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

Jiamin Cao, Ying Liu, Yu Zhou, Chen Sun, Yangyang Wang, Jun Bi
Institute for Network Sciences and Cyberspace, Tsinghua University
Email: cjm18@mails.tsinghua.edu.cn

***Abstract*—As one of the most critical cloud services, Bare-metal Servers introduce stringent performance requirements on data center networks (DCN). Stateful packet filter is an integral DCN component of ensuring connection security for bare-metal servers. However, the off-the-shelf hardware-based and software-based stateful packet filters either are prohibitively costly for cloud DCNs or introduce significant performance bottlenecks. In this paper, we present *CoFilter*, which employs cheap programmable switches to accelerate the stateful packet filter for bare-metal servers. *CoFilter* consists of two key designs. First, to support complex stateful packet filtering logic in programmability-limited switching ASICs, *CoFilter* partitions the stateful packet filtering logic between programmable ASICs and switch CPU. Most packets are directly processed in switching ASICs to achieve high performance, while only a small number of packets go to switch CPU for connection tracking. Second, to track massive connections with constrained hardware memory, *CoFilter* employs hash to compress connection states and provides an efficient settlement for hash collisions. We build a prototype of *CoFilter* and evaluate it on the Tofino switch under various data center traffic traces with real-world flow distribution. The evaluation shows that *CoFilter* largely outperforms NetFilter, *i.e.,* forwarding packets at line rate (13x throughput of NetFilter), keeping packet delay at 1us, and freeing a significant quantity of CPU cores. Furthermore, *CoFilter* presents great scalability and accommodates over ten million connections with only 16MB SRAM.**

***Index Terms*—bare metal server; stateful packet filter; programmable switch**

I. Introduction

The Bare-metal Server (BMS) is becoming an increasingly important service for data center networks (DCN) [1]. Via offering customers sole access to the entire physical server, instead of running a hypervisor or being virtualized, the BMS provides an excellent single-tenant environment, raw processing power, and high reliability [2]. For high-performance computing and data-intensive applications which are delay sensitive, BMS cloud is far better than virtualized cloud services to achieve high efficiency and high performance.

BMSs are appealing but also introduce unique 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 many cloud applications, building a *high-performance* and *high-scalability* stateful packet filter is very critical to BMS cloud providers.

There are several off-the-shelf solutions which might satisfy the requirement of BMSs on the stateful packet filter. We categorize them into two types. First, *dedicated hardware-based solutions* [3, 4] deliver high performance but incur unacceptable costs. Depending on the bandwidth to cope with, a hardware packet filter can be quite expensive. For example, a security appliance with only 40Gbps stateful packet inspection throughput can cost as much as 66K dollars. The costs may rise even worse once the appliance needs to be upgraded. Second, *software-based solutions* like NetFilter/ConnTrack [5] 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 [6]. In summary, the above solutions either are *prohibitively expensive* or result in a *significant performance compromise*, making them impractical for real-world BMS cloud.

Given the limitations of existing solutions, we propose *CoFilter*, which aims to implement a high-performance and low-cost stateful packet filter with programmable ASICs [7]. Compared with expensive dedicated hardware and low-performance software, programmable ASICs deliver both high performance, including high throughput (Tbps) and low delay (nanosecond), and low cost which is comparable to fixed function switching ASICs with the same forwarding rate. However, designing *CoFilter* is non-trivial, and the primary challenge lies in the disparity between the stringent requirements of the stateful packet filter and the limitations on programmable ASICs in the following two aspects.

Complexity of stateful packet filtering logic vs. limited programmability of programmable ASICs. The stateful packet filter typically tracks TCP connections in the form of the Finite State Machine (FSM). There can be multiple subsequent states at a specific state and altogether tens of state transitions among all states in the FSM. Nevertheless, programmable ASICs have strict constraints on stateful operations [8, 9]. In programmable ASICs, only a small number of stateful Arithmetic Logic Units (ALU) attach to the stateful memory (*i.e.,* SRAM) to read and write values. However, the stateful ALU only supports very limited branching operations. This means that programmable ASICs could only implement very few state update operations, which is insufficient to support up to tens of state transitions in the stateful packet filter FSM.

Scalability requirement for tracking massive connections vs. limited memory space in programmable ASICs. Track-

978-1-7281-1856-7/19/$31.00 ©2019 IEEE

ing states of massive connections requires much memory. To store connection states in registers, an intuitive method would be using exact match-action tables to map each 5-tuple to the index of a register directly. Each table entry includes a 5-tuple (104 bits for IPv4 connections and 296 bits for IPv6 connections) as the match key and some additional action data bits. Ten million such entries take hundreds of MB SRAM, which is far more than tens of MB SRAM available in the latest generation of switching ASICs [6].

CoFilter addresses the above two challenges via the following two key designs.

(1) **Implementing high-performance stateful packet filtering via process partition between programmable switching ASICs and switch CPU.** We classify the packets of a connection into two categories, a small number of *control packets* that trigger state transitions and should be tracked, such as SYN, FIN packets, and the majority *data packets* that only need to be inspected and filtered according to current connection states. *Control packets* are processed by the switch CPU, which provides powerful process capability for connection tracking, and meanwhile updates the state stored in ASICs. *Data packets* can be directly processed based on the state stored in ASICs to guarantee high performance.

(2) **Achieving high scalability by hash compression and hash collision settlement.** To overcome the challenge of limited memory space in programmable ASICs, we exploit a general hash operation to save memory, and introduce exact match tables for hash collision resolution. First, we apply hash functions to the 5-tuple of each connection. If there is no hash collision, the hash value can be directly used as the corresponding register index to store states. Otherwise, we use an exact match table, each entry of which matches with the 5-tuple of the collided connection and rewrites the hash value to resolve collision. As the probability of hash collision is quite low, the scarce SRAM in existing ASICs would be sufficient. However, hash collision calculation requires considerable computing power, which is beyond the capability of programmable ASICs. Therefore, we propose to use the switch CPU to calculate hash collisions at the arrival of each connection and insert a table entry to ASICs for collided connections. Furthermore, we adapt our design to the actual memory layout in programmable ASICs to make full use of the fixed per-stage resources.

