CoFilter: High-Performance Switch-Accelerated Stateful Packet Filter for Bare-Metal Servers

Jiamin Cao, Ying Liu, Yu Zhou, Lin He, Chen Sun, Yangyang Wang, and Mingwei Xu, *Member, IEEE*

Abstract—As one of the most critical cloud services, Bare-Metal Servers (BMS) introduce stringent performance requirements on data center networks (DCN). Stateful packet filter is an integral DCN component of ensuring connection security for BMS. However, the off-the-shelf stateful packet filters either are costly for cloud DCNs or introduce significant performance bottlenecks. In this article, we present *CoFilter*, which leverages low-cost programmable switches to accelerate the stateful packet filter for BMS. *CoFilter* uses (1) *stateful process partition* to enable complex stateful packet filtering logic on programmability-limited switching ASICs, (2) *state compression* to track tens of millions of connections with constrained hardware memory, and (3) *per-tenant packet rate limit and tenant-aware flow migration* to achieve efficient performance isolation among different tenants. Overall, *CoFilter* implements a high-performance stateful packet filter via the co-design of programmable switching ASIC and CPU. We evaluate *CoFilter* under various data center traffic traces with real-world flow distributions. The evaluation results show that *CoFilter* remarkably outperforms NetFilter, i.e., forwarding packets at line rate (13x throughput of NetFilter), keeping packet delay within 1us, and freeing a significant quantity of CPU cores, with rather small memory usage, i.e., accommodating over 10^7 connections with only 16MB SRAM.

Index Terms—Bare-metal server stateful packet filter, programmable switch

1 Introduction

The Bare-Metal Server (BMS) is becoming an increasingly important service for data center networks (DCN) [2]. Via offering customers sole access to the entire physical server, instead of running a hypervisor or being virtualized, BMS provides an excellent single-tenant environment, raw processing power, and high reliability [3]. For high-performance computing and data-intensive applications, BMS cloud is far better than virtualized cloud services to achieve high efficiency and high performance.

BMSs are appealing but also introduce distinctive requirements to DCNs. Due to the lack of a hypervisor layer, many essential DCN components have to be implemented outside of BMSs. In particular, *stateful packet filter* is a critical DCN element for BMS security. All incoming and outgoing traffic should go through the stateful packet filter, which provides per-packet inspection and security assurance by tracking connection states and dropping illegal packets. To provide a reliable security guarantee and meanwhile satisfy the stringent performance requirements of cloud applications, building a *high-performance* and *high-scalability* stateful packet filter is critical to BMS cloud providers.

The off-the-shelf stateful packet filter solutions for BMSs can be categorized into two types. First, *dedicated hardware-based solutions* [4], [5] deliver high performance but incur high costs. Depending on the bandwidth to cope with, a hardware packet filter can be quite expensive. For example, a Cisco stateful inspection firewall with 35Gbps stateful packet inspection throughput and 10 million concurrent connections can cost about 90K dollars [6]. Second, *software-based solutions* like NetFilter/ConnTrack [7] are common in practice, but they have significant performance issues. On the one hand, software-based solutions need to occupy more server resources to achieve higher throughput. On the other hand, software-based solutions encounter high packet forwarding delay and jitter, as CPU could introduce a delay of 50 microseconds to 1 ms [8]. In summary, the above solutions either are *prohibitively expensive* or result in a *significant performance compromise*, making them impractical for real-world BMS cloud.

In this paper, we propose *CoFilter*, which aims to implement a high-performance and low-cost stateful packet filter with P4-programmable switches [9]. Compared with expensive dedicated hardware and low-performance software, programmable switches deliver both high performance, i.e., high throughput (Tbps) and low latency (hundreds of nanoseconds), and low cost which is comparable to fixed

- *Jiamin Cao, Ying Liu, Lin He, and Yangyang Wang are with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China, and also with the Beijing National Research Center for Information Science and Technology, Beijing 100084, China. E-mail: cjm21@mails.tsinghua.edu.cn, {liuying, wangyy}@cernet.edu.cn, he-lin@mail.tsinghua.edu.cn.*
- *Yu Zhou and Chen Sun are with Alibaba Group, Alibaba Infrastructure Service, Hangzhou 311121, China. E-mail: {yuzhou.yz, qichen.sc}@alibaba-inc.com.*
- *Mingwei Xu is with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China, and with the Beijing National Research Center for Information Science and Technology (BNRist), Beijing 100084, China, and also with the Department of Computer Science, Tsinghua University, Beijing 100084, China. E-mail: xumw@tsinghua.edu.cn.*

Manuscript received 22 Jan. 2021; revised 14 Nov. 2021; accepted 7 Dec. 2021. Date of publication 20 Dec. 2021; date of current version 14 Feb. 2022. This work was supported in part by the National Natural Science Foundation of China under Grant 61772307, in part by the National Key Research and Development Program of China under Grant 2018YFB1800405, and in part by the research fund of Tsinghua University-Tencent Joint Laboratory for Internet Innovation Technology. (Corresponding author: Ying Liu.) Recommended for acceptance by J. Zhan. Digital Object Identifier no. 10.1109/TPDS.2021.3136575

function switching ASICs with the same forwarding rate. *The key idea of CoFilteris the logic and storage partition between the high-performance programmable ASIC and flexible CPU*. The CPU employed in *CoFilter* can be either the native switch CPU for cost savings or the external server CPU for larger memory and more processing power, depending on the actual demand. We conclude our contributions as follows.

- We propose *CoFilter*, a high-performance, scalable, and low-cost stateful packet filter based on new-generation P4-programmable switches for BMS cloud.
- *CoFilter* (1) overcomes the limited programmability of switching ASICs via *stateful process partition* between the programmable ASIC and CPU with negligible performance loss, (2) overcomes the limited memory of programmable ASICs via *state compression* in the ASIC with a small number of exact match entries, and (3) provides excellent performance isolation and enhances the system's resistance to attack via *per-tenant packet rate limit and tenant-aware flow migration*.
- We explore the practicality and deployability of *CoFilter*. *CoFilter* can be implemented either with one programmable switch or via accessing an external server for better performance. *CoFilter* can be deployed as the Top-of-Rack (ToR) switch to protect servers under it, or a packet filter cluster to protect all servers in a DCN.
- We build both the single-switch and hybrid prototypes of *CoFilter* and deploy them on a Tofino switch and a server. We evaluate *CoFilter*'s performance and scalability and compare *CoFilter* with *NetFilter* under various data center traces with real-world flow distributions. Evaluation results show that *CoFilter* can forward *data packets* at *line rate* (*13x higher* than *NetFilter*) and keep the upper bound of packet delay at a level of *one microsecond*. Moreover, *CoFilter* accommodates $> 10^7$ *connections* with only *16MB SRAM* and consumes much fewer CPU cores compared with *NetFilter*. When under attack, the performance isolation can further lower the attacked tenant's TCP initial round trip time (RTT) and flow completion time (FCT) by > 2 and > 5 magnitudes, respectively.

2 Background

The stateful packet filter is a critical security element for cloud data centers. Via inspecting each packet and tracking each connection, the stateful packet filter can provide connection-level security. A stateful packet filter generally comprises two parts. The first one is connection tracking. The stateful packet filter maintains a Finite State Machine (FSM) for every connection that passes through it. The state and other information about each connection are stored in a data structure named connection table. A new entry is (1) inserted to the connection table when a new connection comes, (2) updated as the connection goes, and (3) removed when the connection finishes. The second one is packet filtering. According to predefined filtering rules, the decision to permit or reject a packet is made based upon the current connection state maintained in the connection table, on top of the packet header information.

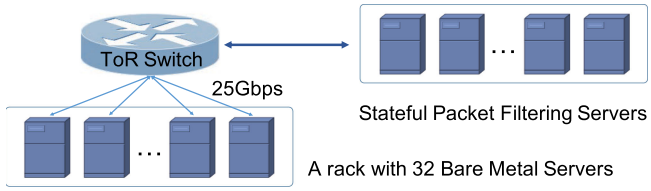


Fig. 1. Software-based packet filter for BMS cloud.

In BMS cloud which aims to provide high-performance services and therefore puts stringent requirements on networks. The stateful packet filter can be implemented in either hardware or software. The hardware packet filter relies on dedicated hardware to achieve high performance. But hardware also adds the deployment costs [6] and is fixed in functionality depending on the vendor.

Sometimes data centers may prefer software-based packet filter, e.g., NetFilter/iptables [7], which is widely used in Linux. In general, the software stateful packet filter has the following two-fold drawbacks. (1) Occupying a significant number of servers. According to our experiment in Section 7.3, a server with an Intel Xeon 6-core CPU can achieve $<$5Mpps throughput for 64-byte packets with ConnTrack enabled. As shown in Fig. 1, assume that 32 BMSs are deployed on one rack, and each server connects to the ToR switch with a 25Gbps link. All incoming and outgoing traffic should first traverse a stateful packet filter cluster which is a group of server nodes. In this case, the pure software-based solution needs 0.37K additional packet filter servers per rack to process the 800Gbps traffic, which is prohibitively costly.

(2) High delay for per-packet tracking. A software-based system adds a high packet delay of 50 microseconds, which is comparable to the end-to-end round-trip time in cloud data centers [8]. Moreover, keeping states of massive flows requires plentiful calculation, iteration, and kernel memory access. Such per-packet costly operations put tremendous pressure on the server CPU and significantly degrade the overall performance.

