SFCache: Hybrid NF Synthesization in Runtime with Rule-Caching in Programmable Switches

Zhihuang Ma, Tingyu Li, Zichen Xu, Nelson L. S. da Fonseca, and Zuqing Zhu, *Fellow, IEEE*

***Abstract*—Data plane programmable (PDP) switches are becoming increasingly popular for network function virtualization (NFV), for their programmability and high packet processing performance. However, the inherent limitations of PDP switches, such as limited memory space, make it challenging to implement certain types of network functions (NFs) (*i.e.*, the stateful ones) on them. This paper proposes SFCache, which combines PDP switches and commodity servers to achieve self-adaptive SFC deployment. SFCache aims to exploit the high packet processing performance of PDP switches while supporting the flexible deployment of a wide range of SFCs (including the stateful ones) with servers. Specifically, SFCache can dynamically improve the packet processing performance of the SFCs that were deployed on servers by selectively caching SFC-level packet processing rules on PDP switches. We design a few key components to facilitate SFCache, including an NF-destructed P4 pipeline that allows customizing packet processing rules in a match-rewrite pattern, a runtime NF synthesis method that can transform a set of NF-level match-rewrite rules into an equivalent SFC-level rule, and a count-min selection strategy to choose the best synthesized rule for being cached in PDP switch pipeline. We prototype SFCache with a PDP switch based on Tofino ASIC and a server, and demonstrate the effectiveness of our proposal experimentally.**

***Index Terms*—Network function virtualization, Service function chain, Programmable data plane switch, Rule caching.**

I. Introduction

RECENTLY, the Internet is undergoing rapid development to adapt to the unprecedented volumes of network traffic, users, and services [1–6]. Middleboxes, also known as network functions (NFs), play a critical role in the process to enhance performance, enforce policies, *etc* [7]. Specifically, a complex network service can be decomposed into atomic NFs based on middleboxes (*e.g.*, firewall and load-balancer), and application traffic will be steered through the NFs in sequence, realizing the network service through a service function chain (SFC) [8, 9]. However, this way of implementing network services with special-purpose middleboxes is becoming increasingly challenging because of the prohibitively-high expenses, intolerable maintenance complexity, and long time-to-market. Therefore, service providers (SPs) have switched to network function virtualization (NFV) [10], which can leverage general-purpose hardware/software platforms (*i.e.*, commodity servers, switches and storages) to realize NFs.

Although NFV greatly improves the flexibility of NF instantiation, it can degrade the packet processing performance of NFs due to the intrinsic performance gap between general-purpose platforms and special-purpose middleboxes, especially when the NFs run on software platforms [11–13]. Hence, NFV acceleration has attracted intensive research efforts, to speed up the packet processing in NFs without sacrificing their flexibility [14]. For instance, a few software NFV platforms [15, 16] and ways to optimize NFV implementations [17–20] have been proposed. Nevertheless, they still cannot completely make up for the aforementioned performance gap, even with the support of high-performance packet I/O techniques such as data plane development kit (DPDK) [21] and netmap [22].

In addition to commodity servers, NFs can also be instantiated on programmable data plane (PDP) switches based on P4 [23] (*e.g.*, those using Tofino ASIC [24]), achieving packet processing throughput at 100 Gbps and extremely low latency ($<$1 μs). Previously, people have realized various specific NFs on PDP switches, such as DDoS mitigation [25], firewall [13], heavy-hitter detection [26, 27], and load-balancer [28, 29]. Meanwhile, studies have also integrated multiple types of NFs into a packet processing pipeline, thereby deploying one SFC on a single PDP switch [30, 31]. However, these proposals were mainly for stateless NFs, which do not need to store or update extensive state information. A PDP switch usually has limited memory resources [31], which makes it difficult to instantiate stateful NFs on it. Therefore, researchers tried to address this challenge by leveraging external memory with the help of remote direct memory access (RDMA) [32, 33]. There were also studies that introduced hybrid processing for packet headers and payloads, where the headers are send to commodity servers for stateful processing and the payloads are stored in PDP switches with stateless pipelines [34, 35].

Nevertheless, the existing approaches on instantiating NFs or SFCs on PDP switches still bear a few drawbacks. First, most of them were designed specifically for certain types of NFs (*e.g.*, the stateful packet filters in [13, 39]) but did not try to achieve the generic support of a wide range of NF types, especially for both stateless and stateful NFs. Second, they either cannot or are not flexible enough to deploy SFCs on a single PDP switch, due to the difficulty of pre-deploying various types of NFs in PDP switch pipelines and realizing agile routing through them in different sequences. Hence, it is relevant and necessary to study how to design an NFV acceleration system that supports a wide range of NF types and facilitates flexible SFC deployment with the NFs. Specifically, the system should be generic, transparent and efficient:

- **Generic:** The system should support a wide range of NF types, including both stateless and stateful NFs, and also adapt to different SFC compositions. When the NF

Z. Ma, T. Li, Z. Xu and Z. Zhu are with the School of Information Science and Technology, University of Science and Technology of China, Hefei, Anhui 230027, P. R. China (email: zqzhu@ieee.org).

N. Fonseca is with the Institute of Computing, State University of Campinas, Campinas, SP 13083-852, Brazil.

Manuscript received on February 2, 2024.

