ), native meters (NM), and logical meters (LM). Although VQ is not scalable, this allows operators to utilize them when they are available. This also allows for hierarchical policies where a group of TCs using meters are also in another TC sharing a single VQ. Finally, `ETYPE` specifies if the rate-limit should be enforced with policing, ECN-shaping, or shaping. However, as Table 1 shows, not all queuing and enforcement types can be used together because shaping is not possible with NM or LM. Even though policing can lead to poor TCP performance at the per-flow level, it is included because it is still useful in environments where ECN marking is not possible and per-flow performance is not critical, *e.g.*, ISPs.

3.3 Controller

The controller in Nimble dynamically monitors traffic patterns and then uses this information to compute and install rate-limits at the appropriate switches in the network. This controller utilizes the same general control loop to monitoring traffic patterns as controllers in similar systems that perform edge-based rate-limiting [9, 10, 35, 36, 38, 46, 53].

Queuing and Rate Detection Type	Enforcement Type		
	Shaping	Policing	ECN-Shaping
Virtual Queues	✓	✓	✓
Logical Queues	✗	✓	✓
Meters	✗	✓	✓

Table 1: The different configurations supported by Nimble.

First, the controller learns about the active tenants, applications, and flows using the network either through network monitoring or coordination with cluster management systems. Although the exact specifics of this monitoring is outside the scope of this paper because prior work has already demonstrated its feasibility, it is straightforward to perform through integration with either virtual switches [42] or In-Network Telemetry (INT) [12, 40, 45, 59].

Next, the controller combines the operator specified policies with knowledge of traffic patterns to compute appropriate meter configurations. As long as the aggregate sum of all of the rate-limits of the TCs using a link in the network are below the network line-rate, this approach can guarantee that packets are not dropped due to congestion and that competing traffic classes are isolated. Additionally, the controller in Nimble is designed to reduce the total number of rate-limiters that need to be configured. This increases scalability, and, when combined with a new approach for consistently updating multiple meters on a single switch, this can lead to more precise policy enforcement during updates.

4 NIMBLE DESIGN

Nimble introduces a new switch program design to support meters and a new algorithm for configuring these meters. This section discusses these components of Nimble in more detail.

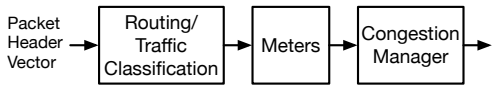
4.1 Switch Program Designs

The switch programs in Nimble use a pipelined design. The modules that compose the pipeline only feed information forward because this enables Nimble to be expressible in P4 [4] without requiring any recirculation. This is important to ensure that Nimble operates at full line-rate.

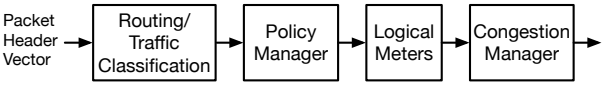
Nimble is also designed to be integrated into other P4 programs, *e.g.*, `dc.p4` [57]. Because P4 does not support libraries [60], Nimble is implemented as separate *micro-pipelines* that can be manually integrated into other programs. Figure 4 shows the switch micro-pipelines that can be used to perform in-network rate-limiting with either NM or LM. The different modules in these pipelines behave as follows:

Routing/Traffic Classification: This module determines the output port and traffic class for each packet `(OUTP, TC)`. Although routing is modular and can be changed as needed to integrate with another switch program, our current design uses a packet's 7-tuple to find this information.

Native Meters: This module maps the current TC, update epoch, input port, and output port passed forward from the routing module to a meter, which then outputs a color. The control plane is responsible for programming the appropriate rates for the meter. Also, because this module uses a controller configured epoch to find the appropriate meter, this allows rates for multiple TCs to be consistently updated.



(a) Native Meters



(b) Logical Meters

Figure 4: Designs for different ways of performing rate-limiting in Nimble.

Congestion Manager: The congestion manager provides per-TC decisions on how to mark and drop packets after they has been assigned a color. To support DCTCP and TCP with ECN support, this module ECN marks all yellow packets. Because DCQCN requires that switches perform probabilistic marking, this module uses RED [23] to probabilistically mark yellow packets. Because strict policing is harmful to tail TCP flow performance, this module uses RED to drop yellow packets from TCs using TCP without ECN support. To ensure isolation, all red packets are dropped by default. Discussion centered around deployment can be found in 4.3.

Policy Manager: To support LM, this module looks up the appropriate rate-limiter index, rate, and burst tolerances given the current packet's TC, input port, and output port. With LM, the rate that is looked up by the policy manager may be configured from either the control plane or the data plane, which can lead to lower and more predictable latency. Programming a rate limiter from the data plane is accomplished by using a lookup table to save the rate received from data in special control packets to a register in an array.

Logical Meters: Logical Meters emulate virtual queues by using counters (registers) to track the number of bytes that would be enqueued in a virtual queue that drains at a rate R. When a packet arrives for a logical meter(LM), the length of the current packet is added to the register unless the packet will be dropped, and the number of bytes that would have been drained from a virtual queue in the amount of time since the last queue update are subtracted from the register ($LM = LM + PktLen - DrainB$). Given a rate R from the policy manager, the drain bytes is computed as R times the amount of time since the last packet using this rate-limiter ($DrainB = R * \Delta T$), where ΔT is computed by taking the difference of the current timestamp and the timestamp that was saved in a register the last time this rate-limiter was used.

Algorithm 1: Multiply Table Population Algorithm

Input: (s,b) // Significant bit and number of bits
Result: table T
$V = \{i | i \in [2^b, 2^{b+1} - 1]]\}$
$T = \{i | i \in [0, 2^b - 1]\}$
while $\forall i \in V$ **do**
  for *j=1;*$s \leq$ *j;j++* **do**
    $i \xleftarrow{j} x$ *// Left shift with don't care (x) as input*
    $T.add(i)$

Since most programmable switches do not support multiplication [12, 56], we use a TCAM lookup table to approximately compute the result of this operation by matching only on the b most significant bits of ΔT. Specifically, Algorithm 1 shows how to populate the table. For a variable I with i bits and b_i significant bits, this algorithm generates $e_i = (i - b_i + 2)2^{b_i - 1}$ entries. The average of the numbers in a group is used for the output of the table to minimize error. This algorithm creates entries that cover ranges all with the same expected percent error.

4.2 Network-Wide Policy Enforcement

Nimble introduces a new algorithm that allows a network controller to dynamically enforce isolation policies across a network of switches. By dynamically configuring in-network rate-limiters, this allows Nimble to achieve isolation across TCs, and Nimble supports policies that provide priorities, weighted fair sharing, and rate-limits.

Nimble performs dynamic network-wide rate-limiter configuration because this overcomes the limitations of naively enforcing policies locally at each switch. Due to potential asymmetries in topology, routing, and job placement, local enforcement cannot accurately enforce policies. Locally policy compliant configurations can still waste bandwidth if a traffic class is limited to a lower rate at another switch.

To ensure that Nimble is compatible with existing networks and routing algorithms, the controller in Nimble does not interfere with routing and instead takes the current set of active routes as an input. For example, this is necessary to support networks that use ECMP for routing.

The input to the algorithm is the network policy, the network topology, and the set of *routing paths* for each TC. This algorithm considers traffic patterns at the coarsest possible granularity. Specifically, a *routing path* is a list of switch output ports used by any flow belonging to a TC. Finer granularities like the TCP flow would lead to more churn. The output of this algorithm is a set of rate-limiter configurations for all of the switches in the network.

Algorithm 2 shows our complete algorithm. Our insight is that TCs only need be rate-limited at the most bottlenecked switch on the paths they use, so the core loop of this algorithm operates by iteratively finding the most congested output port in the network and allocating bandwidth according to the policy (Function `find_bottleneck`).