3 Overview of *CoFilter*

3.1 Challenges and Key Ideas

We encounter three challenges in *CoFilter* design. $\mathcal{C}_1$: *Limited Data Plane Programmability*. Switching ASICs pose many programmability constraints. We introduce two of them which greatly affect *CoFilter* design. First, switching ASICs limit the concurrent memory access to a single memory location [10]. The programmable switch has a pipeline of stages, and each stage only has a small number of stateful Arithmetic Logic Units (ALU) attached to the stateful memory (i.e., register) to read and write consistent states. However, the stateful memory can be accessed only once in the particular stage as the packet goes through the pipeline. This implies that we should define all state transition rules and finish state read-and-write operations within one stage. Second, the stateful ALU only supports two branching operations [11], i.e., an if-else operation to update registers differently. However, for stateful packet filter, there can be multiple subsequent states at a specific state and altogether tens of state transitions among all states. Therefore, the limited programmability of ASIC makes it difficult for *CoFilter*

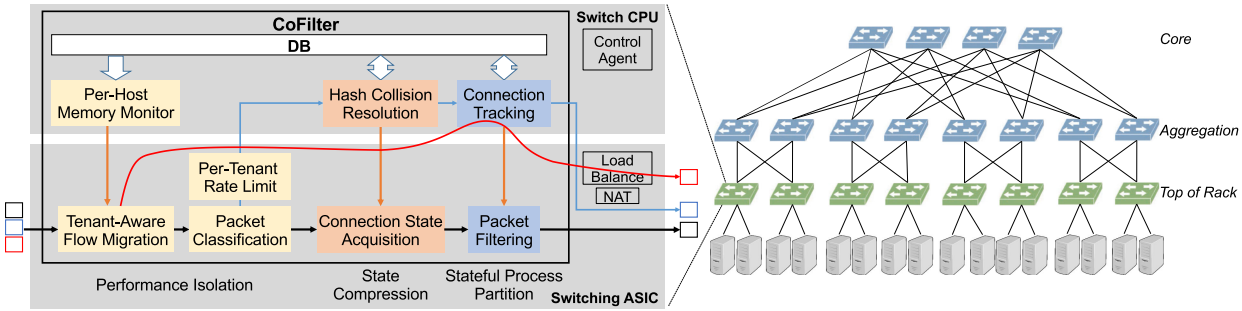


Fig. 2. Architecture and Deployment of *CoFilter*.

to cover tens of state transition rules for stateful packet filtering.

To handle C_1, we partition the complex stateful processing logic between the programmable ASIC and switch CPU (Section 4.1). We observe that although the state transition is quite complex, only a small number of packets trigger state transitions. Based on this, we classify packets into two types: a small number of *control packets* that trigger state transitions and should be tracked, e.g., SYN and FIN packets, and the majority *data packets* that just need to be filtered based on current connection states. Thus, we propose to slice the stateful packet filtering logic: control packets are processed by CPU, while data packets are directly processed on the ASIC. In this way, *CoFilter* can implement the complex state transition based on minority control packets and provide high-performance process for most data packets.

C_2: Limited Data Plane Resource. Tracking states of massive connections requires much memory. However, programmable ASICs have very limited SRAM for state storage. To store connection states in registers, an intuitive method would be using exact match-action tables to map each flow to a register instance directly. Each table entry includes a flow key, e.g., the 5-tuple (104 bits for IPv4 connections and 296 bits for IPv6 connections) as the match key and some additional action data bits referring to the register index. Ten million such entries take hundreds of MB SRAM, which is far more than 50-100MB SRAM available in programmable ASICs [8]. Therefore, the limited SRAM in ASIC renders it difficult to achieve per-flow tracking.

To handle C_2, we use *state compression* to fit tens of millions of connections with limited SRAM in programmable dataplane (Section 4.2). Instead of using an exact match-action table entry for each connection, we apply a general hash operation to all connections and only *maintain exact match entries to rewrite hash values for collided connections*. As the probability of hash collision is quite low, the scarce SRAM in existing ASICs would be sufficient for hash collision resolution. However, the collision calculation requires considerable programmability, which is beyond the capability of programmable ASICs. Therefore, we employ the switch CPU to calculate hash collisions at the arrival of each new connection and insert an exact match-action table entry to ASIC for collided connections. Furthermore, we adapt our design to the actual memory layout in programmable ASICs to make full use of the per-stage SRAM.

C_3: Lack of Efficient Performance Isolation. Performance isolation is a critical Quality of Service (QoS) guarantee to network in multi-tenant environments. Multiple tenants share resources on the switch and thus the behaviour of one tenant is affected by other tenants. Specifically, if one tenant comes under attack or a bug causes it to occupy too many resources, the performance of other tenants can be adversely impacted. Based on the type of shared resources, we identify two performance isolation requirement. First, tenants share the PCIe bandwidth between the ASIC and switch CPU. Congested PCIe can cause packet loss and larger packet delay. Second, tenants share the limited SRAM on the ASIC. If one tenant has too many concurrent flows, the ASIC may not have spare memory for other tenants.

To handle C_3, we propose two targeted methods for the two isolation requirements identified above. First, for the *shared PCIe bandwidth*, we introduce *per-tenant packet rate limit* to avoid any tenant occupying too much PCIe bandwidth and thus impacting the other tenants' performance (Section 5.1). Second, for the *shared memory on the ASIC*, we migrate flows to CPU when there is not enough SRAM on the ASIC. However, flow migration incurs performance degradation and actually loses the original performance advantages of programmable ASICs. Instead of directly migrating new-coming flows to CPU when the ASIC is full, we propose a migration algorithm which considers the practical demands for providing differential services, and selectively migrates more flows from the tenant that has lower priority and occupies more ASIC memory (Section 5.2).

3.2 Architecture of *CoFilter*

Fig. 2 shows the architecture of *CoFilter*. We take the fat-tree topology as an example. For simplicity of presentation, we assume that the x86 CPU is the original switch CPU, and *CoFilter* is deployed only at the top-of-rack (ToR) switches. We will discuss more flexible implementation methods and deployment scenarios in Section 6.

CoFilter contains three phases. (1) The Performance Isolation phase (Section 5) can prevent any tenant from occupying excessive resources and impeding other tenants' performance via two isolation modules, including Per-Tenant Packet Rate Limit and Tenant-Aware Flow Migration. The next two phases are about the design of stateful packet filter logic and storage. (2) The State Compression phase (Section 4.2) compresses the storage space with only a small number of exact match-action table entries. (3) The Stateful Process Partition phase (Section 4.1) slices the stateful packet filtering logic between the ASIC and switch CPU. Specifically, the Packet Classification module distinguishes control packets and data packets. The CPU performs Connection Tracking according to uploaded control packets and updates states stored on the

ASIC, while the ASIC performs Packet Filtering for data packets.

Based on the above three phases, *CoFilter* contains three packet workflows in Fig. 2, which are represented by lines of different colors. (1) The *red line* represents packets of migrated flows. The Performance Isolation phase sends packets of migrated flows to CPU. The connection tracking and packet filtering for migrated flows are completed by the CPU. Then the non-migrated packets are classified by the Packet Classification module into control packets and data packets. These two types of packets also have different workflows. (2) The *blue line* represents *control* packets of non-migrated flows. Control packets are sent to the switch CPU. CPU first executes hash collision calculation for each new connection and modifies the collision resolution entries in ASICs, and then updates the connection state and synchronizes states stored on the ASIC. (3) The *black line* represents data packets, which are directly processed on the ASIC. The State Compression phase uses only a small number of exact match entries to map each connection to an individual register instance. Then the Packet Filtering module in the Stateful Process Partition phase filters illegal packets. The migrated flows are totally processed in CPU and undergo the slowest path, while the non-migrated flows have the majority data packets directly processed on the ASIC and only the minority control packets going to CPU.

4 *CoFilter* Design

In this section, we present the *CoFilter* design which implements a high-performance stateful packet filter based on the cooperation of the programmable ASIC and switch CPU. In Section 4.1, *CoFilter slices the stateful packet filtering logic between the ASIC and CPU* to implement stateful processing while guaranteeing performance. In Section 4.2, *CoFilter slices the state storage between the ASIC and CPU* to scale to tens of millions of connections with limited SRAM.

4.1 Stateful Process Partition

To handle the limited programmability of switching ASICs and fully express the state transition logic of stateful packet filter, we slice the stateful packet filtering logic between the programmable ASIC and switch CPU.

A key observation is that for the stateful packet filter, packets that trigger state transition take up only a small proportion of a connection, while most packets have no impact on the state. We call the former *control packets*, including three packets (SYN, SYN/ACK, and ACK) in the three-way handshake before a connection is established, four packets (FIN, FIN/ACK, and ACK*2) in the four-way handshake when a connection is about to close, and RST packets. We call the latter *data packets*, including ACK and PSH packets.

Based on this observation, the insight of *CoFilter* is to partition the stateful packet filtering logic between the ASIC and switch CPU, as shown in Fig. 3. The ASIC does not need to be concerned about how states should be updated. Instead, they identify control packets and send them to the switch CPU. The CPU tracks connection states and updates the states stored on the ASIC in real time. Then the ASIC can perform packet filtering for data packets based on the stored states. In this way, *CoFilter* can process most data

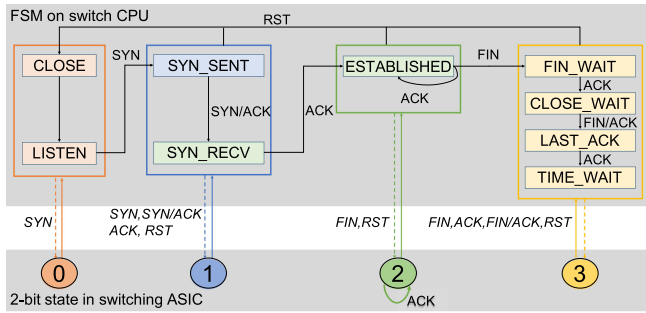


Fig. 3. Stateful process partition.

packets on the ASIC with high performance and also implement the complex state transition logic for those minority control packets with the help of the switch CPU.

Instead of storing the original TCP connection states, we store simplified 2-bit states on the ASIC to save memory. Each state on the ASIC may correspond to multiple realistic states on the CPU. The CPU is responsible for converting the original connection states to the corresponding compressed states in switch registers, as shown in Fig. 3. At first, the state is initialized as 0, representing a nonexistent connection. Only the SYN packet is a valid control packet and changes the state from 0 to 1. At state 1, SYN/ACK, ACK, RST, and retransmitted SYN packets are taken as control packets which are sent to CPU. Notice that SYN/ACK and retransmitted SYN packets just change the state on the CPU, i.e., between SYN_SENT and SYN_RECV, while the state on the ASIC remains at 1. In other words, both SYN_SENT and SYN_RECV on the CPU correspond to state 1 on the ASIC. The last ACK packet in the three-way handshake triggers state transition from SYN_RECV to ESTABLISHED on the CPU. When a connection is established, data transmission is allowed, and thus the state on the ASIC should change, i.e., from 1 to 2. During data transmission, ACK packets are always considered as data packets and processed on the ASIC at line rate until the FIN packet is received. On receiving the FIN packet, the state on the ASIC changes from 2 to 3. During the four-way handshake, control packets include FIN/ACK, ACK, RST, and retransmitted FIN packets. The four states on the CPU, i.e., FIN_WAIT, CLOSE_WAIT, LAST_ACK, and TIME_WAIT, are merged into the common state 3. Finally the CPU will reset the register state as 0 when the four-way handshake finishes.