We conclude our contributions as follows.

- We propose *CoFilter*, a high-performance, scalable, and cheap stateful packet filter based on programmable ASICs for BMS cloud. *CoFilter* can be deployed as Top-of-Rack (ToR) switches in data center networks.
- We address the limited programmability and limited memory challenges with an ingenious co-design which combines the high performance of programmable ASICs and the powerful process capability of switch CPU.
- We implement a prototype of *CoFilter* and deploy it on a Tofino switch. We evaluate the performance and scalability of *CoFilter* and compare *CoFilter* with NetFilter, under various data center traces with real-world flow distributions. Evaluation results show that *CoFilter* can always 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* can accommodate more than 10^7 *connections* in programmable ASICs with only *16MB SRAM* and consumes much fewer CPU cores compared with NetFilter.

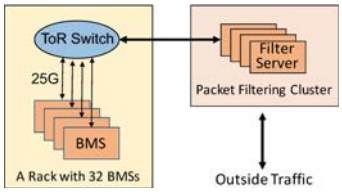


Figure 1. **Software-based packet filter for BMS cloud.**

II. Background on Stateful Packet Filter

In this section, we first present the background on the stateful packet filter. Next, we discuss the limitations of existing software solutions.

A. Stateful Packet Filter

The stateful packet filter has been a critical security element for cloud data centers. Via inspecting each packet and tracking each connection, the stateful packet filter can achieve workload and application-level security. A stateful packet filter generally comprises two parts.

The first is *connection tracking*. The stateful packet filter maintains an 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 inserted to the connection table when a new connection comes, updated as the connection goes and will be removed when the connection finishes.

The second 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.

B. Limitations of Software Stateful Packet Filter

In BMS cloud which aims to provide high-performance services and therefore puts stringent requirements on networks, current software packet filter solutions are far from ideal.

In general, the software stateful packet filter has the following two-fold drawbacks. (1) A high cost of occupying a significant number of servers. As shown in Figure 1, 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. According to our experiment in §IV-B, a server with an Intel Xeon 6-core CPU can achieve 4.22Mpps throughput for 64-byte packets with ConnTrack enabled. In this case, the pure software-based solution needs 1.28K additional packet filter servers per rack, 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

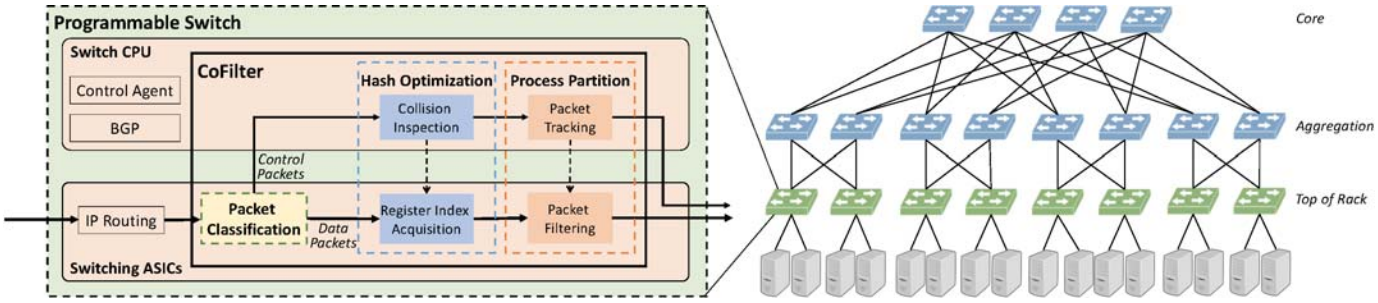


Figure 2. **Architecture and deployment of *CoFilter*.**

round-trip time in cloud data centers [6]. 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.

III. Design of *CoFilter*

A. Design Overview

In this section, we present the *CoFilter* design. The challenges are two-fold, including limited programmability of ASICs to support complex logic, and limited memory to support massive concurrent connections. To this end, *CoFilter* proposes to combine the high performance of programmable ASICs and the powerful process capability of the switch CPU. In essence, *CoFilter* tries to settle the following problem: *In a co-designed system composed of an x86 CPU and P4-programmable ASICs, how to partition the logic and state between the CPU and ASICs to achieve both high performance and high scalability.*

(1) To overcome the limited programmability of ASICs and provide high performance, we slice the stateful packet filtering logic between programmable ASICs and the switch CPU. A small number of *control packets* that trigger state transition, such as SYN and FIN packets, go to the CPU, which provides considerable processing power for connection tracking, while the majority *data packets* are directly processed in switch pipelines to guarantee performance.

(2) To fully utilize the limited memory in ASICs and scale to massive connections, we slice the state storing between ASICs and the switch CPU. We propose a hash-based optimization to map each connection to an individual register instance. The ASICs apply a hash function to the 5-tuple and rewrites the hash values for those collided connections to resolve hash collisions. The new hash values are calculated by the CPU based on the SYN packet for each connection.

Figure 2 depicts how *CoFilter* is deployed in cloud data center networks. We take the fat-tree topology as an example. *CoFilter* can be deployed on ToR switches to provide stateful packet filtering to safeguard BMSs in the rack. *CoFilter* consists of three modules: (1) Packet Classification distinguishes *control packets* and *data packets*. *Control packets* are sent to the switch CPU. (2) Hash Optimization is responsible for connection state storage in switch pipelines. (3) Process Partition is in charge of partitioning the stateful packet filtering logic between ASICs and the switch CPU.