Determining which switch port is the bottleneck depends on the network policy. To support priorities, this algorithm assigns bandwidth to the current highest unassigned priority for every bottleneck port in the network before considering the TCs in the next priority. Weights are supported by allocating the unassigned bandwidth of a bottleneck port in proportion to the relative weights of the active TCs at the current priority level. Rate-limits are supported by capping the maximum rate that will be allocated to a TC per-port.

Because this algorithm does not overallocate the bandwidth of a port, it ensures that congestion cannot overload any port. Additionally, by configuring the burst allowances of a one rate three color meter, the amount of buffered packets can be bounded. By default, Nimble sets a low yellow threshold (*e.g.* 16KB) because this threshold is used to perform ECN-shaping or random dropping (RED). To set the red threshold for a meter, the operator configures a maximum per-port buffer allocation, Nimble uses weights to subdivide this across TCs.

Algorithm 2: Rate Allocation to Traffic Classes

Inputs:
Set of paths: $W = \{w_1, w_2...\}$
Set of traffic classes: $C = \{c_1, c_2...\}$
Set of ports: $P = \{p_1, p_2, ...\}$
Output:
$r(c,p)\ \forall\ c \in C, p \in P$; initially $r(c,p) = 0$
$r(w)\ \forall\ w \in W$; initially $r(w) = 0$
Definitions:
$p.rate$: line rate of the output port p
C_a := set of classes whose rates have been assigned
$p.classes()$:= set of active classes in port p
$p.paths(c)$:= set of active paths in port p and class c
$used(p,c)$:= capacity already used in port p for class c := $\sum_{w \in p.paths(c)} r(w)$
$used\_total(p)$:= total capacity already used in port p := $\sum_{c \in p.classes() \cap C_a} used(p,c)$
$p.unassigned\_classes()$:= set of traffic classes in port p whose rates have not been assigned := $p.classes() - C_a$
$c.assigned\_paths()$:= set of paths in traffic class c whose rates have been assigned
$p.unassigned\_paths(c)$:= set of paths in traffic class c in port p whose rates have not been assigned
:= $p.paths(c) - c.assigned\_paths()$
Initialization:
$C_a = \phi$
$\forall c, p : r(c,p) = 0,\ c.assigned\_paths() = \phi$
$\forall w : r(w) = 0$
Function `main()`**:**
  while $P \neq \phi$ **do**
    $find\_bottleneck(bottleneck\_rate, bottleneck\_p, bottleneck\_c)$
    for $\forall w \in bottleneck\_p.unassigned\_paths(bottleneck\_c)$ **do**
      $r(w) = bottleneck\_rate$
      $r(c,p) += r(w)$
      $c.assigned\_paths().add(w)$
    if $|c.assigned\_paths()| == |c.paths()|$ **then**
      $C_a = C_a \cup \{c\}$
    if $p.unassigned\_paths(c) == \phi$ **then**
      $P.remove(p)$

`/* Find the bottleneck rate based on high-level policy */`

Function `find_bottleneck()`**:**
  for $\forall p \in P,\ c \in p.unassigned\_classes()$ **do**
    Calculate "$class\_rate$" by divvying the unallocated rate ($p.rate - used\_total(p)$) as per policy using *priorities*, *weights*, and *rate limits*
    for $\forall w \in p.unassigned\_paths(c)$ **do**
      Calculate "$path\_rate$" by divvying the "$class\_rate$" among $p.unassigned\_paths(c)$
      if $path\_rate < bottleneck\_rate$ **then**
        $bottleneck\_rate = path\_rate$
        $bottleneck\_p = p$
        $bottleneck\_c = c$
  return bottleneck_rate, bottleneck_p, bottleneck_c

Operators of large networks design high-level network policies in order to balance the needs of different applications. For instance, operators typically prioritize traffic classes from latency-sensitive applications ahead of classes from bandwidth-hungry applications but set rate limits for the former to avoid starving the latter. It is also common for operators to share the network capacity among some set of applications in a weighted-fair manner based on many factors (e.g., price tier, business considerations). While designing such policies is beyond the scope of this paper, we argue that using a combination of strict priorities, weighted fair-share policies, and rate limits, we can support a broad range of typical policies. Our goal is two fold: (1) *Policy compliance*: To allocate rates to traffic classes such that the allocation is policy-compliant (i.e., at each bottleneck link, the link capacity is shared in accordance with the global policy). (2) *Utilization:* To allocate rates to traffic classes using global knowledge of bottleneck links so that no traffic class is allocated more capacity than its bottleneck rate. We evaluate these two aspects of our design in Section 6.3.2 and compare our allocation to a purely switch-local allocation for a policy that combines priority and weights.

To minimize the overhead, Nimble invokes this algorithm only when rates have to be recomputed, which happens when: (1) network policy changes. (2) links/switches fail, (3) tenants are added/removed, and (4) tenant routing paths are added/removed. Additionally, this algorithm results in fewer rate-limit updates than a comparable system that performs edge-based rate-limiting (*e.g.*, BwE [38]). Because Nimble places rate-limits at the switch that has a bottleneck port, there are times when in-network rate-limits do not need to be updated while edge-based rate-limiters would (Figure 1).

Additionally, this algorithm supports incremental rate-limiter updates. For correctness, we rerun the entire algorithm on updates. However, because this algorithm is deterministic, the configuration of many rate-limiters will be unchanged after an update. The controller takes advantage of this by tracking the delta between configuration epochs and only issuing updates for rate-limiters that have changed. Further, when updating rate-limiters, Nimble consistently updates all of the rate-limiters on the same switch at once by using versioning in the switch program to find the appropriate meter.

4.3 Discussion

Logical meters enable ECN shaping, which works efficiently with TCP and achieves superior performance than both policing and shaping (Figure 3). However, not all TCP stacks support ECN marks and the congestion response of ECN-supported TCP stacks vary.

Recall from Section 4 that logical meters ECN mark all yellow packets and drop all red packets. To support TCP stacks that do not react to ECN marks, Nimble emulates Random Early Detection (RED) by *probabilistically* dropping yellow packets. Therefore, logical meters enable TCP stacks that do not support ECN to perform better than both naive policing and shaping. If stacks require probabilistic ECN marking (e.g., DCQCN), we can configure logical meters to probablisitically mark yellow packets as well (Section 4.1). Because logical meters fall back to policing (i.e., dropping all red packets) when everything else fails, malicious/buggy TCP or UDP senders cannot exceed their policy-compliant allocation (Figure 6 demonstrates our stronger isolation between UDP and DCTCP). This

serves as a *fail-safe* mechanism and enforces a stronger isolation between traffic classes than existing mechanisms.

In addition to enforcing rates, the controller in Nimble also bounds the amount of packet buffering at switches by configuring burst sizes. Thus, Nimble is compatible with delay-based congestion control algorithms such as TCP Vegas and QUIC. Because traffic from one traffic class does not lead to increased queuing, Nimble enables delay-based TCP variants to safely co-exist with more aggressive loss-based algorithms like TCP Cubic, which is not possible without the stronger isolation provided by logical meters. Fortunately, this stronger isolation also enables TCP stacks that respond differently to ECN marks (e.g., DCTCP vs. legacy ECN) to co-exist without exceeding their allocations. Additionally, the congestion manager in Nimble can also be easily modified to support rate-based congestion control algorithms like RCP [20] or PERC [37].