4.2 State Compression

To handle the limited dataplane resource problem and fit tens of millions of connections with limited SRAM in programmable ASICs, we compress connection states via slicing the state storing between the ASIC and switch CPU. We propose two key ideas. First, we design a CPU-assisted three-phase collision settlement scheme which employs a small number of exact match-action table entries to store connections. Second, we adapt our design to the per-stage memory layout in programmable ASICs. Next, we will respectively illuminate these two ideas.

Collision Settlement Scheme. Before demonstrating our design, we first briefly introduce how to store states in registers and why hash is needed to track plentiful connections. In programmable ASICs, registers are organized into arrays

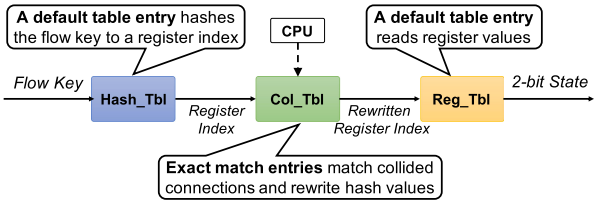


Fig. 4. Three-phase hash collision settlement scheme.

of instances. A naive method to store connection states is to use exact match-action tables to map flow keys, such as 5-tuples, to register indexes. As analyzed in Section 2, this method is impractical due to overmuch SRAM consumption. Hash is a commonly used technique to compress memory space. If we use hash, e.g., crc32, to map flow keys to register indexes, the SRAM for exact match-action table entries can be freed.

However, hash inevitably brings collisions, i.e., two different flow keys are mapped to the same register index. Hash collision resolution requires considerable memory and programmability, and cannot be implemented by the programmable ASIC alone. Therefore, we propose a CPU-assisted three-phase hash collision settlement scheme, as shown in Fig. 4. First, *Hash_Tbl* generates a hash value based on the flow key of each connection via a default table entry. Then *Col_Tbl* exactly matches with collided connections and rewrites hash values. Specifically, each table entry corresponds to a collided flow whose hash values should be rewritten. The *Col_Tbl* entries are added and deleted by the CPU, which maintains the mapping relationships between all flow keys and register indexes, and allocates a new available register instance for a collided connection. Finally, *Reg_Tbl* uses a default table entry to read connection states from the corresponding register instances and store the state into a 2-bit state metadata, which will be transmitted to the Packet Filtering module for further filtering.

Take a whole TCP connection as an example. (1) The first SYN packet of a connection is taken as a control packet and is uploaded to the CPU. The Hash Collision Resolution module on the CPU calculates hash values for this packet to decide whether this new connection collides with existing connections. If a collision happens, CPU will allocate a new available register instance and rewrite the hash value for this connection. Meanwhile, CPU will insert a new table entry to *Col_Tbl* in the ASIC, with the flow key as the exact match field, and updating the hash value to the new index as the action data. (2) The subsequent control packets also go to CPU but can skip the Hash Collision Resolution module. (3) The subsequent data packets remain on the ASIC and are matched by the above three tables. (4) When the last control packet arrives and the connection is to finish, CPU will delete the corresponding table entry from *Col_Tbl*.

Adaption to Multiple-Stage Programmable ASICs. The architecture of programmable switches [12] posses two restrictions on our design. First, the per-stage memory limitation demands that we should deliberate on the resource occupation of each table and place them in proper stages. Second, the concurrency of tables within one stage and dependency between cross-stage tables demand that sequential operations must be implemented in multiple stages in sequential order. Therefore, we design the state storing specifically so that it can fit into ASIC architecture and make full use of limited resources.

Fig. 5 shows how to adapt our three-phase collision settlement scheme to the multi-stage programmable ASIC. Notice that the basic collision settlement requires three stages, while the multi-stage optimization can further utilize more stages (6 stages in Fig. 5 as an example) to store more connections. We allocate register arrays in multiple stages and define an individual *Reg_Tbl* to access the register array from stage 3 to 6. An additional *Col_Tbl* pairs with each *Reg_Tbl* and is placed at the stage before *Reg_Tbl*, i.e., from stage 2 to 5. *Hash_Tbl* at stage 1 implements a default hash operation and is shared by all *Col_Tbl-Reg_Tbl* pairs. Combined with Packet Classification and Packet Filtering modules in Stateful Process Partition, *CoFilter* workflow is as follows: (1) When packets arrive at the ASIC, *Classify_Tbl* in the Packet Classification module singles out control packets, and sets the field *ctl* of metadata *md* as 1. These control packets skip the Connection State Acquisition module and directly go to *Upload_Tbl* in the Packet Filtering module, which will upload them to the CPU. Data packets with

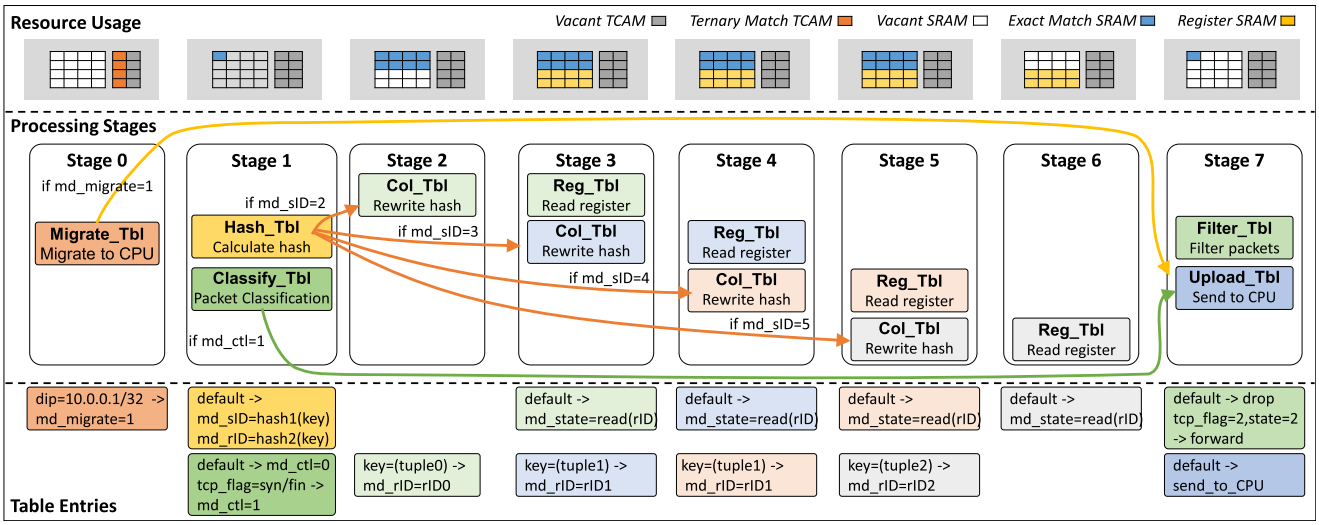


Fig. 5. Cross-stage optimization to store massive connections.

$md.ctl = 0$ will get into the Connection State Acquisition module. (2) In the Connection State Acquisition module, first *Hash_Tbl* maps the flow key (104-bit 5-tuple) to an s-bit metadata $md.sID$ and an r-bit metadata $md.rID$ respectively. $md.sID$ corresponds to 2^s *Col_Tbls* at the next 2^s stages. $md.rID$ corresponds to 2^r register instances in each register array and should be smaller than the upper register memory size within one stage. At the $md.sID^{th}$ stage, the data packet will hit *Col_Tbl* if this packet belongs to a collided connection, and *Col_Tbl* will rewrite $md.sID$ and $md.rID$. Then at the $md.sID + 1^{th}$ stage, *Reg_Tbl* reads the state from the $md.rID^{th}$ register instance and writes it into the 2-bit metadata $md.state$. (3) In the Packet Filtering module, *Filter_Tbl* exactly matches with $md.ctl$ and $md.state$ and drops packets according to filtering rules. *Upload_Tbl* also exactly matches with $md.ctl$ and $md.state$ to send control packets to CPU.

Once the register is full and no available instances can be allocated, we should migrate flows to CPU. To achieve this, we use a migration table, i.e., Migrate_Tbl, to identify flows that should be migrated. Then the packets that hit this table should be directly forwarded to CPU. However, flow migration incurs performance degradation and thus fairness problem among tenants, we will present how to decide the migrated flows in detail in Section 5.2.

5 Multi-Tenant Performance Isolation

Keeping good performance isolation among tenants is critical but challenging for cloud data center networks. To satisfy service-level agreement on the stateful packet filter, *CoFilter* should provide good performance isolation for connections of different tenants. In this paper, we identify two important performance isolation problems in the *CoFilter* system.

The first one refers to *control packet isolation*. The process partition uploads a small number of control packets via PCIe to achieve an efficient division between the ASIC and switch CPU, but also brings some hidden troubles in the following two aspects. First, the PCIe bandwidth between the ASIC and CPU is limited and shared by tenants. According to our experiment in Section 7.6, only 2.4 Gbps can cause PCIe congestion, with TCP initial RTT increasing from < 1000 microseconds to $> 10^6$ microseconds. Second, the performance of the shared switch CPU will degrade with more packets to be processed. Once any tenant is under attack (e.g., SYN flood) and sends massive control packets, both the congested PCIe and busy CPU make other innocent tenants suffer severe performance degradation. Thus, *CoFilter* should provide control packets isolation to prevent any tenant from taking up excessive PCIe bandwidth and CPU resources. Otherwise, the other innocent tenants might undergo unfair performance degradation during connection establishment and release.

The second one refers to *data packet isolation*. *CoFilter* achieves high performance via storing connection states and processing the majority data packets on the ASIC. However, the ASIC has limited memory (mainly SRAM) for state storage which are shared by all tenants. If any tenant has too many connections and exhausts the SRAM, connections from other tenants have to fall back on the CPU and data packets are also unfairly directed to the low-performance software datapath. As a result, the innocent tenants that take up fewer network resources actually have to suffer serious performance reduction caused by the aggressive tenant. Therefore, *CoFilter* should prevent the aggressive tenants from taking up too much ASIC memory to protect the performance of tenants that occupy few resources.