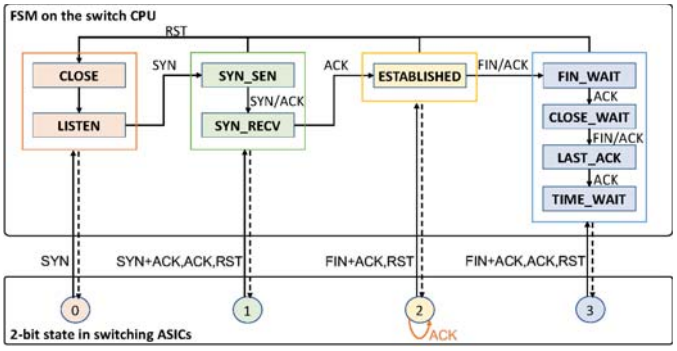


Figure 3. **Process partition of *CoFilter*.**

Packet workflow. Packets are first classified by the Packet Classification module into *control packets* and *data packets*, based on the TCP header. These two types of packets have different workflows: (1) *Control packets* are sent to the switch CPU. The CPU executes hash collision calculation for each new connection and modifies the collision resolution entries in ASICs. Then the CPU updates the connection state for all *control packets* and synchronizes the states stored in ASICs. (2) *Data packets* are directly processed in ASICs. The hash and collision rewriting tables map each connection to an individual register instance. Then the Packet Filtering module drops those illegal packets based on the register values.

B. Process Partition

In this part, *CoFilter* answers the following question: how to slice the stateful packet filtering logic between the programmable ASICs and the CPU.

Observation. 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 (*i.e.,* SYN, SYN/ACK, ACK) in the three-way handshake before a connection is established, four packets (*i.e.,* 2 FIN/ACK and 2 ACK) in the four-way handshake when a connection is about to close, and RST packets. We call the latter *data packets*, including ACK, PSH packets.

Based on the above observation, the insight of *CoFilter* is to partition the stateful packet filtering logic between ASICs and the switch CPU, as demonstrated in Figure 3. The ASICs store the connection states to implement filtering logic for *data packets*, while *control packets* are sent to the switch CPU. The CPU offers powerful processing capability to maintain connection states based on the *control packets* and in turn

updates the state stored in ASICs. In this way, *CoFilter* provides a high-performance process for most *data packets* and also implements the complex state update for those minority *control packets*. To achieve such an efficient process partition, three key problems need to be solved.

Distinguish *control packets* and *data packets*. The ASICs do not have to be concerned about how the state should be updated for different *control packets*. Instead, they only need to screen out *control packets* and send them to the switch CPU. The SYN, SYN/ACK, and FIN/ACK packets can easily be identified by exactly matching with the flag field in the TCP header, while the ACK *control packets* and ACK *data packets* should be differentiated by combining both the TCP flag field and the current connection state.

State compression in ASICs. The ASICs merely process *data packets* and screen out *control packets*. Therefore, the accurate connection states that have no difference in terms of these two functions actually can be merged in ASICs. For example, at the four states in the four-way handshake, we have the same definition for *control packets* and take the same strategies for *data packets*. Therefore, these four states can be merged into a common state. However, at the ESTABLISHED state, the ACK packets should be considered as *data packets*, which is different from the states in the four-way handshake. Therefore ESTABLISHED cannot be merged with the above four states. To this end, we use 2 bits to store four states in ASICs, and each state in ASICs can correspond to multiple realistic states in the CPU. The state 0, 1, 2 and 3 correspond to CLOSE and LISTEN, the three-way handshake, ESTABLISHED, and the four-way handshake, respectively.

Process partition between ASICs and the switch CPU. The CPU serves two functions. The first is to track connection states and perform packet filtering for all uploaded *control packets*. We implement a degraded Connection Tracking of NetFilter in the CPU. The second is to convert the realistic states to the compressed 2-bit states in switch registers and update register values in real time. The ASICs distinguish and upload *control packets* to the CPU, and process *data packets* based on the states stored in registers. As demonstrated in Figure 3, the first SYN and SYN/ACK packets are sent to the CPU as *control packets*. The CPU sets the register state as 1. Then the ACK packet is also considered as a *control packet* and is sent to CPU, which set the register state as 2. The subsequent ACK packets are perceived as *data packets* until the state is set as 3 when the FIN packet arrives. ACK packets are again considered as *control packets* and sent to the CPU until the connection finishes and the state returns to 0.

C. Hash Optimization

In the above part, we introduce how to implement stateful packet filtering logic with the cooperation of ASICs and the switch CPU. The design is based on an assumption that we can easily determine where the state is stored for a given connection in switch pipelines. But actually, ***massive state storing is not easy for programmable ASICs***. In this part, we introduce how *CoFilter* slices the state storing between ASICs and the switch CPU, to utilize the ASIC memory resources and accommodate massive connections fully.

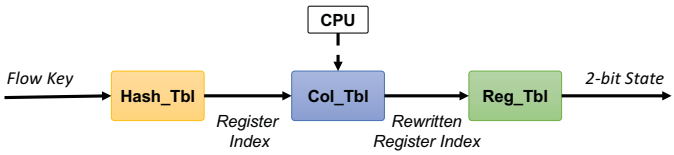


Figure 4. **A CPU-assisted three-phase hash collision settlement scheme on programmable ASICs.**

Programmable ASICs have limited memory resources to store connection states. What is more, these resources are equally allocated to all stages in a pipeline. To make full use of the limited per-stage memory resources and scale to more connections, we propose two key ideas. The first one can overcome the memory challenge for storing massive connections, via reasonable hash compression and collision settlement assisted by the switch CPU. The second one adapts our design to the per-stage memory layout in programmable ASICs. Next, we will respectively illuminate the two ideas.