Unlike datacenters where fine-grained information about routing paths and traffic classes are available, WANs may only expose coarse-grain information per policy (i.e., peering agreements between ISPs). Nevertheless, Nimble would still calculate policy-compliant allocations and enforce them efficiently in WANs. Operators could use monitoring systems such as UnivMon [40] to identify traffic classes and feed this information to our rate allocation algorithm. While designing such mechanisms for WANs is beyond the scope of this paper, we think it is possible to deploy Nimble outside datacenters by tracking traffic classes and routing paths at a coarser granularity and by using network monitoring systems.

5 METHODOLOGY

Nimble is implemented in P4 [4] and in the ns-3 simulator [7][1]. This allows us to perform two different types of experiments to evaluate Nimble: Experiments on a local cluster with a commercially available PISA switch and experiments with large scale simulations.

In the cluster experiments, P4 programs are compiled and run on a Barefoot Tofino [11] Wedge 100BF-32X Ethernet switch with a line-rate of 100Gbps. The cluster also contains four servers each with an 8-core/16-thread Intel Xeon 1.80GHz CPU and 64 GB of memory. These servers run Ubuntu 18.04, and they use a 100Gbps Mellanox ConnectX-5 [43] NIC to connect to one port of the switch. We developed a gRPC-based control plane client to dynamically program various rate limiters on the switch. We also developed Thrift client programs to interact with the switch.

We assess Nimble using a variety of real-world applications in the cluster experiments. To demonstrate the correctness of Nimble, we use network benchmarking tools like iPerf [3] and sockperf [6] because they provide precise controlled network traffic. We also use a number of significant data center applications like Apache [1] and Redis [5] to profile Nimble's behavior with real throughput-sensitive and latency-sensitive applications competing in the network. We use ab and redis-benchmark to evaluate these applications.

Next, we perform ns-3 simulations [7] to demonstrate that Nimble works on larger topologies as well as to perform a deep-dive into the workings of our system. We implemented Nimble in the ns-3 switch model and our implementation captures all aspects from Figure 4, including approximate multiplication using Algorithm 1. Further, we also implemented the controller that programs rate limit values in switches (Algorithm 2). In our simulations, the hosts are connected by 100 Gbps links with a link delay of 1 μs. We set the per-port buffering capacity to be 400 KB. The ECN threshold is set at 20% of the buffer size, as recommended by the DCTCP [8] paper for physical queues as well as logical queues. Experiment-specific aspects (workload, topology, TCP versions) are described in Section 6.

[1]The code can be found in this github repository.

6 EVALUATION

This section presents results from experiments and a scalability analysis performed on a local testbed with a Barefoot Tofino switch [11] (Section 6.1 and Section 6.2). Finally, it presents results from experiments with the ns-3 simulator (Section 6.3).

6.1 Testbed Experiments

Through testbed experiments, we demonstrate that Nimble is feasible on today's hardware, accurate, and TCP-friendly. To do this, we perform experiments where there is only a single active TC to demonstrate accuracy, and we perform experiments with multiple TCs to demonstrate that Nimble effectively isolates different competing applications and supports a variety of different congestion control algorithms.

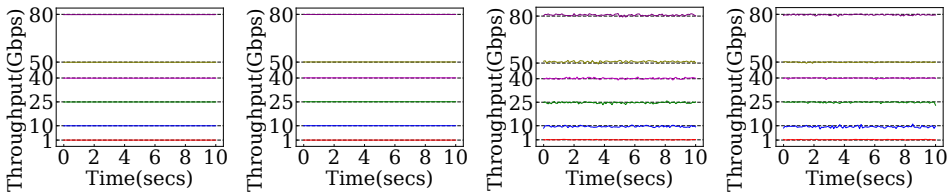
6.1.1 One TC Experiments. We start by evaluating Nimble given a workload where all traffic belongs to a single TC and is between two machines. In these experiments, we use `iperf3` to generate traffic from a client to a server at full line-rate, and we enabled DCTCP on both the client and server. These experiments use 16 parallel `iperf3` clients each with 8 parallel connections.

Then, we program the switch to provide a range of different in-network rate-limits from 1Gbps–80Gbps for four different approaches to rate-limiting that are supported by Nimble: shaping with virtual queues (VQ), ECN-shaping with virtual queues (VQ(ECN)), ECN-shaping with native meters (NM), and ECN-shaping with logical meters (LM).

Figure 5 shows the result from this experiment. As expected, VQ accurately enforces rate-limits despite having problems related to latency and scalability. From the VQ(ECN) results, we find that ECN-Shaping is able to improve enforcement accuracy over traditional shaping. While the VQ and the VQ(ECN) primitives administer the assigned rates more smoothly, they are not scalable to a large number of flows. They are a very scarce resource on the Tofino ASIC. The Nimble primitives achieve similar performance as VQ and VQ(ECN) without the associated overhead of dedicated queues.

Next, Figure 5c and 5d show the performance of NM and LM. These figures show that both logical and native meters can enforce a wide range of different rate-limits. Also, this experiment use a multiply table that groups entries based on the first four significant bits, and a total of 764 multiplication entries were used for all the rates in Figure 5d.

6.1.2 Application Behavior. To demonstrate that Nimble is able to provide bandwidth reservations for competing programs without any dedicated queuing resources, we performed experiments where the 100Gbps line-rate is subdivided amongst competing applications: the Apache httpd web server [1], Redis [5], and iperf [3]. End-hosts run DCTCP, and switches perform ECN-shaping.



(a) Virtual Queue Shaping (b) Virtual Queue ECN Shaping (c) Native Meters (d) Logical Meters

Figure 5: A comparison of the accuracy of four different rate-limiting approaches.

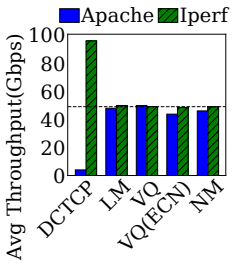


Figure 6: Data center applications

Program	Latency Percentile(ms) Apache 50th,99th		Redis 50th,99th	
DCTCP	450	2032	13.52	20.54
LM	40	238	1.37	4.92
VQ	40	239	2.5	9.32
VQ(ECN)	45	74	2.15	7.79
NM	35	281	1.14	15.69

Table 2: Latency Distribution

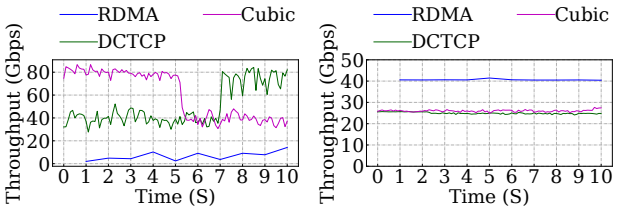
To generate congestion, there is a 2:1 incast to a single server. One of the clients is a malicious UDP client, and the other is web server sending data to the same destination. We assign a 49Gbps rate for the iperf tenant and a 49 Gbps for the Apache tenant. There are 16 UDP clients with 8 parallel connections, and there are 6 instances of ab with 10 parallel connections requesting data from the web server.

Figure 6 shows the results of this experiment. With the DCTCP baseline we observed a large number of request timeouts, and the throughput of the Apache webserver achieves significantly less throughput than its fair share as it is starved by the competing UDP flows. In contrast, all four of the different Nimble configurations provide performance isolation.

Next, we perform an experiment with Redis, and iperf to evaluate if Nimble can provide low latency without any dedicated queuing resources. To generate load, we run redis-benchmark with 50 clients. The results are shown in Table 2. DCTCP does not provide performance isolation, and therefore, suffers from high tail latency, in both Redis and iperf. In contrast, all the other schemes (LM, VQ, VQ(ECN), NM) provide provide performance isolation and achieve about an order of magnitude reduction in tail latency. We perform a deeper analysis of their queuing behavior in Section 6.3. The key takeaway here is that Nimble achieves similar performance as virtual queues without requiring dedicated queues. Thus, Nimble has the potential to scale to a large number of TCs.