Next, we will respectively introduce the *per-tenant control packet rate limit* and *tenant-aware flow migration* to solve the above two problems and guarantee performance isolation. Note that we focus on the specific performance isolation problems caused by our co-design. However, some general-purpose aspects such as bandwidth are rather typical in latest software-defined frameworks and thus not our major concerns.

5.1 Per-Tenant Packet Rate Limit

To provide control packet isolation among different tenants, we design a dynamic per-tenant rate limit mechanism in the switching ASIC. We use meters in the Color-Blind mode offered by the programmable switch to detect the congestion. We define $N + 1$ meters for N tenants. $meter_0$ is used for the overall control packet rate, and $meter_1$ to $meter_N$ are used for the N tenants. Although the total rate should be the sum of measurements for all N tenants, we still need N +1 meters because it is difficult to add up these N meters to get the sum in real time on the ASIC. The meters can mark packets as green, yellow, or red based on two threshold rates, Peak Information Rate (PIR) and Committed Information Rate (CIR), and their associated burst sizes (PBS and CBS) [13]. Congestion occurs when the overall control packet rate reaches its respective threshold. We can then rate limit on the tenants whose control packet rates reach their respective threshold rates. Note that our rate limiting approach includes both congestion detection and rate limiting. Congestion notification is not our primary concern, but can be easily supported on programmable switches, for example, by sending notification packets or marking packets when congestion is detected.

5.2 Tenant-Aware Flow Migration

To provide data packet isolation among tenants, intuitively we should reallocate SRAM on the ASIC, i.e., migrate flows to CPU from aggressive tenants that occupy more SRAM, so that released memory can be used by other tenants.

However, flow migration is far from straightforward. First, we need to decide how many flows from each tenant should be migrated to the CPU. Our principal is to take tenant priorities into consideration and migrate more flows from tenants with lower priorities. Second, we need to decide which flows should be migrated to the CPU for each tenant. Our principle is to keep as much traffic on the ASIC as possible to reduce the overall performance degradation from flow migration. Next we will carefully design the flow migration process based on the above two aspects.

Goal 1: Take tenant priority into consideration when deciding tenant's flows should be migrated. The cloud data centers often hope to provide differential services for different tenants to make more economic benefits. In this case each tenant is given a priority. Tenants with higher priorities can be allocated with more ASIC resources.

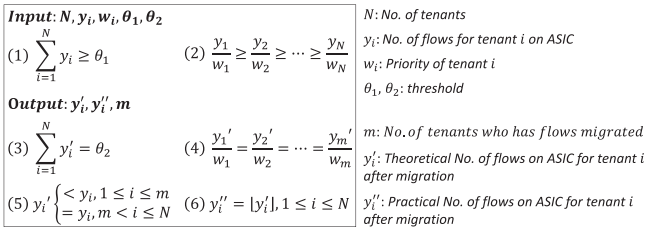


Fig. 6. Flow migration problem.

Therefore, we define the priority-based flow migration problem as shown in Fig. 6. Formula (1) and (3) give the trigger condition and ultimate goal of flow migration. We define two SRAM utilization thresholds, i.e., the quantity of flows stored on the ASIC. Specifically, θ_1 is the SRAM utilization that triggers flow migration, and θ_2 is the expected resultant utilization after migration. When the SRAM utilization reaches θ_1, we should migrate at least $\theta_1 - \theta_2$ flows to lower the utilization to θ_2. Intuitively, given the flow number y_i and priority w_i of tenant i, for tenants with the same number of flows on the ASIC, we should migrate fewer flows from those with higher priorities. Also, for tenants with the same priority, we should migrate more flows from those having more flows on the ASIC. *Therefore, we use $\frac{y_i}{w_i}$ as the criterion to determine the number of flows to be migrated for tenant i.* Assume that tenants are sorted in the order of decreasing $\frac{y_i}{w_i}$ as expressed in Formula (2). First, we migrate flows from the first tenant with the largest $\frac{y_i}{w_i}$, until it has the same $\frac{y_i}{w_i}$ as the second tenant. Then we migrate flows from the first two tenants until they have the same $\frac{y_i}{w_i}$ as the third one. And so on until the total flow number on the ASIC reduces to the security threshold θ_2. Finally, only flows from the first m tenants are migrated, while the last $N - m$ tenants have no flows to be migrated because their $\frac{y_i}{w_i}$ is small enough, as shown in Formula (4) and (5). Based on the above procedure, the calculated flow number for each tenant $y_i^{'}$ may be a decimal instead of an integer. Therefore, the final number of flows for each tenant $y_i^{''}$ is a number rounded down to $y_i^{'}$ as shown in Formula (6). We summarize the above procedure as the Basic Algorithm in Fig. 7. However, this intuitive algorithm brings a high complexity of $\Theta(\theta_1 - \theta_2)$, since the concurrent flow number can be tens of millions.

We further optimize the algorithm and reduce the complexity to $\Theta(N^2)$. Considering that a BMS server is for exclusive occupation by one tenant, the tenant number N in a rack should not be larger than the BMS number, e.g., a few dozen. Therefore the complexity can be greatly reduced. The key idea of the Optimized Algorithm is to traverse all possible values of m from 1 to N and to calculate whether m can satisfy. Only the first m tenants with the largest $\frac{y}{w}$ have flows to be migrated, and their $\frac{y}{w}$ after migration should still be larger than the largest $\frac{y}{w}$ among the last $N - m$ tenants, which is $\frac{y_{m+1}}{w_{m+1}}$, denoted as u_{min}. Therefore, there are at least $U_{min} = \sum_{i=1}^{m} u_{min} * w_i + \sum_{i=m+1}^{N} y_i$ flows on the ASIC after migration. If U_{min} is larger than θ_2, the current m is infeasible. Otherwise, we can find an object u and $u > u_{min}$, so that if the first m tenants migrate $y - u * w_i$ flows, the resultant $\frac{y}{w} = u$ should be larger than u_{min}. This algorithm traverses N possible values for m, and has a complexity of $\Theta(N)$ in line 8 during each iteration. Therefore, the Optimized Algorithm brings a $\Theta(N^2)$ complexity.

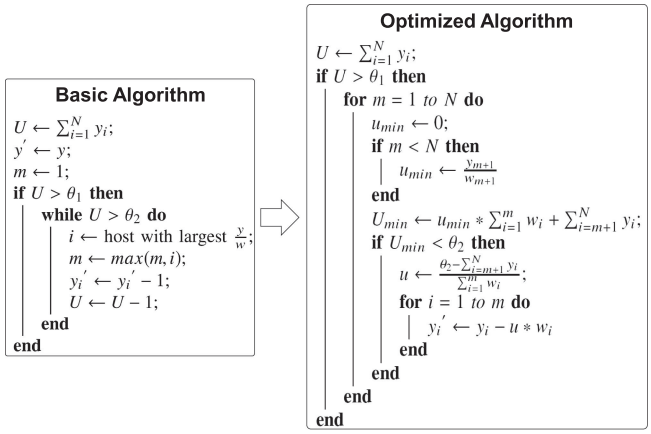


Fig. 7. Priority-based flow migration algorithm.

Goal 2: Optimize overall flow performance when deciding which flows should be migrated for each tenant. So far, we have designed the flow migration algorithm to speculate how many flows should be migrated for each tenant from a global perspective. Now we need to decide which flows should be migrated for each tenant.

To make full use of the high-performance ASIC and lower the performance reduction brought by migrating flows to CPU, we hope to *keep as much traffic on the ASIC as possible*. An intuitive method would be migrating flows with the smallest sizes to CPU. However, this needs per-flow counters and an exact match-action table entry for each migrated flow, which bring large memory overhead. We use LPM instead of exact match to match migrated flows. Specifically, we maintain per-prefix flow size counters on the ASIC and migrate all flows under the common prefix each time instead of just one flow. Larger-length prefixes can be merged into smaller ones, and vice versa, thus can save much memory. We use a *Migrate_Tbl* which contains LPM entries to identify flows that should be migrated. The largest prefix length is defined by users and is taken as the basic prefix unit for prefix merging and splitting. The counters on the ASIC record flow sizes and are periodically cleared by the CPU. The CPU also periodically pulls up-to-date counter values and chooses the prefix with the smallest size to migrate.

We make classified discussions about the dynamic flow migration process as shown in Fig. 8. First, when a new flow f_1 arrives and if the dataplane utilization exceeds θ_1, the flow migration algorithm outputs $y_i^{'}$, which is the supposed number of flows on the ASIC for each tenant after flow migration. We traverse all tenants according to the below procedure. Each time we find the prefix p with the smallest flow size. If p can be aggregated with existing migrated prefixes to produce a smaller prefix, we delete the existing entries and add the new entry. Otherwise, we add an entry with the match field p to *Migrate_Tbl*. Second, if an old flow f_2 which is migrated to CPU finishes, we check whether the related *Migrate_Tbl* entries can be deleted or split, so that if flows with this prefix are established again, they can have the opportunity to remain on the ASIC, instead of matching with the entries and going to CPU again. Suppose that prefix p is the match field of the table entry in *Migrate_Tbl* that matches f_2. We iterate all sub-prefixes that contain f_2 (including p itself) in the order

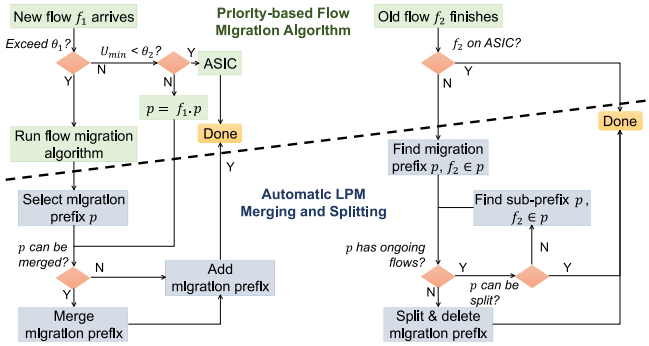


Fig. 8. Dynamic flow migration process.

of increasing prefix length until the largest prefix length, i.e., the basic prefix unit. If we find a sub-prefix p_1 that contains no flows except f_2, we split p into p_1 and other prefixes $\{p_i\}$, delete the p entry and add $\{p_i\}$ entries. Both the prefix splitting and aggregation are necessary. The prefix aggregation can save memory space especially under large-scale attack. The prefix splitting can make larger prefixes easier to be deleted, thus the connections which have finished can return to the ASIC when they are established again.