Hash compression and collision settlement. 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 of instances. An instance of one register array is referenced by its index. A naive connection storage method is to use exact match-action tables to map the flow keys, such as 5-tuples, to register indexes. As analyzed in §II-B, 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 table entries can be freed.

However, hash inevitably brings hash collisions, which mean that two different flow keys are mapped to the same register index. Hash collision resolution requires considerable memory and programmability, and cannot be implemented by programmable ASICs. Therefore, we propose a CPU-assisted three-phase hash collision settlement scheme, as shown in Figure 4. First, the *Hash_Tbl* generates a hash value for each connection. Then *Col_Tbl* exactly matches with flow keys of collided connections and rewrites indexes. The *Col_Tbl* entries are calculated and determined by the CPU, which maintains states of all ongoing connections. Finally, *Reg_Tbl* reads from the corresponding register instance and stores the value into 2-bit state metadata, which will be transmitted to *Filter_Tbl* in Packet Filtering for further processing.

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 CPU calculates hash values for this packet to decide whether the new connection collides with existing connections. If this is a collided connection, the CPU will rewrite the hash value and allocate a new available register instance for the connection. Meanwhile, the CPU will insert an exact match-action table entry to *Col_Tbl* in ASICs, with the flow key as the match field, and the new index as the action data. (2) The subsequent *control packets* also go

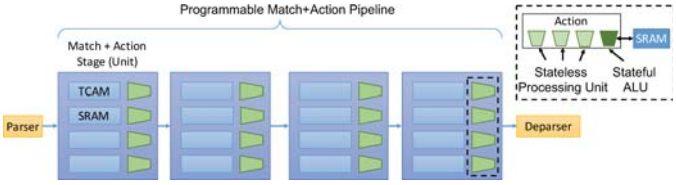


Figure 5. **The architecture of programmable switching ASICs.**

to the CPU but can skip the collision calculation module. (3) The subsequent *data packets* remain in switch pipelines and are matched by the above three tables. (4) When the last ACK packet of a connection arrives, the CPU will delete the corresponding table entry from the *Col_Tbl* if the packet belongs a collided connection.

Cross-stage table placement. We design the hash-based state storing specifically so that it can fit into ASICs and make full use of limited resources. Below, we depict the architecture of programmable ASICs [10] and how to map the tables of *CoFilter*, especially the cross-stage hash collision settlement, to the restrictive programmable ASICs.

As shown in Figure 5, each pipeline of programmable ASICs contains a fixed number of match action stages, which form the packet processing engine. Each stage is organized as several separate tables, each of which can initiate a new match operation on every clock cycle. These tables use SRAM for exact match and TCAM for other types of match. SRAM is also used to implement stateful memories such as registers. The actions are handled by various processing units: a vector of stateless processing units can operate on the packet header or metadata fields with constant parameters, and a small number of stateful processing units associate the table entries with a stateful memory.

This architecture guarantees high performance for P4 programs, but also 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 simultaneity of tables in one stage and dependency of cross-stage tables demands that sequential operations must be implemented in multiple stages in sequential order.

Figure 6 shows how to adapt our three-phase collision settlement scheme to programmable ASICs. We allocate register arrays in multiple stages and define an individual *Reg_Tbl* to access the register array at each stage. An additional *Col_Tbl* pairs with each *Reg_Tbl* and is placed at the stage before *Reg_Tbl*. Therefore, a pipeline with n stages can hold at most $n-3$ pairs of *Col_Tbls* and *Reg_Tbls*, except the first and the last stage. When packets arrive at ASICs: (1) First, a *Classify_Tbl* singles out the *control packets* with the SYN or FIN flag, and sets the ctl metadata as 1. A *Hash_Tbl* maps the flow key (104-bit 5-tuple) to an s-bit sID and an r-bit rID respectively. The sID corresponds to 2^s *Col_Tbls* at the following 2^s stages, and should be limited by $2^s <= (n-3)$. The rID corresponds to 2^r register instances in each register array and should be smaller than the upper register memory limit in one stage. (2) A packet can skip the Register Index Acquisition module if the ctl is 1. In this module, the packet is

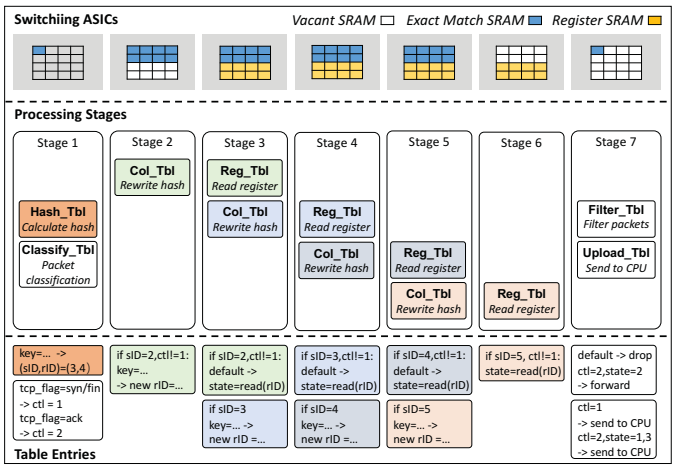


Figure 6. **Hash-based optimization to store massive connections.**

matched by the *Col_Tbl* at the sIDth stage, which will rewrite the rID if the packet belongs to a collided connection, and then matched by *Reg_Tbl* at the $(sID+1)$th stage, which will read the state from the rIDth register instance to the 2-bit $state$ metadata. (3) The *Filter_Tbl* allows the ACK packet when the ctl is set as 2, and otherwise drops them. The *Upload_Tbl* exactly matches with both the ctl and $state$ to send *control packets* to the CPU.