6.1.3 Multiple Congestion Control Algorithms. Networks like IXPs and ISPs may carry traffic from various transports and congestion control algorithms. To evaluate this, we generate traffic from three nodes. Each node is assigned its own TC. Node 1 uses TCP Cubic (disabled ECN marking), node 2 uses an RDMA NIC to use DCQCN, and node 3 uses DCTCP. The RDMA traffic has a 40 Gbps rate limit, and the other two transports are each subject to 25 Gbps rate-limits. For DCQCN and TCP Cubic traffic, we perform probabilistic ECN marking and dropping, respectively.

The results from this experiment are presented in Figure 7. Without Nimble, there is not a stable point of sharing between DCTCP and Cubic, and RDMA traffic receives significantly worse than its



(a) Without Rate-limiting (b) With Rate-limiting

Figure 7: Experiment with Different Congestion Control Algorithms

Program # Limiters	7-Tuple 100K/4K	5-Tuple 100K/4K	2-Tuple 100K/4K	No Class 100K/4K
NM	9/4	8/4	7/4	4/2
LM	-/8	9/8	9/8	7/7

Table 3: Switch Program Overheads (# Stages)

fair share. In contrast, with Nimble, the different congestion control algorithms achieve a more stable throughput, and each TC achieves the appropriate rate-limit.

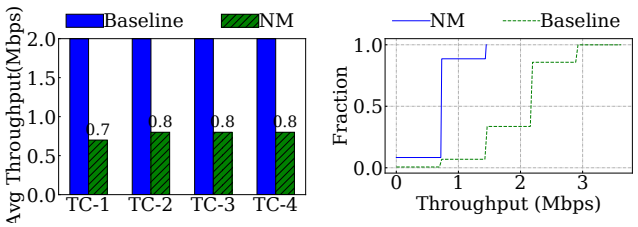
6.2 Scalability Analysis

To demonstrate Nimble is scalable, we analyze the overheads of supporting 100K and 4K rate-limiters per-switch, and we perform testbed experiments with 50K rate-limiters.

6.2.1 Switch Stage Overheads. To demonstrate scalability, we created programs for the Tofino switch that provide 100K and 4K per-switch rate-limiters for both native and logical meters. Because these programs compile, they are guaranteed to run at line-rate [12], so this demonstrates that Nimble is able to scale.

To understand the overheads of Nimble, we analyzed the number of stages required to implement NM and LM switch programs. Table 3 presents the resource allocation in terms of number of RMT stages for different variants of Nimble. Because supporting a larger number of rate-limiters may require additional stages, this table shows results for programs that provide both 100K and 4K rate-limiters. Similarly, because different approaches to network classification have different overheads, this table shows the overheads of using a different number of inputs to the lookup tables used for routing and classifying packets into TCs.

In Table 3, the overheads of meters range from 2–9 stages. The only program that did not compile was 100K rate-limiters with a



(a) Avg Throughput **(b) CDF of all TC**

Figure 8: Large number of traffic classes

7-tuple lookup. Also, the final column shows the overhead of just doing rate-limiting without any traffic class assignment per packet. Additionally, these results are for using LM with a single rate meter.

All the results in Table 3 use exact key matches for classification. We also explored the corresponding overheads of using ternary key matches. We were able to build a switch program that supports 40K and 20K rate-limiters when using a 4-tuple ternary match with NM and LM, respectively.

6.2.2 50K Active Traffic Classes. To demonstrate that Nimble can accurately enforce rate-limits even while supporting a large number of TCs, we performed an experiment where a single server generates packets from 50K different TCs and each TC is subject to a rate-limit of 1 Mbps. We use BESS [2, 26] to generate UDP packets using 10 cores each in a round robin fashion, and we use 9KB jumbo frames because the server is otherwise not able to generate packets at 100 Gbps. In this experiment, each TC should ideally receive 1 Mbps of throughput after rate-limiting. Without rate-limiting, each TCs fair share is 2 Mbps, although some TCs may send faster.

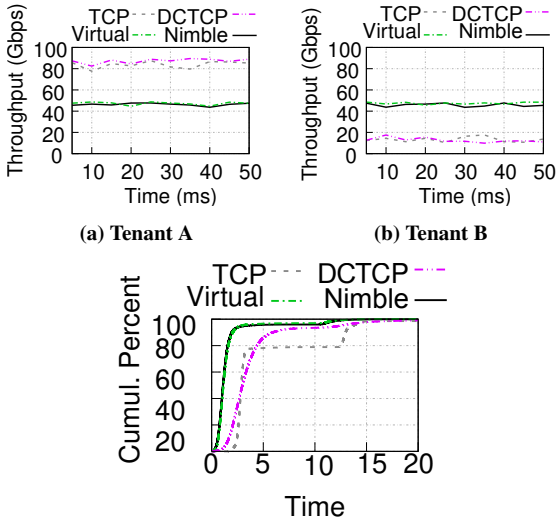
The results from this experiment are presented in Figure 8. Figure 8a plots the average throughput of 4 random TCs, and Figure 8b is a CDF of all of the throughputs of the 50K TCs. These figures show that meters can effectively enforce rate-limits even at a large number of flows and low rate-limits.

6.3 Simulation

We utilize simulation to evaluate performance at scale, to isolate the performance improvement of our centralized controller, to analyze the effect of updating rate limiters, and to understand the trade-offs in multiplication tables.

6.3.1 At-scale performance. To study performance at large scales, we simulate a 128-node 3-tier fat-tree with two TCs and closed-loop tenants generating randomized all-to-all traffic where tenant A generates eight times more flows than tenant B. We randomize senders and receivers every 1 ms and periodically introduce short flows from tenant B to measure network delay. We use a fair-share network policy between TCs. Our simulation settings (e.g., ECN thresholds) are as specified in Section 5. We compare four systems: TCP and DCTCP (without in-network rate limiters), Virtual Queues (VQ(ECN)), and Nimble with NM and ECN-shaping. In our implementation, we do not limit the number of virtual queues.

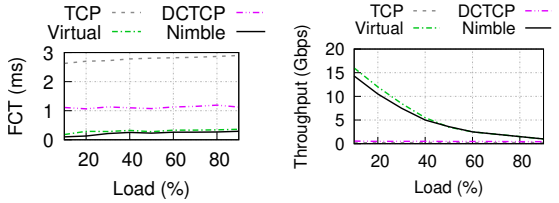
Figure 9 shows the throughput of tenants (normalized per server) and the CDF of network delay (from short flows). From Figures 9(a)(b), we see that TCP and DCTCP suffer from unfairness. Tenant A generates more flows and gets more throughput. Virtual



(a) Tenant A **(b) Tenant B**

(c) CDF of FCTs

Figure 9: Performance isolation at scale



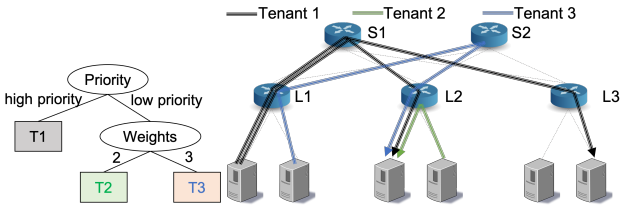
(a) 99^{th} percentile FCT **(b) Long flow throughput**

Figure 10: Realistic workload with varying loads at scale

Queues and Nimble achieve close to a fair share. Figure 9(c) shows the CDF of short flow completion times. Here, we see that Virtual Queues and Nimble achieve a much tighter distribution (i.e., lower queuing delays) by isolating tenant B from the aggressive tenant A.