6 Implementation and Deployment

6.1 Implementation Patterns

The implementation of *CoFilter* relies on both the high-performance switching ASIC and flexible CPU. We can directly implement the control plane functions, such as state transition, as a *CoFilter* Agent process in the native switch CPU (single-switch pattern) or on a remote CPU (hybrid pattern). *CoFilter* Agent communicates with the switching ASIC through a switch driver in the switch OS.

Single-Switch Pattern. The single-switch pattern partitions the stateful processing between the switch ASIC and the switch CPU. The PCIe channel between the switch CPU and switch ASIC is used to transmit packets. Therefore, a programmable switch alone can implement the complete stateful packet filtering, with advantages of low cost and easy management. However, using the switch CPU has two drawbacks. First, the switch CPU might have quite poor computing power and impede the overall performance. To solve this, we can just replace the original weak CPU with a superior one. Second, the PCIe bandwidth between the switch CPU and the ASIC is limited. Thus the rate limit module in Section 4.1 is very necessary to avoid PCIe congestion.

Hybrid Pattern. The hybrid pattern can improve the above two drawbacks of single-switch pattern by introducing an external server. First, the CPU of the external server provides better performance than the native switch CPU. Second, the server and switch are connected with Ethernet links, which have much larger bandwidth than the PCIe channel.

As shown in Fig. 9, the *data packets* are processed directly on the switch ASIC, and the *control packets* are forwarded to the *CoFilter* Agent on the server. *CoFilter* Agent provides stateful processing for *control packets* and indirectly controls the switch ASIC (e.g., install rules and update registers) through a switch driver in the switch OS. We implement

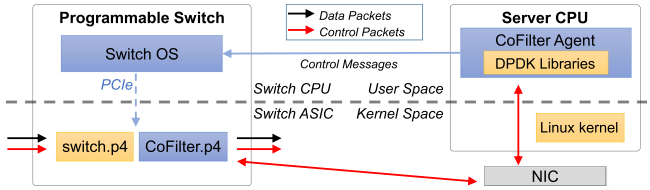


Fig. 9. *CoFilter* Implementation of the hybrid pattern.

CoFilter Agent as a DPDK application, which can bypass the Linux kernel to further improve packet process performance.

6.2 Deployment

Based on the actual needs of data centers, *CoFilter* has two deployment methods.

First, *CoFilter* can be deployed on ToR switches to protect the BMSs under the rack. We simply replace the original ToR switch with a programmable switch deployed with *CoFilter*. Besides the stateful packet filtering of *CoFilter*, the programmable switch can also perform other network functions, e.g., load balancing. If the hybrid pattern is adopted and external server CPUs are required, we can reserve resources on the servers under the ToR switch, and turn them into a remote memory and calculation resource pool, which can be directly accessed through Ethernet links.

Second, considering the limited resources on the programmable switch (as shown in Section 7.5, we can combine multiple *CoFilter* instances as a stateful packet filtering cluster to support more connections. Specifically, *CoFilter* can be deployed on many programmable switches, which combine together to form a cluster to provide stateful packet filtering for more selected BMSs. All traffic should pass through this cluster for traffic filtering before they get into the BMSs. Also, these programmable switches can share resources on one server for the hybrid pattern.

7 Evaluation

7.1 Overview

Setup. The test-bed is shown in Fig. 10. The physical topology consists of a programmable switch and two servers. The switch is equipped with a Tofino chip [9] and an Intel Pentium 4-core 1.60GHz CPU. Both servers are equipped with a 6-core Intel Xeon E5-2620 2.40GHz CPU and are connected to the switch via four 10Gbps links. By reusing the switch and servers, we can implement a logical topology that includes two *CoFilter* prototypes, a single-switch pattern and a hybrid pattern, with *NetFilter* as a comparison.

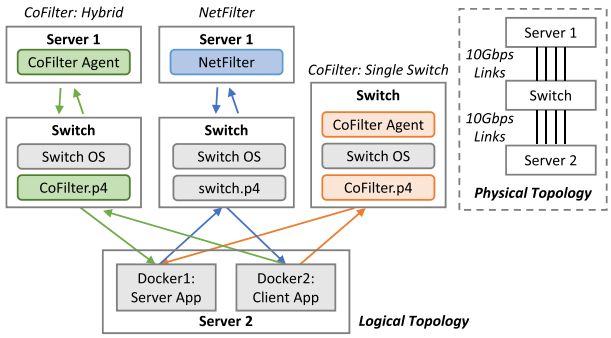
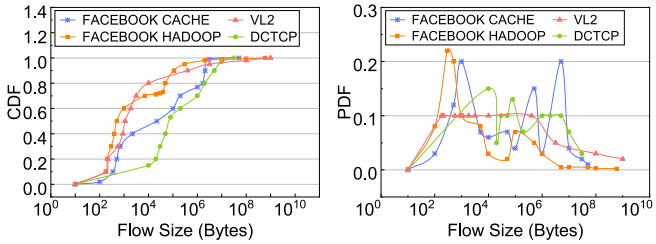
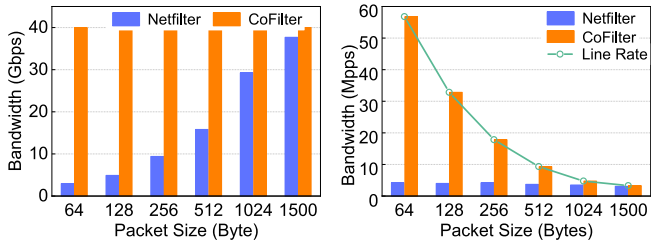


Fig. 10. Test topology used throughout the evaluation.



(a) CDF (b) PDF

Fig. 11. Empirical traffic distributions.



(a) Bytes per second (b) Packets per second

Fig. 13. Throughput comparison.

The switch is deployed with *CoFilter*.p4 for *CoFilter* and the Switch.p4 function for *NetFilter*. For throughput and packet delay test, we use MoonGen [14], a scriptable high-speed packet generator to generate packets in line rate. For end-to-end performance, we create two Docker containers in Server 2 to establish a simple client/server application and use an empirical traffic generator [15] to generate traffic and measure the flow completion time (FCT). For multi-tenant performance isolation testing, we use different IP prefixes to simulate flows from different tenants.

Packet Traces. We use four realistic flow distributions to generate traffic for our experiments, i.e., DCTCP [16], VL2 [17], FACEBOOK CACHE [18], and FACEBOOK HADOOP [18]. All of these traces derive from patterns observed in production data centers. DCTCP is from a web search data center and contains a mix of short and long flows. FACEBOOK CACHE is from Facebook's Cache data center. VL2 and FACEBOOK HADOOP mainly handle offline analysis and data mining. Fig. 11 shows the cumulative distribution function (CDF) and probability density function (PDF) of flow sizes for four traces. Data mining traces (VL2 and FACEBOOK HADOOP) are quite heavy-tailed: most flows are small, while only a small portion of large flows contributes to a substantial portion of the traffic. In comparison, FACEBOOK CACHE and DCTCP are less heavy-tailed.

Objectives. We evaluate *CoFilter* in four aspects. First, we conduct end-to-end experiments and measure FCT of *CoFilter* and *NetFilter* under different workloads (Section 7.2). Second, we perform micro-benchmarking comparisons of *CoFilter* and *NetFilter* concerning throughput and packet delay (Sections 7.3 and 7.4). Third, we make a comprehensive analysis of *CoFilter*'s scalability in terms of resource usage and ASIC capacity, and compare *CoFilter* with *NetFilter* with regards to the CPU utilization (Section 7.5). Finally, we test the effect of the performance isolation proposed in Section 5 (Section 7.6).

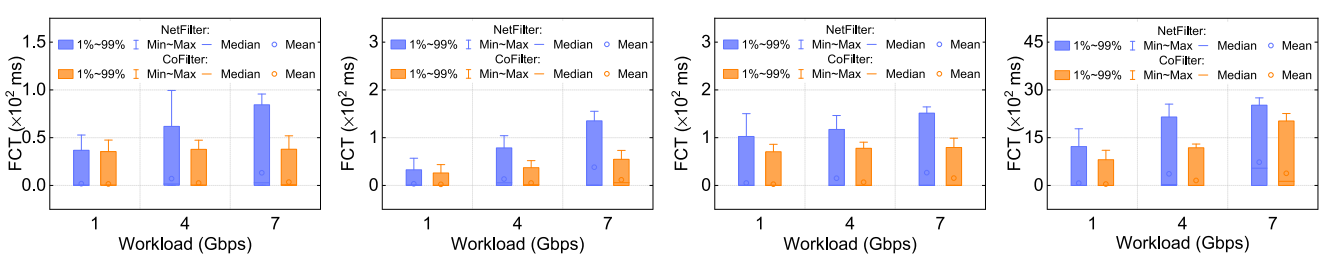
7.2 End-to-End Evaluation

We implement an end-to-end experiment and take FCT as an overall performance metric. We refer to an empirical traffic generator [15] to generate traffic patterns and measure FCT. The client in the Docker container sends flows with size drawn from the empirical flow size distributions of four real-world data center networks, and the server in the other container just replies with a flow with the requested size for each request.

Flow Completion Time. Fig. 12 shows the FCT for four traces at workloads of 1, 4, and 7Gbps. Because control packets take up a small part of a flow, there is little difference between *CoFilter*'s two patterns. With workload increasing, more flows are competing for the same link, and therefore, FCT increases. We can see that *NetFilter* is more vulnerable to the impacts of workload volumes, as the FCT of *NetFilter* increases faster than *CoFilter* with larger workloads. Among the four traces, VL2 has the largest FCT, mainly because that VL2 has more large flows. Under the background workload of 7Gbps, *CoFilter* can lower the average FCT by 10 to 376 microseconds compared with *NetFilter*. We also measure the 99% FCT in consideration of small flows, and we find that the variation trend of the 99% FCT is similar to that of the average FCT. These results indicate that although *CoFilter* has larger control packet delay than *NetFilter*, the overall end-to-end performance of *CoFilter* remarkably wins out because FCT is primarily decided upon the majority data packets.