If the register array is full and no available instance can be allocated, the CPU will insert a *Col_Table* entry with the 5-tuple as the match field and uploading to CPU as the action. Therefore, all packets of this connection will match with the special entry and be sent to CPU for stateful processing.

IV. Evaluation

A. Overview

Implementation and testbed. We implement a prototype of *CoFilter* with about 400 lines of P4 code to configure programmable ASICs and 200 lines of C code for packet processing on the switch CPU. Furthermore, we deploy the *CoFilter* prototype on a Tofino switch [7] equipped with 32 100G QSFP ports and 4 Intel Pentium 1.60GHz CPU cores. Our experiment topology is shown in Figure 4. In our experiment setup, we use another two servers, one for the native software-based implementation with NetFilter as a comparison, and the other acts as a BMS. We create two Docker containers in the BMS and establish a simple client/server application between them. Figure 7 shows the testbed topology used in the experiment. Each server is connected to the switch with four 10G links. The servers both have 12-core Intel Xeon E5-2620 2.40GHz CPUs. We use MoonGen [11], a scriptable high-speed packet generator to test throughput and packet delay. For end-to-end performance, we use an empirical traffic generator [12] to generate traffic and measure the flow completion time (FCT).

Workload. We use four realistic flow distributions to generate traffic for our experiments, *i.e.,* DCTCP [13], VL2 [14], FACEBOOK CACHE [15], and FACEBOOK HADOOP [15]. All of these traces derive from patterns observed in production

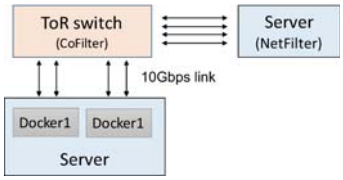


Figure 7. **Testbed topology used throughout the evaluation.**

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.

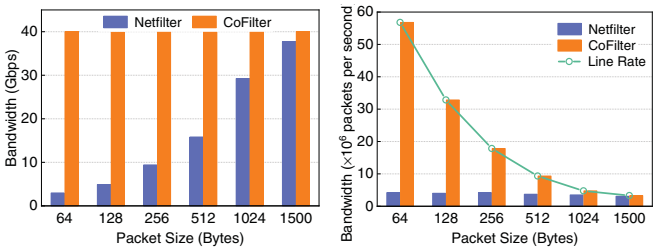
Metrics. We evaluate *CoFilter* in three aspects. First, we perform micro-benchmarking comparisons of *CoFilter* and NetFilter concerning throughput and packet delay (§IV-B). Second, we conduct end-to-end experiments with the above real-world data center packet traces and measure the FCT of *CoFilter* and NetFilter under different workloads (§IV-C). Third, we present a comprehensive analysis of *CoFilter*'s scalability in terms of ASIC resource usage and connection capacity. We also compare *CoFilter* with NetFilter with regards to the CPU utilization (§IV-D).

Overview of results. We summarize major results below.

- *CoFilter* can always forward *data packets* at *line rate* (40Gbps), while NetFilter reduces throughput sharply with smaller packet sizes. For 64-byte packets, NetFilter achieves only 3Gbps, *13x smaller* than *CoFilter*.
- *CoFilter* maintains the data packet delay *at a level of around one microsecond*. In comparison, NetFilter has growing average data packet delay from 27 to more than 80 microseconds with packet size increasing. Furthermore, *CoFilter* keeps the delay fluctuation at a minimal extent, yielding stable packet forwarding performance.
- *CoFilter* outperforms NetFilter and can markedly lower the average FCT by a range from *10 microseconds to more than 150 microseconds* for different traces.
- *CoFilter* mainly requires ASIC memory resources to store connections in the switch, especially the SRAM. With only 16MB SRAM, which is moderate for off-the-shelf programmable ASICs [6], *CoFilter* can accommodate more than *ten million* connections.
- Compared to NetFilter, *CoFilter* has much lower CPU utilization and can *free a significant quantity of CPU cores* from NetFilter servers.

B. Micro Benchmark Evaluation

Throughput. Figure 8 shows the throughput of various packet sizes for *CoFilter* and NetFilter when processing normal data packets. In data center networks, small packets with no more than 1000 bytes can take up more than 50% among all packets [16]. At an average packet size of 1500 bytes, *CoFilter* and NetFilter get similar throughputs. *CoFilter* always achieves the 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 brief, *CoFilter* inherits the high performance