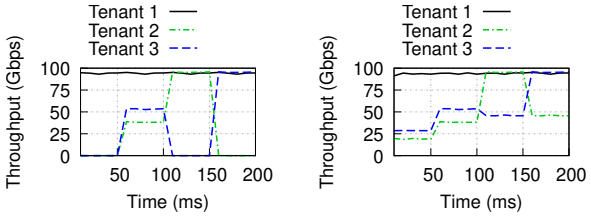
Next, we show an open-loop experiment with a fair-share policy between two tenants. Tenant A generates short (16 KB) and long flows (64 MB) with a skewed distribution. Figure 10 shows the tail latency and throughput for four systems: TCP, DCTCP, Virtual Queues, and Nimble. We clearly see that while TCP and DCTCP suffer from high latency and low throughput, Nimble performs as well as virtual queues without the associated hardware costs of virtual queues. Figure 13 shows the CDF of queue lengths of a port over time for the four systems when running preceding workload at 70% load. We clearly see that Nimble isolates the tenants and achieves much shorter queue lengths as compared to other systems including virtual queues without requiring *physically* distinct queues for each tenant.

6.3.2 Global vs. Local Policy Enforcement. In this section, we isolate the effect of using global knowledge in a centralized controller to compute rates as opposed to computing rates locally in switches based on the presence of paths. For this study, we simulate a simple leaf-spine topology consisting of three leaf switches that each connect to two servers on downlinks and two spine switches on uplinks; all the links operate at 100 Gbps. We simulate three



(a) Simulated policy (b) Simulated traffic (0–50 ms)

Figure 11: Simulated policy and traffic



(a) Switch-local rate calc. (b) Global (Algorithm 2)

Figure 12: Global vs. local policy enforcement

tenants ($Tenant$ 1–3). Network policy is set such that $Tenant$ 1 is high priority, whereas $Tenant$ 2 and $Tenant$ 3 are lower in priority than $Tenant$ 1, but their weights are 2 and 3, respectively (see Figure 11a). The tenants generate a new set of random paths every 50 ms. $Tenant$ 1 has *two* paths and $Tenant$ 2 and $Tenant$ 3 have *one* path each.

Figure 12 shows the throughput of three tenants with (1) local rate computation that honors network policy but it is agnostic to paths being bottlenecked in other switches (Figure 12a), and (2) global rate computation based on our algorithm (Figure 12b). The plot shows four 50 ms intervals that capture a diverse set of traffic contentions.

In the first 50 ms (shown in Figure 11b), one path from $Tenant$ 1 competes with $Tenant$ 2 and $Tenant$ 3 at the switch $L2$; the other path from $Tenant$ 1 has no contention. The two paths from $Tenant$ 1 are bottlenecked at the source , and cannot individually fully utilize the link capacity. Since the local assignment at switch $L2$ and switch $L3$ is agnostic to other bottlenecked paths, it assigns 100% of line rate for each of the two paths of the high priority tenant $Tenant$ 1 and throttles $Tenant$ 2 and $Tenant$ 3 to 0%. However, because the two $Tenant$ 1 paths contend at the source, they can only use 100% together (50% each). Thus, the local assignment wastes bandwidth as Tenants 2 and 3 are fully throttled even though there is spare bandwidth of 50% at switch $L2$. In contrast, our global allocation finds out that the two $Tenant$ 1 paths are each bottlenecked to 50% of line rate and assigns the remaining 50% of line rate to $Tenant$ 2 and $Tenant$ 3 in the ratio 2 ($\frac{2}{5} \times 50Gbps$) and 3 ($\frac{3}{5} \times 50Gbps$) respectively.

In the second interval, $Tenant$ 2 and $Tenant$ 3 compete at the same port but they do not share any port with $Tenant$ 1. In this case, the local and global allocations are identical the tenants share the capacity proportional to their weights. In third and fourth intervals, one path of $Tenant$1 competes with $Tenant$ 3 and $Tenant$ 2, respectively. As before, $Tenant$ 1 is limited at the source to 50% of line rate. The local allocation being agnostic of global bottlenecks assigns the full line rate to $Tenant$ 1 and throttles the other tenant,

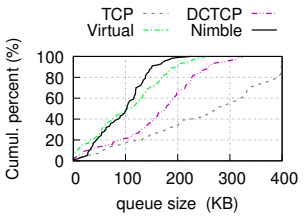


Figure 13: CDF of Queue lengths

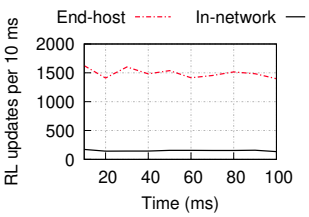


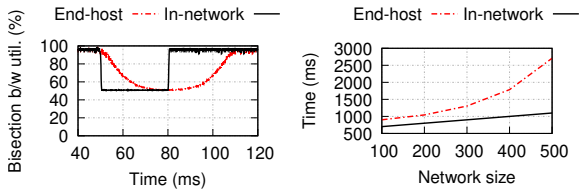
Figure 14: Updates per 10 ms

which is clearly sub-optimal. On average, we found that global rate allocation achieves *24% higher network utilization* than local allocation.

6.3.3 Updating Rate-Limiters. To study path churn, we use a leaf-spine topology with 10 leaf switches, 5 spine switches, and 10 servers per rack (100 servers). On this topology, the worst case total number of rate limiters that could need to be updated in edge-based scheme and Nimble are 9900 and 400, respectively, which is a 24x reduction. Additionally, to study the average case reduction in updates, we generate random traffic with the same open-loop workload as before at 70% load and count the number of rate-limiters updated in every 10 ms interval. Figure 14 shows the number of rate-limiters updates for both the schemes. Nimble reduces the number of such updates by about a *factor of 10x*. By reducing the number of updates, Nimble reduces the bandwidth loss during transition.

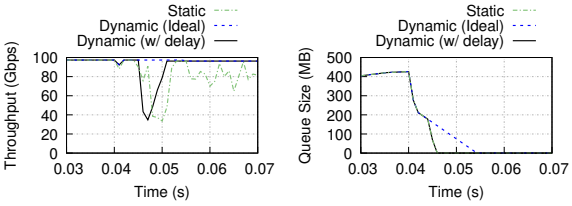
We now study the performance and scalability of edge-based and in-network rate limiters (Nimble). Figure 15(a) shows how long it takes for the throughput to converge as we reduce/increase rate limiter values for a leaf-spine topology with 256 servers using *random* all-to-all traffic. We model the delay to program rate limiters as a normal distribution with a mean of 1 ms and variance of 10 ms. At about 50 ms, we reduce the rate limiter values to 50% and at about 80 ms, we restore the values to 100%. We see a clear difference in the performance of the two designs. Because edge-based designs require more updates, which would likely take longer time, they converge slowly. Slower convergence ultimately would lead to under-utilization of capacity and/or congestion. Figure 15(b) shows the *worst case* convergence time to complete updating all the rate limiters as we vary the network size. For worse case, we generate a full mesh traffic as opposed to Figure 15(a), which uses a random all-to-all traffic. In an edge-based design, there is one rate limiter for every combination of send-receive pairs (per tenant/traffic class). So, in the worst case, all of them may need to be updated, which is $O(n^2)$ for n servers. In contrast, Nimble requires rate limiters only at the output ports of switches (per tenant/traffic class) and so the update complexity is $O(n)$. Our results are in line with this observation and we see that the time to finish updating the rate limiters grows at a much faster rate in an edge-based design whereas it remains fairly low for Nimble. Thus, Nimble achieves better scalability than edge-based designs.