7.3 Throughput of *CoFilter*

Data Packet Throughput. Fig. 13 shows the throughput of various packet sizes for *CoFilter* and *NetFilter* when processing data packets. To generate small packets in line rate, we use MoonGen (using four 10Gbps ports) with four CPU cores. Because data packets only pass the switching ASIC without going through CPU, the single-switch pattern and hybrid pattern have the same throughput. In DCNs, small packets



(a) FACEBOOK CACHE (b) DCTCP (c) FACEBOOK HADOOP (d) VL2

Fig. 12. Flow completion time comparison under different volumes of background workload.

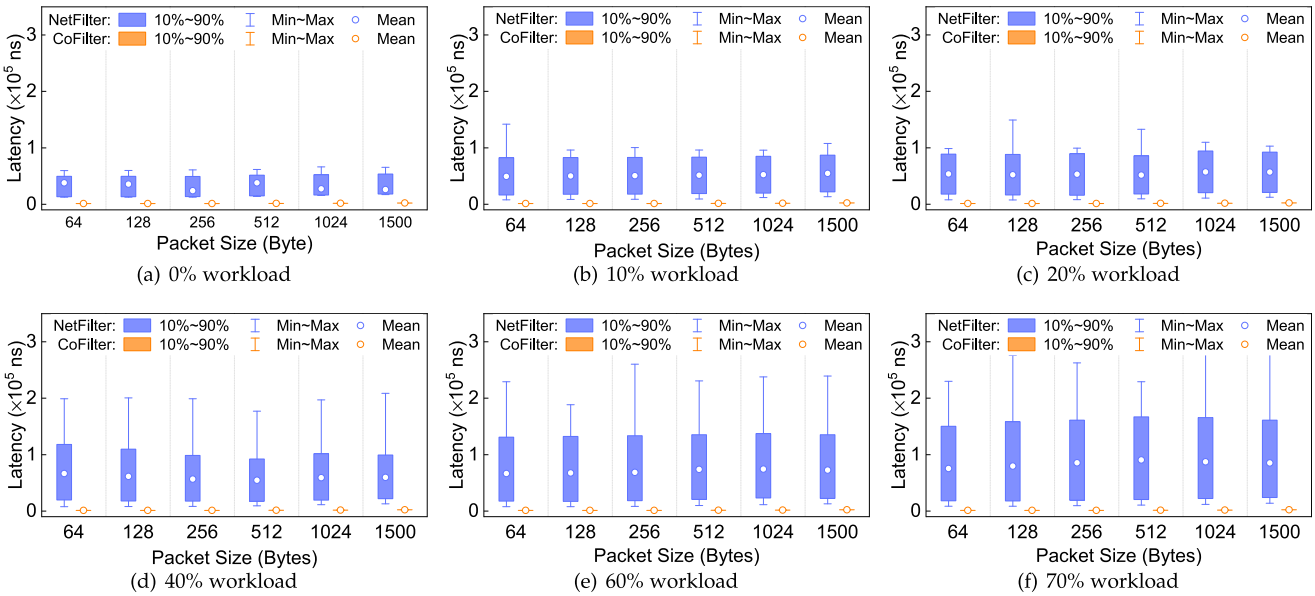


Fig. 14. Data packet delay comparison for different sizes under different volumes of background workload.

with no more than 1000 bytes can take up more than 50% among all packets [19]. At an average packet size of 1500 bytes, *CoFilter* and *NetFilter* have similar throughput. *CoFilter* always achieves full 40Gbps line rate, while *NetFilter* significantly degrades performance for smaller packets. When the packet size come to 64 bytes, *NetFilter* achieves only 3Gbps, 13x smaller than *CoFilter*. In summary, *CoFilter* inherits the high performance of programmable ASICs and greatly improves throughput especially for small packets.

7.4 Delay of *CoFilter*

As for the packet delay experiment, we use MoonGen to start two transmission tasks to send two types of traffic: one for the background workloads of different volumes and one for the prioritized traffic to test packet delay. As control packets and data packets take different paths on *CoFilter*, we measure the delay of them separately.

Data Packet Delay. Fig. 14 compares the delay of data packets under different workloads for different packet sizes. The packet delay of *NetFilter* rises tremendously as background workload increases and also trends larger with larger packet sizes, while *CoFilter* always has small packet delay. The more specific comparison is shown in Fig. 15, which is the delay and delay variation measurement for 1024-byte packets. According to Fig. 15a, as background workload increases from 0% to 70%, the mean delay of *NetFilter* increases from 27 to 87 microseconds, and the 99% delay (P99) increases from 55 to 194 microseconds. In contrast, different delay lines of *CoFilter* coincide with each other and maintain at the level of 1 microsecond. Fig. 15b shows the instantaneous packet delay variation (IPDV) and Standard Deviation (STDDEV). IPDV is the difference of the delay between successive packets [20], and STDDEV is the standard deviation of all packet delays. With background workload increasing, the STDDEV and IPDV of *NetFilter* increase from 10^4 to 10^5 nanoseconds, while those of *CoFilter* remain at around 10 nanoseconds. Compared with *NetFilter*, *CoFilter* considerably reduces data packet delay (20x) and jitter (10^4x).

Control Packet Delay. To evaluate the overhead of processing control packets at switch CPU, we use SYN packets to represent control packets and measure SYN packet delay. Fig. 16 demonstrates the SYN packet delay of *CoFilter* and *NetFilter* under varied background workload. We can see that both the single-switch and hybrid pattern of *CoFilter* keep the delay at a rather stable range, i.e., 200 and 50 microseconds respectively, but the SYN packet delay of the single-switch pattern is larger than that of the hybrid pattern, mainly caused by the poor performance of the native switch CPU. In contrast, *NetFilter* is easier to be influenced by the background traffic. As the workload increases from 0 to 7Gbps, *NetFilter*'s *SYN packet* delay increases by 2.5x, from 25 to 63 microseconds. This is because that for *NetFilter*, data packets are competing for CPU resources with SYN packets, while data packets are not processed by CPU for *CoFilter*.

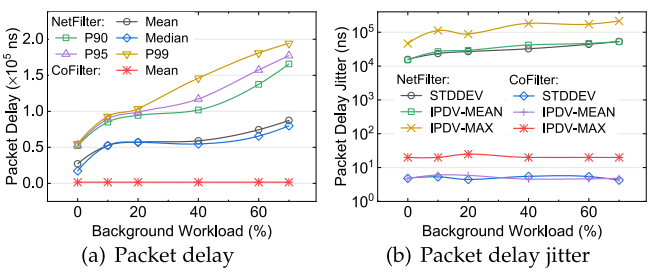


Fig. 15. Data packet delay under different workload.

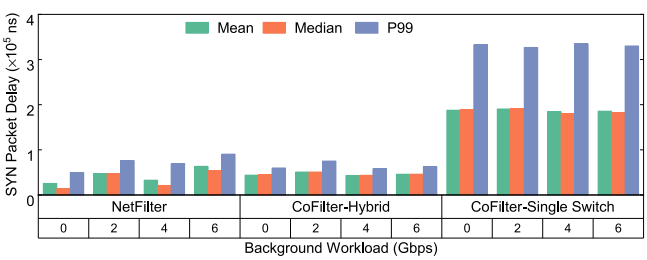


Fig. 16. SYN packet delay variations with varied workloads.

TABLE 1
Hardware Resources Consumed by *CoFilter* With the Capacity of 2M Connections and a Tolerance for 10k Hash Collision

Resources	Usage Percentage
Match Crossbar	18.2%
Static Random Access Memory	43.7%
Ternary Content Addressable Memory	11.5%
Very Long Instruction Word Actions	19.3%
Hash Bits	25.8%
Stateful Arithmetic and Logic Units	9.1%
Packet Header Vector	24.5%

The values are normalized by the usage of Switch.p4 [21].

TABLE 2
Cost Comparison Between *CoFilter* and Hardware-Based Solutions

Metrics (Per Tbps)	Equipment Cost	Power Cost	Stateful Throughput
CoFilter	$3,600	150W	1Tbps
Cisco Firepower 4110 Stateful Firewall	$288,989	3,520W	112Gbps
Netronome Agilio CX 2x25GbE SmartNICs	$10,792	$<$ 512W	135Gbps
Mellanox ConnectX-5 EN 2x25GbE NIC	$14,540	465W	59Gbps

From above, we conclude that for long-lived connections, *CoFilter* brings significant performance improvement by processing *data packets* on the ASIC. For short-lived connections, the performance is hardly improved, since most packets are processed on the CPU for both *CoFilter* and NetFilter. However, when short-lived and long-lived connections co-exist, *CoFilter* still outperforms NetFilter, because the packets of all connections compete for CPU resources on NetFilter, while only the *control packets* occupy CPU resources on *CoFilter*.

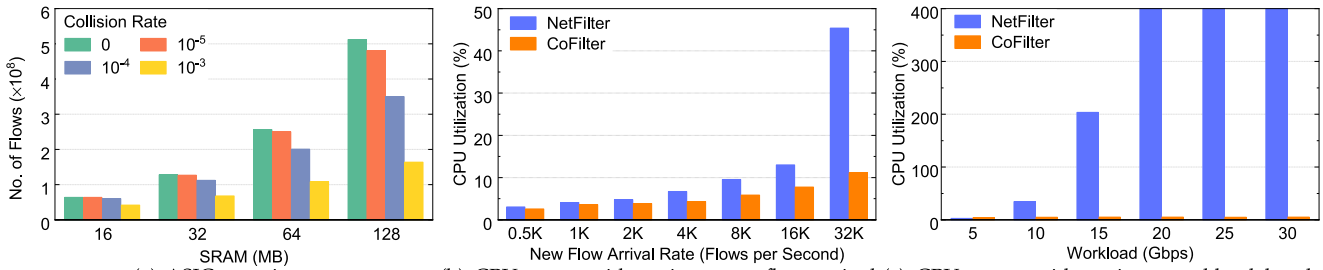
7.5 Scalability and Cost of *CoFilter*

ASIC Resource Usage. We evaluate the hardware resources that *CoFilter* needs on top of the baseline switch.p4 [21]. Table 1 shows the normalized hardware resources usage of *CoFilter* when storing 2M ($M = 10^6$) connections and 10K ($K = 10^3$) *Col_Tbl* entries, which means that 10k collided connections of 2M in total are acceptable. We can see that *CoFilter* mainly consumes 43.7% SRAM for exact match action tables (*Col_Tbl*) and stateful memory, and double Meter ALU for packet rate limit. Besides, *CoFilter* needs 18.2% match crossbars to select keys for the exact match, 11.5% TCAM for LPM match, 19.3% very long instruction word actions to implement compound actions, 25.8% hash bits for table lookup and 24.5% packet header vector to transmit metadata across different stages. We can see that SRAM and meter ALU are the primary consumption for *CoFilter*.