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

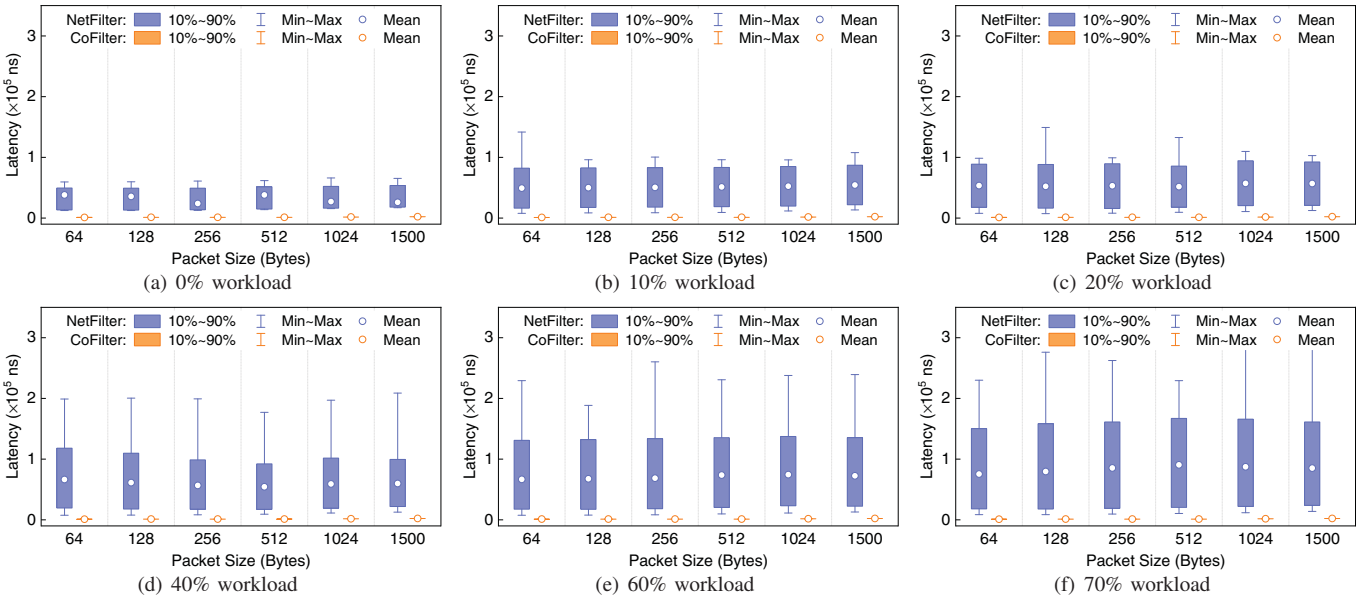
Figure 8. **Throughput comparison.**

of programmable ASICs and greatly improves throughput especially for small packets.

Data packet delay. 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.

Figure 9 compares the delay of data packets under different workloads for different packet sizes. The packet delay of NetFilter rises tremendously as the background workload increases and also trends larger with larger packet sizes, while *CoFilter* keeps the packet delay rather low all along. The more specific comparison is shown in Figure 10, which is the delay and delay variation measurement for 1024-byte packets. According to Figure 10(a), as the 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, the different delay lines of *CoFilter* coincide with each other and maintain at the level of 1 microsecond. Figure 10(b) shows the instantaneous packet delay variation (IPDV) and the Standard Deviation (STDDEV). IPDV is the difference of the delay between successive packets [17], and STDDEV is the standard deviation of all the packet delays. With the 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).

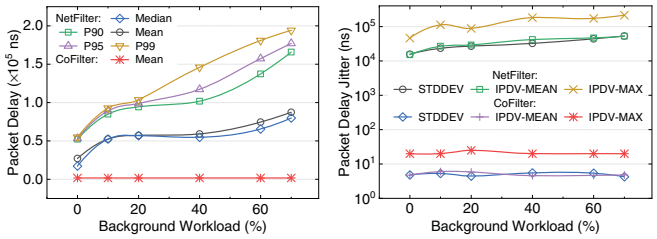
Control packet delay. To evaluate the overhead of processing *control packets* at the switch CPU, we use SYN packets to represent *control packets* and measure the delay. Figure 11 demonstrates the average SYN packet delay of *CoFilter* and NetFilter under varied background workload. The figure shows that *CoFilter* has larger SYN packet delay compared with NetFilter. However, *CoFilter* keeps the delay at a constant value, and the gap between *CoFilter* and NetFilter is narrowed from 7.48x to 3.79x as the workload increases from 0 to 7Gbps. The larger SYN packet delay of *CoFilter* can be mostly attributed to the low-performance switch CPU. Therefore, we can use better CPU to lower the process delay. Furthermore, *control packets* merely take a pretty small proportion of overall traffic for long flows, implying that *the control packet delay increase of CoFilter incurs negligible overhead.*



(a) 0% workload (b) 10% workload (c) 20% workload

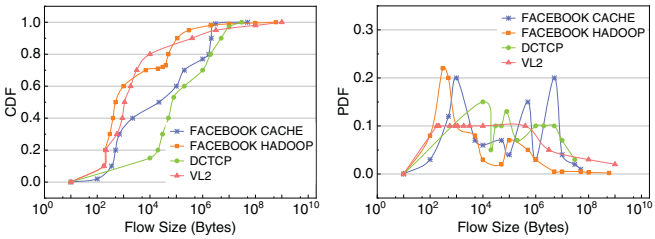
(d) 40% workload (e) 60% workload (f) 70% workload

Figure 9. **Packet delay comparison for different sizes under different volumes of background workload.**



(a) Packet delay (b) Packet delay jitter

Figure 10. **Packet delay and delay variation under different workload.**



(a) CDF (b) PDF

Figure 12. **Empirical traffic distributions.**

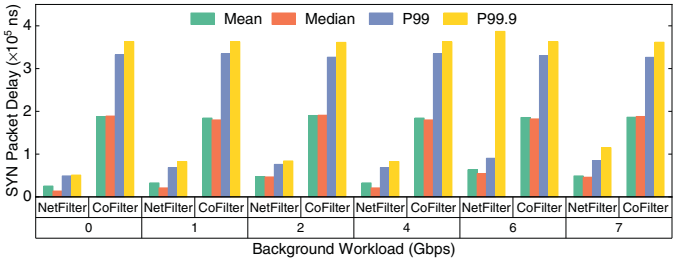


Figure 11. **SYN packet delay variations with varied workloads.**

C. End-to-end Evaluation

We implement an end-to-end experiment and take the FCT as an overall performance metric. We refer to an empirical traffic generator [12] to generate traffic patterns and measure FCT. The client in one Docker container sends flows with size drawn from the empirical flow size distributions of the four real-world data center networks.