We study the effect of dynamic control loop update latency on performance with a dumbbell topology and two tenants. Tenant A sends data between the top two servers, and Tenant B sends data between the bottom two servers. The rate-limiters are set to share the 100 Gbps bottleneck link equally between the two tenants. At the start, the servers on the left send data to the servers on the



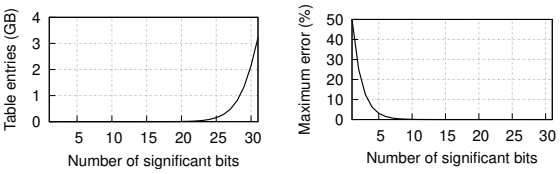
(a) Performance **(b) Scalability (Worst-case)**

Figure 15: Effect of update latency



(a) Throughput **(b) Queue size**

Figure 16: Effect of dynamic loop update



(a) Table size **(b) Error**

Figure 17: Error vs. state overhead

right. At 0.04 seconds, the second server belonging to tenant A stops. Ideally, the rate-limits should be updated to 33 *Gbps* for Tenant A and 67 *Gbps* for Tenant B. We compare three systems: (1) Rate-limiters are not updated (*Static*), (2) Rate-limiters are updated instantaneously (*Dynamic (Ideal)*), and (3) Rate-limiters are updated after 10 ms (*Dynamic w/ delay*).

Figure 16 shows the aggregate throughput and queue size of the port in the leftmost switch that connects to the other switch for the three systems. From Figure 16(a), we see that *Static* fails to achieve 100% throughput, whereas *Dynamic w/ delay* quickly catches up with *Dynamic (Ideal)* after 10 ms of delay. From Figure 16(b), we see that *Static* and *Dynamic w/ delay* underflow at around 0.04 s, which causes loss of throughput. After 0.05 s, all the three systems converge to their desired rates and queue size becomes close to zero.

6.3.4 Overhead of precision in multiplication. There is a trade-off between the number of table entries and the error in rate calculation for NM. We characterize the maximum error and state overhead as a function of the number of significant bits in Figure 17. We observe that the error drops significantly after 5 bits and the overhead remains minimal until 25 bits, giving a large operating range for Nimble.

To quantify the effect of limited precision, we simulated a dumbbell topology and set rate-limits to be 50% of line rate and compare three systems: perfect multiplication and two other systems that approximate multiply tables with number of significant bits (see

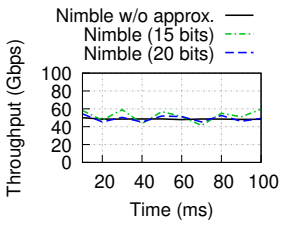


Figure 18: Effect of precision

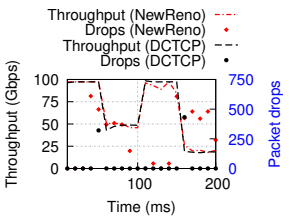


Figure 19: TCP friendliness

Algorithm 1) being 15 and 20, respectively. Figure 18 shows the throughput of flows over time for the three systems. We observe that the system with 15 bits varies in throughput more than the others due to errors in rate calculation. Fortunately, the error is small and all of these three systems perform with 5% of each other on average.

6.3.5 TCP-Friendliness. We want to ensure that (1) Nimble enforces the rate correctly, (2) the rate converges to the desired value and does not oscillate, and (3) the packet losses happen only during the rate transition. We conducted an experiment with a dumbbell topology and traffic flowing from left to right. We start the experiment without any rate limits. At 50 ms, we set the rate limit to be 50% of line rate; at 100 ms, we remove the rate limit; at 150 ms, we set a lower rate limit of 20%. Figure 19 shows the throughput and number of packet drops over time for two TCP variants: DCTCP and TCP New Reno. Both variants converge to desired rates and remain stable.

7 RELATED WORK

Wang *et al.* created a P4 implementation of rate-limiting that uses meters and ECN marking [65]. However, their implementation uses a timer to refill tokens, and this approach is not scalable to a large number of rate-limiters or fast rates. Moreover, because congestion is a common occurrence in many of the real world networks, there is high likelihood of the refill tokens being dropped before processing, which would cause sub-optimal performance. Further, the number of hardware timers and traffic classes are limited, so there are challenges to scaling their idea to a large number of traffic classes.

Brown *et al.* [13] discuss the problems with congestion control on the Internet, and Nimble provides new mechanisms that can support the types of isolation they argue for.

Nimble is complementary to improvements to programmable virtual queuing like PIFO [58] and PIEO [54]. Nimble can rate-limit leaf TCs while policies that aggregate TCs can be implemented with scheduling queues.

Nimble is also complementary to advances in rate-limiting at end hosts like Carousel [49] and Eiffel [50]. Similarly, Loom [61] and SENIC [47] are new NIC designs that offload the rate-limiting to the hardware. However, these systems do not eliminate the need for in-network rate-limiting.

AC/DCTCP [28] and VirtTCP [17] are both complementary to Nimble. In virtualized cloud environments, these systems can be used to ensure that flows from non-compliant tenants and tenants that do not implement DCTCP can still be forced to use a policy-compliant congestion window.

Similarly, TIMELY [44] and Swift [39] are a new delay-based congestion control algorithms that are also complementary to Nimble. With shaping, queue build up can lead to unacceptable increases in latency, and TIMELY and Swift can detect this buildup and react at end hosts.

Nimble is related to systems that use rate-limiting as a primitive to compute policy-compliant rates. For example, BwE is used in Google's B4 to allocate bandwidth [31, 38]. However, the algorithm used for computing rate-limits is not applicable to Nimble because their algorithm is designed only for edge-based rate-limiters and does not consider the placement of in-network rate-limiters. Accuracy could be improved and rate limiter update frequency could be reduced if Nimble was used to enforce rate-limits inside the network.

Nimble is also complementary to Silo [35]. Nimble can be used to improve upon its end-host based rate-limiting scheme, and Nimble's algorithm could be adapted to provide bandwidth reservations as in Silo. Additionally, EyeQ [36] and Hull [9] are two additional systems that compute rate-limits that can be enforced inside the network by Nimble.

Chen *et al.* propose a fine-grained way to measure queue occupancy [15] but it's not straightforward to infer the rate of a particular flow from the queue length(or queueing delay). Inferring rate is a key component in providing performance isolation. This further shows the need for a system like Nimble.

Finally, there is existing work on creating building blocks in programmable switches for some congestion control and load balancing protocols [51]. Nimble uses logical meters as primitives which is more generally applicable. We leverage logical meters and ECN shaping for global network policy enforcement and present an algorithm for computing policy-compliant rates. This [51] work also introduces approximate multiplication, a fair comparison of TCAM and SRAM resources to determine the efficacy between the two approaches is not possible as they use a Cavium Xpliant CNX880xx [14] switch.

8 CONCLUSIONS

A significant problem facing today's networks is that there is no scalable, accurate, and TCP-friendly approach for in-network rate-limiting to enforce network policies. To address this deficiency, this paper introduces the design of Nimble, a new system that provides new configurable mechanisms for accurate, scalable, and TCP-friendly in-network rate-limiting. As networks continue to grow in scale and speed and as multi-tenancy becomes more common, flexible in-network rate-limiters such as Nimble will be necessary to meet the stringent service demands of applications.

The contributions of Nimble are introducing logical meters (LM), and a new algorithm for enforcing complex network-wide isolation policies. To evaluate Nimble, we perform experiments with a 100Gbps Barefoot Tofino switch and the ns-3 simulator. We demonstrate that native meters (NM) and logical meters (LM) are implementable on commodity switches and scale to 100K independent rate-limiters. ECN-shaping overcomes the limitations of shaping and policing and enables end-hosts to converge to policy-compliant rates. When compared with edge-based rate-limiting, Nimble results in 10x–24x fewer rate-limiter updates. When compared with local per-switch enforcement, Nimble results in more accurate policy enforcement and higher network utilization (24%).