ASIC Capacity. We test the capacity of connections with varied resource sizes. Fig. 17a demonstrates the relationship between the required SRAM and 2-bit connection states that *CoFilter* can accommodate, considering the additional SRAM consumption of *Col_Tbl* for collided connections. We define the *collision rate* θ as the collision probability for connections. For example, $\theta = 10^{-5}$ means that we need one *Col_Tbl* entry for every 10^5 connections. If $\theta = 0$, i.e., no connection collides with each other and all the xMB SRAM on the ASIC can be used to store 2-bit connection states, theoretically we can store $4x * 10^6$ connections, ignoring the additional SRAM needed in hardware implementation. Furthermore, considering hash collisions, i.e., when $\theta > 0$, *Col_Tbl* entries also requires SRAM. As θ increases, more *Col_Tbl* entries are needed to rewrite collided hash values, and therefore fewer connections can be stored. As is shown in this figure, even when θ reaches 10^{-3}, *CoFilter* can store $> 10^7$ connections with 16MB SRAM, and $> 10^8$ connections with 64MB SRAM.

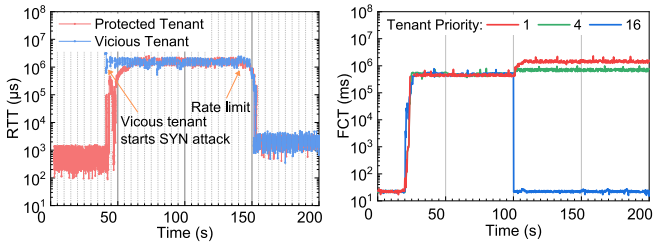
CPU Resource Usage. We compare the CPU utilization of *NetFilter* and single-switch pattern *CoFilter* under different circumstances. We first vary the arrival rate of new flows, as shown in Fig. 17b. With new flow arrival rate increasing, *CoFilter* occupies more CPU because more control packets are sent to the CPU. *NetFilter* also has increasing CPU utilization for the same reason. However, we can see that the new flow arrival rate has a more significant impact on *NetFilter*, which claims up to a 45.4% CPU utilization when the new flow arrival rate is 32K flows/second. Fig. 17c shows the CPU utilization under different workload bandwidth. *NetFilter* achieves up to 400% CPU utilization under the workload of 20Gbps, while *CoFilter* keeps the usage at less than 13% all along. Notice that CPU usage $> 100\%$ is because of the *NetFilter* server has four CPU cores. These results indicate that the server CPU of *NetFilter* becomes the bottleneck at high network speeds and new flow arrival rates, while *CoFilter* can significantly save CPU.

Cost. We compare the equipment and power costs between *CoFilter* and three other hardware-based solutions, as well as the stateful throughput supported, as shown in Table 2. According to [22], a programmable switch costs about $3,600 and 150 Watts (W) per Tbps. Programmable switches can



(a) ASIC capacity (b) CPU usage with various new flow arrival rates (c) CPU usage with various workload bandwidth

Fig. 17. ASIC capacity and CPU resource usage of *CoFilter*.



(a) TCP initial RTT with control packet isolation (b) FCT with data packet isolation

Fig. 18. Performance isolation effect.

always process packets at line rate, so the stateful throughput can be up to 1Tbps. For commodity hardware, we take the Cisco Firepower 4,110 stateful inspection firewall [6] as an example. According to [6], a 320Gbps firewall costs about $90,309 and supports only 35Gbps of stateful throughput. We can conclude that this firewall costs about $288,989 per Tbps, which is 8 times more than *CoFilter*, and supports only 112Gbps of stateful throughput per Tbps, which is 9 times smaller than *CoFilter*. We also analyze two smartNIC-based OVS offload solutions, including the Netronome Agilio CX 2x25GbE and the Mellanox ConnectX-5 EN 2x25GbE. Based on the data from [23], [24], the prices per 50Gbps are $527 and $710, so the prices per Tbps are approximately $10,792 and $14,540. However, the high performance of smartNICs applies only to minimal flows. When there are about 64K flows, the performance of these two smartNICs drops dramatically. At a 64B packet size, they can deliever only 6.6Gbps and 2.9Gbps respectively [25], which means 135Gbps and 59Gbps per Tbps. Based on the above, we can conclude that *CoFilter* has a significant cost advantage over other hardware-based solutions.

7.6 Performance Isolation of *CoFilter*

In this part, we test the effectiveness of two performance isolation techniques proposed in Section 5.

Control Packet Isolation of Per-Tenant Packet Rate Limit. The per-tenant rate limit in Section 5.1 isolates control packets by imposing rate limiting on tenants that send large numbers of control packets. The most direct manifestation of too many control packets is the increased packet delay of control packets. Therefore, we use the TCP initial RTT, i.e., the time between the client sending SYN packets and receiving SYN/ACK packets, as the metric to perform the evaluation. We add two tenants to test the performance affection of one vicious tenant on the other innocent tenant. We use the IP address prefix to distinguish the traffic of different tenants. Specifically, tenant one is a protected tenant that sends SYN packets at a low rate, while tenant two is a malicious tenant that suffers or initiates a SYN flooding attack. We test the RTT variations of both tenants when tenant two maliciously initiates an attack and when we perform control packet isolation. Fig. 18a shows RTT as a function of time. At the first stage starting from time $0s$, only tenant one has 5000 new flow arrivals per second, i.e., 5000 SYN packets are sent per second. We can see that when there is no attack, tenant one's RTT is rather small, ranging from 100 microseconds to 1000 microseconds. At the second stage, starting at $40s$, tenant two is suddenly attacked by SYN flooding with $5M$ new connections per second. The PCIe channel is soon saturated by tenant two, and we can see that RTT increases dramatically to more than 10^6 microseconds. That is to say, without isolation, the performance of tenant one is heavily damaged by tenant two. At the third stage, we rate limit tenant two by dropping packets and reducing the rate of tenant two to the same as that of tenant one. As the figure shows, with control packet isolation, RTT of tenant one is largely improved, dropping by $2 \sim 3$ magnitudes.

Data Packet Isolation of Tenant-Aware Flow Migration. To evaluate the data packet isolation effect of the flow migration in Section 5.2, we use FCT as the metric and test FCT variations without and with isolation. We add thirty tenants with the same priority but different flow sending patterns. We set the thresholds θ_1 and θ_2 in the flow migration algorithm, as 10K and 5K, respectively. Fig. 18b shows the average FCT versus time for these tenants. At the beginning, all tenants send short TCP connections (evenly distributed between 1 and 10 kB) with an average load of 1Gbps. We can see that the FCT is rather small ($10 \sim 100$ microseconds) because short connections can be soon finished and the ASIC memory are released for newly-arrived connections. At time $20s$, these tenants suddenly send different numbers (100, 400, 1600) of long-lived connections which last for more than 200 seconds. We can see that the FCT of all tenants increase to larger than 10^5 milliseconds. This is because that when no isolation is applied, the subsequent new-coming short connections are migrated indiscriminately to the CPU as a result of the long-lived connections occupying the ASIC memory without releasing. We apply the flow migration algorithm at time $100s$. Since the ASIC utilization reaches θ_1, the flow migration algorithm calculates the supposed numbers of flows for each tenant and migrates flows from the ASIC to the CPU. The tenants with the smallest number of long-lived connections have the least flows on the ASIC, and no migration entries are installed to the ASIC for them. As shown in the figure, the average FCT decreases to tens of millisecond, because the newly arrived short connections can be directly processed on the ASIC. However, the other twenty tenants have to migrate part of their newly-arrived short connections to the CPU, which causes varying degrees of delay in their FCT.

8 Related Work

In this section, we show related works on stateful packet filter and maintaining states on programmable data planes.

Commodity Stateful Packet Filter. Based on dedicated hardware, the commodity stateful packet filter [4], [5] can provide rich security features and high packet processing bandwidth, but are inevitably expensive. Depending on the bandwidth to cope with, a hardware packet filter can be quite expensive. For example, a security appliance with only 35Gbps stateful packet inspection throughput can cost up to $90K [6]. Once the device needs to be upgraded, the cost may rise even more. Furthermore, commodity stateful packet filter can process packets with user-defined formats and filter packets based on customized patterns. *CoFilter* yields much higher flexibility and is much cheaper compared to stateful packet filters with the same packet bandwidth.

Software-Based Stateful Packet Filter. There are already many software stateful packet filters in the literature. In the early stages, software network packet filters such as

NetFilter/iptables [7] are based on Kernel functions. These software solutions are commonly used but suffer from serious performance problems. Later, to improve the performance of software-based packet processing, DPDK proposes to bypass the kernel and process packets in user space. However, as a general computing platform, CPUs have inherent limitations on packet processing bandwidth. Processing larger traffic requires additional CPU cores, which also introduces a linear increase in cost.

SmartNIC-Based Stateful Packet Filter. To improve performance and release more CPU cores on the server, some work offloads the stateful packet filtering function to programmable NICs, i.e., the smartNIC. The manycore smartNICs, such as Netronome [26] and Cavium, offload stateful packet filtering across many embedded CPU cores. However, they suffer from serious performance issues because the CPU cores add tens of microseconds of additional latency, and their scalability for higher speeds (such as 100Gbps and 200Gbps) looks bleak [27]. The FPGA smartNICs, such as Mellanox [28], offload stateful packet filtering to FPGA. Compared with *CoFilter* using P4-programmable ASICs, FPGA smartNICs (i) are harder to program due to the complexity of FPGA; (ii) provide worse performance, such as < 20 Mpps throughput for 64B packets when scaling to higher flows, while *CoFilter* can promise line rate due to the ASIC design; (iii) are mroe expensive when produced in large volumes and consume more power.