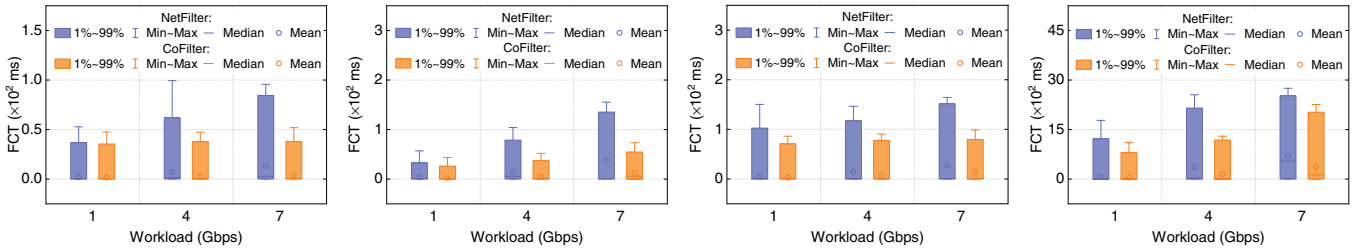
Packet trace overview. Figure 12 shows cumulative distribution function (CDF) and probability density function (PDF) of flow sizes for four traces. Data mining traces (VL2 and FACEBOOK HADOOP) are quite heavy-tailed. Most flows are small, while a small portion of large flows contributes to a substantial portion of the traffic. In comparison, FACEBOOK CACHE and DCTCP are less heavy-tailed.

Flow completion time. Figure 13 shows the FCT for four traces at workloads of 1, 4, and 7Gbps. 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 NetFilter 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. 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 the FCT is primarily decided upon the majority *data packets*.

D. Scalability of CoFilter

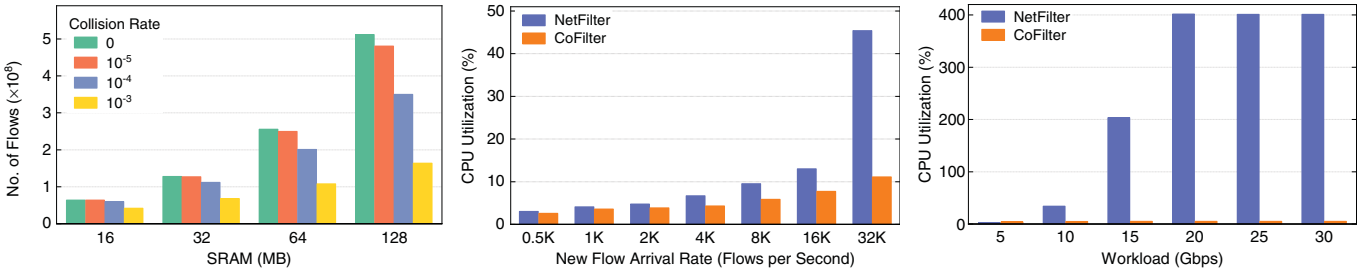
In this part, we first demonstrate the hardware resource usage and connection capacity of *CoFilter*. Then we compare the *CoFilter*'s CPU utilization with that of NetFilter.

ASIC resource usage. We evaluate the hardware resources that *CoFilter* needs on top of the baseline switch.p4 [18]. Table I shows the normalized hardware resources usage of *CoFilter* when storing 2M (10^6) connections and 10k (10^3) *Col_Tbl* entries, which means that 10k collided connections of 2M in total are acceptable. We can see that *CoFilter* mainly consumes 42.3% SRAM for exact match tables (*Col_Tbl*) and register memory. Besides, *CoFilter* needs 20.9% match crossbars to select keys for the exact match, 21.5% very



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

Figure 13. **Flow completion time comparison under different volumes of background workload.**



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

Figure 14. **ASIC capacity and CPU resource usage of *CoFilter*.**

long instruction word actions to implement compound actions, 33.7% hash bits for table lookup, and 30.4% packet header vector to transmit metadata across different stages. In brief, *CoFilter* takes up no more than 50% hardware resources of all types, and SRAM is the primary consumption.

ASIC capacity. We test the capacity of connections with varied resource sizes. Figure 14(a) demonstrates the relationship between the required SRAM and the 2-bit connection states *CoFilter* can accommodate, considering the additional SRAM consumption of *Col_Tbl* to resolve the collided connections. We define the collision rate as the collision probability for connections. For example, a collision rate of 10^{-5} means that we need one *Col_Tbl* entry for every 10^5 connections. If all xMB SRAM is used to store 2-bit connection states, and none of them collides with each other, theoretically we can store $4x * 1024 * 1024$ connections, ignoring the additional memory needed in hardware implementations. Furthermore, considering hash collisions, we need to use additional SRAM to store *Col_Tbl* entries in *CoFilter*. With the collision rate increasing, more *Col_Tbl* entries are needed to rewrite collided hash values, and therefore fewer connections can be stored in the ASICs with the same SRAM. As the figure shows, even when the collision rate is up to 10^{-3}, *CoFilter* can store more than 10^7 connections with 16MB SRAM, and more than 10^8 connections with 64MB SRAM.

Table I
HARDWARE RESOURCES CONSUMED BY *CoFilter* WITH THE CAPACITY OF 2M CONNECTIONS AND A TOLERANCE FOR 10K HASH COLLISION. THE VALUES ARE NORMALIZED BY THE USAGE OF SWITCH.P4 [18].

Resources	Usage Percentage
Match Crossbar	20.9%
Static Random Access Memory	42.3%
Ternary Content Addressable Memory	0%
Very Long Instruction Word Actions	21.5%
Hash Bits	33.7%
Stateful Arithmetic and Logic Units	0%
Packet Header Vector	30.4%

CPU resource usage. We compare the CPU utilization of NetFilter and *CoFilter* under different circumstances. We first vary the arrival rate of new flows, as shown in Figure 14(b). With new flow arrival rate increasing, *CoFilter* occupies more CPU because more *control packets* are sent to switch CPU. NetFilter also has increasing CPU utilization, because more packets need to be tracked. 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. Figure 14(c) shows the CPU utilization under different workload bandwidth. The NetFilter server can achieve up to 400% CPU utilization under a workload of 20Gbps, while *CoFilter* can keep the usage at less than 13% all along. These results indicate that the server CPU of NetFilter becomes a considerable bottleneck at high network speeds and new flow arrival rates, while *CoFilter* can significantly save CPU.