Acknowledgments: We would like to thank our shepherd Eric Rozner and the SOSR reviewers for their comments that helped in improving our paper. This work was supported by NSF award CNS-2008273.

REFERENCES

[1] Apache—HTTP Server Project. https://httpd.apache.org/.
[2] BESS: Berkeley Extensible Software Switch. https://github.com/NetSys/bess.
[3] iperf3: Documentation. http://software.es.net/iperf/.
[4] P4 Language Consortium. https://p4.org/.
[5] Redis. https://redis.io/.
[6] sockperf: Network benchmarking utility. https://github.com/Mellanox/sockperf.
[7] The ns-3 discrete-event network simulator. http://www.nsnam.org.
[8] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data Center TCP (DCTCP). In *SIGCOMM* (2010).
[9] ALIZADEH, M., KABBANI, A., EDSALL, T., PRABHAKAR, B., VAHDAT, A., AND YASUDA, M. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *NSDI* (2012).
[10] BALLANI, H., COSTA, P., KARAGIANNIS, T., AND ROWSTRON, A. Towards predictable datacenter networks. In *SIGCOMM* (2011).
[11] BAREFOOT. Barefoot Tofino. https://www.barefootnetworks.com/technology/#tofino, 2017.
[12] BOSSHART, P., GIBB, G., KIM, H.-S., VARGHESE, G., MCKEOWN, N., IZZARD, M., MUJICA, F. A., AND HOROWITZ, M. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. In *SIGCOMM* (2013).
[13] BROWN, L., ANANTHANARAYANAN, G., KATZ-BASSETT, E., KRISHNAMURTHY, A., RATNASAMY, S., SCHAPIRA, M., AND SHENKER, S. On the future of congestion control for the public internet. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)* (2020), Association for Computing Machinery.
[14] CAVIUM CORPORATION. Cavium CNX880XX. http://www.cavium.com/pdfFiles/CNX880XX_PB_Rev1.pdf.
[15] CHEN, X., FEIBISH, S. L., KORAL, Y., REXFORD, J., ROTTENSTREICH, O., MONETTI, S. A., AND WANG, T.-Y. Fine-grained queue measurement in the data plane. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies* (2019), p. 1529.
[16] CHEN, Y.-C., AND XU, X. An adaptive buffer allocation mechanism for token bucket flow control. vol. 4, pp. 3020 – 3024 Vol. 4.
[17] CRONKITE-RATCLIFF, B., BERGMAN, A., VARGAFTIK, S., RAVI, M., MCKEOWN, N., ABRAHAM, I., AND KESLASSY, I. Virtualized congestion control. In *SIGCOMM* (2016).
[18] DALTON, M., SCHULTZ, D., ADRIAENS, J., AREFIN, A., GUPTA, A., FAHS, B., RUBINSTEIN, D., ZERMENO, E. C., RUBOW, E., DOCAUER, J. A., ALPERT, J., AI, J., OLSON, J., DECABOOTER, K., DE KRUIJF, M., HUA, N., LEWIS, N., KASINADHUNI, N., CREPALDI, R., KRISHNAN, S., VENKATA, S., RICHTER, Y., NAIK, U., AND VAHDAT, A. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *NSDI* (2018).
[19] DATA CENTER BRIDGING TASK GROUP. http://www.ieee802.org/1/pages/dcbridges.html.
[20] DUKKIPATI, N. *Rate Control Protocol (Rcp): Congestion Control to Make Flows Complete Quickly*. PhD thesis, Stanford, CA, USA, 2008. AAI3292347.
[21] FIRESTONE, D., PUTNAM, A., MUNDKUR, S., CHIOU, D., DABAGH, A., ANDREWARTHA, M., ANGEPAT, H., BHANU, V., CAULFIELD, A., CHUNG, E., CHANDRAPPA, H. K., CHATURMOHTA, S., HUMPHREY, M., LAVIER, J., LAM, N., LIU, F., OVTCHAROV, K., PADHYE, J., POPURI, G., RAINDEL, S., SAPRE, T., SHAW, M., SILVA, G., SIVAKUMAR, M., SRIVASTAVA, N., VERMA, A., ZUHAIR, Q., BANSAL, D., BURGER, D., VAID, K., MALTZ, D. A., AND GREENBERG, A. Azure accelerated networking: SmartNICs in the public cloud. In *NSDI* (2018), USENIX Association.
[22] FLACH, T., PAPAGEORGE, P., TERZIS, A., PEDROSA, L., CHENG, Y., KARIM, T., KATZ-BASSETT, E., AND GOVINDAN, R. An internet-wide analysis of traffic policing. In *SIGCOMM* (2016).
[23] FLOYD, S., AND JACOBSON, V. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw.* (1993).
[24] GEMBER-JACOBSON, A., VISWANATHAN, R., PRAKASH, C., GRANDL, R., KHALID, J., DAS, S., AND AKELLA, A. OpenNF: Enabling innovation in network function control. In *SIGCOMM* (2014).
[25] GOYAL, P., SHAH, P., SHARMA, N. K., ALIZADEH, M., AND ANDERSON, T. E. Backpressure flow control, 2019.
[26] HAN, S., JANG, K., PANDA, A., PALKAR, S., HAN, D., AND RATNASAMY, S. Softnic: A software nic to augment hardware. Tech. Rep. UCB/EECS-2015-155,

EECS Department, University of California, Berkeley, May 2015.

[27] He, K., Qin, W., Zhang, Q., Wu, W., Yang, J., Pan, T., Hu, C., Zhang, J., Stephens, B., Akella, A., and Zhang, Y. Low latency software rate limiters for cloud networks. In *APNet* (2017), ACM.

[28] He, K., Rozner, E., Agarwal, K., Gu, Y. J., Felter, W., Carter, J., and Akella, A. AC/DC TCP: Virtual congestion control enforcement for datacenter networks. In *SIGCOMM* (2016).

[29] Heinanen, J., Finland, T., and Guerin, R. Rfc2697: A single rate three color marker, 1999.

[30] Heinanen, J., and Guerin, R. Rfc2698: A two rate three color marker, 1999.

[31] Hong, C.-Y., Kandula, S., Mahajan, R., Zhang, M., Gill, V., Nanduri, M., and Wattenhofer, R. Achieving high utilization with software-driven WAN.

[32] Hong, C.-Y., Mandal, S., Al-Fares, M., Zhu, M., Alimi, R., B., K. N., Bhagat, C., Jain, S., Kaimal, J., Liang, S., Mendelev, K., Padgett, S., Rabe, F., Ray, S., Tewari, M., Tierney, M., Zahn, M., Zolla, J., Ong, J., and Vahdat, A. B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined WAN. In *SIGCOMM* (2018).

[33] Iorgulescu, C., Azimi, R., Kwon, Y., Elnikety, S., Syamala, M., Narasayya, V., Herodotou, H., Tomita, P., Chen, A., Zhang, J., and Wang, J. Perfiso: Performance isolation for commercial latency-sensitive services. In *USENIX ATC* (2018), USENIX Association.

[34] Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., Venkata, S., Wanderer, J., Zhou, J., Zhu, M., Zolla, J., Hölzle, U., Stuart, S., and Vahdat, A. B4: Experience with a globally-deployed software defined wan. In *SIGCOMM* (2013).

[35] Jang, K., Sherry, J., Ballani, H., and Moncaster, T. Silo: Predictable message latency in the cloud. In *SIGCOMM* (2015).