Maintaining States on Programmable Switches. The high performance and flexibility of programmable switches have motivated much research work to enhance network functions and applications [22], [29], [30], [31], [32], [33], [34], [35]. However, maintaining massive states (e.g., per-flow states) on the programmable switches is not scalable due to memory limitations. Some network monitoring and measurement applications, such as UnivMon [36] and Elastic Sketch [37], use sketches that make a good trade-off between memory and accuracy, but focus on estimating statistical traffic metrics, such as flow size. Therefore, they are not suitable for storing accurate per-flow states. There are also some generic solutions for external memory expansion [38], e.g., by using RDMA, which allows programmable switches to leverage external DRAM on servers. Unlike *CoFilter*, external memory expansion does not increase the state capacity on the ASIC, but rather improves the performance of accessing external memory. External memory expansion can be seen as a complementary solution to *CoFilter*.

9 Conclusion

Stateful packet filter is a critical component for BMS cloud but has stringent requirements for both high performance and high scalability on networks. *CoFilter* leverages programmable switches to meet these requirements and proposes a co-design between programmable ASICs and CPU. Furthermore, *CoFilter* overcomes the limited memory and programmability of switching ASICs via process partition and hash optimization, and proposes two performance isolation techniques to enhance the system's resistance to attacks. *CoFilter* inherits the advantages of programmable ASICs, including high throughput, low packet delay, and low cost, while achieving high scalability, low switch CPU usage, and excellent performance isolation as demonstrated by our hardware implementation and evaluation.

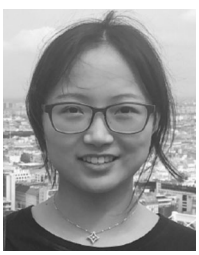
Acknowledgments

A previous version of this article has been published at ICCCN'19 [1] with Best Paper Award.

References

[1] J. Cao *et al.*, "Cofilter: A high-performance switch-accelerated stateful packet filter for bare-metal servers," in *Proc. 28th Int. Conf. Comput. Commun. Netw.*, 2019, pp. 1–9.
[2] E. Castro-Leon *et al.*, *Cloud as a Service: Understanding the Service Innovation Ecosystem*. Berkeley, CA, USA: Apress, 2016.
[3] Wikipedia, "Bare-metal server." Accessed: Dec. 28, 2021. [Online]. Available: https://en.wikipedia.org/wiki/Bare-metal_server
[4] J. Frahim *et al.*, *Cisco ASA: All-in-One Firewall, IPS, Anti-X, and VPN Adaptive Security Appliance*. London, U.K. Pearson Education, 2009.
[5] D. Holmes, "Mitigating DDOS attacks with F5 technology," *Seattle, WA, USA, F5 Netw., Inc*, White Paper, pp. 2099–2104, 2013.
[6] "Cisco firepower 4100 series." Accessed: Dec. 28, 2021. [Online]. Available: https://www.cisco.com/c/en_uk/products/security/firepower-4100-series/index.html
[7] P. N. A. Harald Welte, "netfilter/iptables project." Accessed: Dec. 28, 2021. [Online]. Available: https://www.netfilter.org
[8] R. Miao *et al.*, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2017, pp. 15–28.
[9] Barefoot Networks, "Barefoot tofino." Accessed: Dec. 28, 2021. [Online]. Available: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html
[10] X. Chen *et al.*, "Catching the microburst culprits with snappy," in *Proc. Afternoon Workshop Self-Driving Netw.*, 2018, pp. 22–28.
[11] R. Ben-Basat *et al.*, "Efficient measurement on programmable switches using probabilistic recirculation," in *Proc. IEEE 26th Int. Conf. Netw. Protoc.*, IEEE, 2018, pp. 313–323.
[12] P. Bosshart *et al.*, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," in *Proc. ACM SIGCOMM Conf. SIGCOMM*, 2013, pp. 99–110.
[13] J. Heinanen *et al.*, "A two rate three color marker," Internet Requests. Comments, Internet Eng. Task Force, RFC 2698, Sep. 1999. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2698.txt
[14] P. Emmerich *et al.*, "Moongen: A scriptable high-speed packet generator," in *Proc. Internet Meas. Conf.*, 2015, pp. 275–287.
[15] M. Alizadeh, "Empirical traffic generator," 2017. [Online]. Available: https://github.com/datacenter/ empirical-traffic-gen.
[16] M. Alizadeh *et al.*, "Data center TCP (DCTCP)," in *Proc. ACM SIGCOMM Comput. Commun. Rev.*, 2010, pp. 63–74.
[17] A. Greenberg *et al.*, "Vl2: A scalable and flexible data center network," in *Proc. ACM SIGCOMM Comput. Commun. Rev.*, 2009, pp. 51–62.
[18] A. Sapio *et al.*, "Daiet: A system for data aggregation inside the network," in *Proc. Symp. Cloud Comput.*, 2017, p. 626.
[19] T. Benson *et al.*, "Network traffic characteristics of data centers in the wild," in *Proc. 10th ACM SIGCOMM Conf. Internet Meas.*, 2010, pp. 267–280.
[20] Wikipedia, "Packet delay variation," [Online]. Available: https://en.wikipedia.org/wiki/Packet_delay_variation
[21] The P4 Language Consortium, "Consolidated switch repository," [Online]. Available: https://github.com/p4lang/switch
[22] Y. Zhou *et al.*, "Hypertester: High-performance network testing driven by programmable switches," in *Proc. 15th Int. Conf. Emerging Netw. Exp. Technol.*, 2019, pp. 30–43.
[23] "Netronome agilio cx dual-port 25 gigabit ethernet smartnic." Accessed: Dec. 28, 2021. [Online]. Available: https://www.colfaxdirect.com/store/pc/viewPrd.asp?idproduct=3144
[24] "Nvidia mcx516a-gcat connectx-5 en adapter card 50gbe." Accessed: Dec. 28, 2021. [Online]. Available: https://store.mellanox.com/products
[25] "Netronome 25gbe smartnics with open vswitch hardware offload performance." Accessed: Dec. 28, 2021. [Online]. Available: https://www.netronome.com/media/documents/WP_Netronome_25GbE_SmartNICs_with_Open_vSwitch_Hardware_Offload.pdf

[26] NETRONOME, "Nfp-6xxx flow processor." Accessed: Dec. 28, 2021. [Online]. Available: https://www.netronome.com/media/documents/PB_NFP-6000-7-20.pdf
[27] D. Firestone *et al.*, "Azure accelerated networking: Smartnics in the public cloud," in *Proc. 15th USENIX Conf. Netw. Syst. Des. Implementation*, 2018, pp. 51–64.
[28] M. TECHNOLOGIES, "Innova - 2 flex programmable network adapter." Accessed: Dec. 28, 2021. [Online]. Available: https://www.mellanox.com/files/doc-2020/pb-innova-2-flex.pdf
[29] N. Katta *et al.*, "Hula: Scalable load balancing using programmable data planes," in *Proc. Symp. SDN Res.*, 2016, pp. 1–12.
[30] Z. Liu *et al.*, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 101–114.
[31] V. Sivaraman *et al.*, "Heavy-hitter detection entirely in the data plane," in *Proc. Symp. SDN Res.*, 2017, pp. 164–176.
[32] Y. Zhou *et al.*, "Keysight: Troubleshooting programmable switches via scalable high-coverage behavior tracking," in *Proc. IEEE 26th Int. Conf. Netw. Protoc.*, 2018, pp. 291–301.
[33] X. Jin *et al.*, "Netcache: Balancing key-value stores with fast in-network caching," in *Proc. 26th Symp. Oper. Syst. Princ.*, 2017, pp. 121–136.
[34] H. T. Dang *et al.*, "Netpaxos: Consensus at network speed," in *Proc. 1st ACM SIGCOMM Symp. Softw. Defined Netw. Res.*, 2015, pp. 1–7.
[35] X. Jin *et al.*, "Netchain: Scale-free sub-RTT coordination," in *Proc. 15th USENIX Conf. Netw. Syst. Des. Implementation*, 2018, pp. 35–49.
[36] Z. Liu *et al.*, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 101–114.
[37] T. Yang *et al.*, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2018, pp. 561–575.
[38] D. Kim *et al.*, "Tea: Enabling state-intensive network functions on programmable switches," in *Proc. Annu. Conf. ACM Special Int. Group Data Commun. Appl., Technol. Architectures, Protoc. Comput. Commun.*, 2020, pp. 90–106.



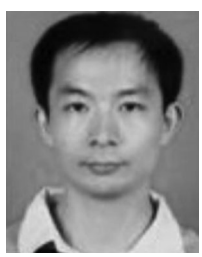
Jiamin Cao received the BS degree in 2018 from the Department of Electronic Engineering, Tsinghua University, Beijing, China, where she is currently working toward the PhD degree with the Institute for Network Sciences and Cyberspace. Her research interests include programmable data planes and network verification.

Ying Liu received the MS degree in computer science and the PhD degree in applied mathematics from Xidian University, China, in 1998 and 2001, respectively. She is currently a professor with Tsinghua University, China. Her research interests include network architecture design, next generation Internet architecture, and routing algorithm and protocol.

Yu Zhou received the PhD degree from the Department of Computer Science and Technology, Tsinghua University, Beijing, China, in 2021. He is currently a network engineer with Network Research Team, Alibaba Group. His research interests include software-defined networking and programmable data planes.

Lin He received the BS degree from the Beijing University of Posts and Telecommunications, Beijing, China, in 2014 and the PhD degree from the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing, China, in 2019. He is currently an assistant professor with the Institute for Network Sciences and Cyberspace, Tsinghua University. His research interests include network architecture and protocol design, Internet accountability and privacy, and network measurement.

Chen Sun received the PhD degree from the Department of Computer Science and Technology, Tsinghua University, Beijing, China, in 2019. He is currently a network engineer with Network Research Team, Alibaba Group. His research interests include software defined networking, network function virtualization, and network monitoring.



Yangyang Wang received the BS degree in computer science and technology from Shandong University, China, in 2002, the MS degree from Capital Normal University, China, in 2005, and the PhD degree from the Department of Computer Science, Tsinghua University, China, in 2013. He is currently a postdoctoral scholar in computer science with Tsinghua University. His research interests include the Internet routing architecture, future Internet design, and SDN.

Mingwe Xu (Member, IEEE) received the BS degree and PhD degree from Tsinghua University. He is currently a full professor with the Department of Computer Science. His research interests include computer network architecture and high-speed router architecture.