V. Related Work

The performance and flexibility provided by programmable switches have motivated much research work to enhance network functions and applications.

Programmable ASICs to offload network functions. Various P4 programs are deployed on programmable switches to improve network functions in many aspects, such as security, performance, and monitor. SilkRoad [6], Hula [19], and Packet Subscriptions [20] offload load balancing from servers into the programmable switches to improve the overall performance. Univmon [21] and HashPipe [22] leverage the massive switch programmability to enable in-network detection of heavy hitters. Marple [23] proposes a monitoring system that executes complex queries by a key-value store primitive on programmable switch hardware.

Programmable ASICs to accelerate Internet applications. Some recent proposals look at implementing application-level functionality in programmable switches. NetCache [24] proposes a key-value store architecture that leverages the performance and flexibility of programmable switches to handle queries on hot items and balance load across storage nodes. Netpaxos [25] and NetChain [26] improve consensus protocol performance to provide a network implementation of a coordination service in programmable switches.

VI. Conclusion and Discussion

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 switch CPU. Furthermore, *CoFilter* overcomes the limited memory and programmability of switching ASICs via process partition and hash optimization. *CoFilter* inherits the inherent advantages of programmable ASICs, including high throughput, low packet delay, and low cost, while achieving high scalability, high connection capacity, and low switch CPU usage, as demonstrated by our hardware implementation and evaluation. Next, we will discuss some security considerations of *CoFilter*.

Security considerations. The switch CPU has a significant role in *CoFilter* but is potentially vulnerable to denial-of-service attacks, *e.g.,* TCP SYN flood. Besides replacing the original puny switch CPU with a superior one, we can also exploit the meter in programmable ASICs to throttle control packets. The meter measures the control packet rate. Once the rate exceeds a certain threshold, control packets will be dropped instead of going to CPU. In this case, control packets are blocked but existing connections will not be influenced.

Acknowledgement

This research is supported by National Key R&D Program of China (2017YFB0801701), the National Science Foundation of China (61872426), and the National Science Foundation of China (61772307). Ying Liu is the corresponding author. We thank Yiran Zhang and Yunsenxiao Lin for their insightful suggestions and greatly appreciate all anonymous reviewers for their constructive comments.

References

[1] E. Castro-Leon *et al.*, *Cloud as a Service: Understanding the Service Innovation Ecosystem*. Apress, 2016.

[2] Wikipedia, "Bare-metal server," Website, https://en.wikipedia.org/wiki/Bare-metal_server.

[3] J. Frahim and O. Santos, *Cisco ASA: All-in-One Firewall, IPS, Anti-X, and VPN Adaptive Security Appliance*. Pearson Education, 2009.

[4] D. Holmes, "Mitigating ddos attacks with f5 technology," *F5 Networks, Inc*, pp. 2099–2104, 2013.

[5] P. N. A. Harald Welte, "netfilter/iptables project," https://www.netfilter.org.

[6] R. Miao *et al.*, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017.

[7] Barefoot Networks, "Barefoot tofino," Website, https://barefootnetworks.com/technology/#tofino.

[8] X. Chen *et al.*, "Catching the microburst culprits with snappy," in *Proceedings of the Afternoon Workshop on Self-Driving Networks*. ACM, 2018.

[9] R. Ben-Basat *et al.*, "Efficient measurement on programmable switches using probabilistic recirculation," in *ICNP*. IEEE, 2018.

[10] P. Bosshart *et al.*, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *SIGCOMM*. ACM, 2013.

[11] P. Emmerich *et al.*, "Moongen: A scriptable high-speed packet generator," in *IMC*. ACM, 2015.

[12] M. Alizadeh, "Empirical traffic generator," Website, 2017, https://github.com/datacenter/empirical-traffic-gen.

[13] M. Alizadeh *et al.*, "Data center tcp (dctcp)," *ACM SIGCOMM computer communication review*, vol. 41, no. 4, 2011.

[14] A. Greenberg *et al.*, "Vl2: a scalable and flexible data center network," in *ACM SIGCOMM computer communication review*, vol. 39, no. 4. ACM, 2009.

[15] A. Sapio *et al.*, "Daiet: a system for data aggregation inside the network," in *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 2017.

[16] T. Benson *et al.*, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM, 2010, pp. 267–280.

[17] Wikipedia, "Packet delay variation," Website, https://en.wikipedia.org/wiki/Packet_delay_variation.

[18] The P4 Language Consortium, "Consolidated switch repository," Website, https://github.com/p4lang/switch.

[19] N. Katta *et al.*, "Hula: Scalable load balancing using programmable data planes," in *SOSR*. ACM, 2016.

[20] T. Jepsen *et al.*, "Packet subscriptions for programmable asics," in *HotNets*. ACM, 2018.

[21] Z. Liu *et al.*, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 2016.

[22] V. Sivaraman *et al.*, "Heavy-hitter detection entirely in the data plane," in *SOSR*. ACM, 2017.

[23] S. Narayana *et al.*, "Language-directed hardware design for network performance monitoring," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017.

[24] X. Jin *et al.*, "Netcache: Balancing key-value stores with fast in-network caching," in *SOSP*. ACM, 2017.

[25] H. T. Dang *et al.*, "Netpaxos: Consensus at network speed," in *SOSR*. ACM, 2015.

[26] X. Jin *et al.*, "Netchain: Scale-free sub-rtt coordination," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.