[36] Jeyakumar, V., Alizadeh, M., Mazieres, D., Prabhakar, B., Kim, C., and Greenberg, A. EyeQ: Practical network performance isolation at the edge. In *NSDI* (2013).

[37] Jose, L., Yan, L., Alizadeh, M., Varghese, G., McKeown, N., and Katti, S. High speed networks need proactive congestion control. In *HotNets* (2015), ACM.

[38] Kumar, A., Jain, S., Naik, U., Kasinadhuni, N., Zermeno, E. C., Gunn, C. S., Ai, J., Carlin, B., Amarandei-Stavila, M., Robin, M., Siganporia, A., Stuart, S., and Vahdat, A. BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing. In *SIGCOMM* (2015).

[39] Kumar, G., Dukkipati, N., Jang, K., Wassel, H. M. G., Wu, X., Montazeri, B., Wang, Y., Springborn, K., Alfeld, C., Ryan, M., Wetherall, D., and Vahdat, A. Swift: Delay is simple and effective for congestion control in the datacenter. In *SIGCOMM* (2020), Association for Computing Machinery.

[40] Liu, Z., Manousis, A., Vorsanger, G., Sekar, V., and Braverman, V. One sketch to rule them all: Rethinking network flow monitoring with UnivMon. In *SIGCOMM* (2016).

[41] Lo, D., Cheng, L., Govindaraju, R., Ranganathan, P., and Kozyrakis, C. Heracles: Improving resource efficiency at scale. In *ISCA* (2015).

[42] Marty, M., de Kruijf, M., Adriaens, J., Alfeld, C., Bauer, S., Contavalli, C., Dalton, M., Dukkipati, N., Evans, W. C., Gribble, S., et al. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019).

[43] Mellanox Technologies. ConnectX-5 EN Card. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-5_EN_Card.pdf.

[44] Mittal, R., Lam, T., Dukkipati, N., Blem, E., Wassel, H., Ghobadi, M., Vahdat, A., Wang, Y., Wetherall, D., and Zats, D. TIMELY: RTT-based congestion control for the datacenter. In *SIGCOMM* (2015).

[45] Narayana, S., Sivaraman, A., Nathan, V., Goyal, P., Arun, V., Alizadeh, M., Jeyakumar, V., and Kim, C. Language-directed hardware design for network performance monitoring. In *SIGCOMM* (2017).

[46] Popa, L., Kumar, G., Chowdhury, M., Krishnamurthy, A., Ratnasamy, S., and Stoica, I. FairCloud: Sharing the network in cloud computing. In *SIGCOMM* (2012).

[47] Radhakrishnan, S., Geng, Y., Jeyakumar, V., Kabbani, A., Porter, G., and Vahdat, A. SENIC: Scalable NIC for end-host rate limiting. In *NSDI* (2014).

[48] Roy, A., Zeng, H., Bagga, J., Porter, G., and Snoeren, A. C. Inside the social network's (datacenter) network. In *SIGCOMM* (2015).

[49] Saeed, A., Dukkipati, N., Valancius, V., Lam, T., Contavalli, C., and Vahdat, A. Carousel: Scalable traffic shaping at end-hosts. In *SIGCOMM* (2017).

[50] Saeed, A., Zhao, Y., Dukkipati, N., Zegura, E., Ammar, M., Harras, K., and Vahdat, A. Eiffel: Efficient and flexible software packet scheduling. In *NSDI* (2019), USENIX Association.

[51] Sharma, N. K., Kaufmann, A., Anderson, T., Krishnamurthy, A., Nelson, J., and Peter, S. Evaluating the power of flexible packet processing for network resource allocation. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)* (2017), pp. 67–82.

[52] Sharma, N. K., Liu, M., Atreya, K., and Krishnamurthy, A. Approximating fair queueing on reconfigurable switches. In *NSDI* (2018), USENIX Association.

[53] Shieh, A., Kandula, S., Greenberg, A., and Kim, C. Sharing the data center network. In *NSDI* (2011).

[54] Shrivastav, V. Fast, scalable, and programmable packet scheduler in hardware. In *SIGCOMM* (2019), ACM.

[55] Singh, A., Ong, J., Agarwal, A., Anderson, G., Armistead, A., Bannon, R., Boving, S., Desai, G., Felderman, B., Germano, P., Kanagala, A., Provost, J., Simmons, J., Tanda, E., Wanderer, J., Hölzle, U., Stuart, S., and Vahdat, A. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *SIGCOMM* (2015), pp. 183–197.

[56] Sivaraman, A., Cheung, A., Budiu, M., Kim, C., Alizadeh, M., Balakrishnan, H., Varghese, G., McKeown, N., and Licking, S. Packet transactions: High-level programming for line-rate switches. In *SIGCOMM* (2016).

[57] Sivaraman, A., Kim, C., Krishnamoorthy, R., Dixit, A., and Budiu, M. Dc.p4: Programming the forwarding plane of a data-center switch. In *SOSR* (2015), ACM.

[58] Sivaraman, A., Subramanian, S., Alizadeh, M., Chole, S., Chuang, S.-T., Agrawal, A., Balakrishnan, H., Edsall, T., Katti, S., and McKeown, N. Programmable packet scheduling at line rate. In *SIGCOMM* (2016).

[59] Sonchack, J., Michel, O., Aviv, A. J., Keller, E., and Smith, J. M. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow. In *USENIX ATC* (2018).

[60] Soni, H., Rifai, M., Kumar, P., Doenges, R., and Foster, N. Composing dataplane programs with μp4. SIGCOMM.

[61] Stephens, B., Akella, A., and Swift, M. Loom: Flexible and efficient NIC packet scheduling. In *NSDI* (2017).

[62] Stephens, B., Cox, A. L., Singla, A., Carter, J., Dixon, C., and Felter, W. Practical DCB for improved data center networks. In *INFOCOM* (2014).

[63] Sundarrajan, A., Feng, M., Kasbekar, M., and Sitaraman, R. K. Footprint Descriptors: Theory and practice of cache provisioning in a global CDN. In *CoNEXT* (2017), ACM.

[64] van Haalen, R., and Malhotra, R. Improving tcp performance with bufferless token bucket policing: A tcp friendly policer. In *2007 15th IEEE Workshop on Local Metropolitan Area Networks* (2007), pp. 72–77.

[65] Wang, S. Y., Hu, H. W., and Lin, Y. B. Design and implementation of tcp-friendly meters in p4 switches. *IEEE/ACM Transactions on Networking 28*, 4 (2020), 1885–1898.

[66] Yu, Z., Wu, J., Braverman, V., Stoica, I., and Jin, X. Twenty years after: Hierarchical core-stateless fair queueing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2021), USENIX Association.

[67] Zats, D., Das, T., Mohan, P., Borthakur, D., and Katz, R. DeTail: Reducing the flow completion time tail in datacenter networks. In *SIGCOMM* (2012).

[68] Zhang, Q., Liu, V., Zeng, H., and Krishnamurthy, A. High-resolution measurement of data center microbursts. In *Proceedings of the Internet Measurement Conference* (2017), IMC, ACM.

[69] Zhu, Y., Eran, H., Firestone, D., Guo, C., Lipshteyn, M., Liron, Y., Padhye, J., Raindel, S., Yahia, M. H., and Zhang, M. Congestion control for large-scale RDMA deployments. In *SIGCOMM* (August 2015), ACM.

[70] Zhu, Y., Ghobadi, M., Misra, V., and Padhye, J. ECN or delay: Lessons learnt from analysis of DCQCN and TIMELY. In *CoNEXT* (2016), ACM.