TABLE I
COMPARISON BETWEEN EXISTING NFV ACCELERATION SYSTEMS AND OUR PROPOSED SFCACHE

	Support of NF Types	Flexible SFC Deployment	Packet Processing
OpenNetVM [18]	a wide range of	full support	on server only
Metron [36]	a wide range of	partial support	on OpenFlow switch and server
TEA [32]	no support to payload-involved NFs	no support	on PDP switch
RIBOSOME [34]	no support to payload-involved NFs	partial support	on PDP switch and server
Dejavu [31]	no support to stateful NFs	partial support	on PDP switch
Tiara [29]	stateful load-balancer only	no support	on PDP switch and server
LightNF [37]	a wide range of	no support	on PDP switch and server
P4SFC [38]	a wide range of	full support	on PDP switch with recirculation and server
Our SFCache	a wide range of	full support	on PDP switch and server

arrangement in an SFC changes in runtime, the PDP switch should not have to be reprogrammed and reloaded.
- **Transparent:** When deploying an SFC, the SP does not need to worry about where/how to instantiate the NFs.
- **Efficient:** The system should be able to adjust the deployment of SFCs adaptively according to the current network status, so as to optimize the throughput and latency of packet processing under the resource constraints.

In this paper, in order to realize such an NFV acceleration system, we propose SFCache, which combines PDP switches and general-purpose servers for self-adaptive SFC deployment. Specifically, SFCache dynamically improves the packet processing performance of the SFCs that are initially deployed on servers by selectively caching their SFC-level processing rules on PDP switches. We design a few key techniques to enable SFCache, including the NF-destructed P4 pipeline that allows customizing the packet processing rules of SFCs in a match-rewrite pattern, the runtime NF synthesis method to transform a set of NF-level match-rewrite rules into an equivalent SFC-level rule, and a count-min selection strategy to choose the best synthesized rule to cache in the pipeline of a PDP switch.

We integrate the proposed techniques to prototype SFCache with a PDP switch based on Tofino ASIC and a commodity server, and experimentally demonstrate that the techniques can work together to achieve significant performance gain on packet processing. Specifically, the results obtained with real-world testbed indicate that 1) for a stateless SFC, SFCache increases its throughput by up to 41.32% and reduces its average latency by 48.91%, and 2) for a stateful SFC, SFCache respectively decreases the 99% tail and average values of its latency by up to 91.03% and 90.37%.

The rest of the paper is organized as follows. Section II discusses the related work. We present the system design and operation principle of SFCache in Section III. The implementation of SFCache is described in Section IV. In Section V, we show the experimental demonstrations for performance evaluation. Finally, Section VI summarizes this paper.

II. RELATED WORK

A stateful NF typically needs to maintain the state information for each incoming flow, and thus its implementation has to address the scalability challenge posed by the large number of active flows and limited memory in a PDP switch. According to the traffic statistics of recent investigations [34], 1 Gbps of network traffic can contain hundreds of thousands active flows per second, while a stateful load-balancer needs to consume 64 bytes per flow to store the state information [29]. When it comes to deploy an SFC of multiple NFs on a PDP switch, one needs to route packets through consecutive match-action tables in the switch's pipeline [30, 31], where one or more match-action tables correspond to an NF. However, such a scheme only has limited flexibility. Specifically, when we need to add/update an NF, the pipeline has to be reprogrammed and reloaded, which may interrupt the PDP switch's operation for tens of seconds [40]. Moreover, reloading a PDP switch will clear all the state information on it, even though some of the information should be preserved for managing stateful SFCs.

To address the aforementioned challenges, there have been a number of investigations on NFV acceleration in the literature. OpenNetVM [18] leveraged DPDK for high-speed packet I/O and realized agile SFC deployment. However, as a software-based approach purely relying on servers, its packet processing performance is still not comparable to that of hardware-based approaches. Metron [36] synthesized an SFC by running all of its NFs on a consolidated set of CPU cores, and it processed packets with both OpenFlow switches and servers. However, it does not allow a match field that has been rewritten earlier being used by subsequent NFs, making certain combinations of NFs (*e.g.*, a network address/port translator followed by a load-balancer) not suitable for being implemented in an SFC.

TEA [32] considered PDP switches for NFV acceleration and utilized RDMA to offload the state information of each stateful NF to the external memory on servers, thereby decoupling packet processing from state management. Nevertheless, since its packet processing relies entirely on PDP switches, supporting payload-involved NFs is challenging for it, and it only focuses on instantiating NFs but does not consider SFC deployment. RIBOSOME [34] sent the headers of packets to commodity servers for stateful processing and stored their payloads in PDP switches, which cannot accelerate the processing of payload-involved NFs. Dejavu [31] proposed to leverage in-switch pipeline routing for rearranging the NFs in an SFC dynamically, but it did not consider to support stateful NFs or to add NFs into an SFC in runtime. Tiara [29] only addressed a specific type of NFs (the stateful load-balancer), and designed the mechanism to offload the core processing of these NFs to PDP switches. LightNF [37] also combined PDP switches and servers for SFC deployment, but it did not try to improve the versatility of PDP switch pipeline for flexible SFC deployment (*i.e.*, it might still need to shut down the PDP

switch for reprogramming when deploying new SFCs).

The P4SFC in [38] also tried to realize generic and flexible SFC deployment by combining the advantages of PDP switches and commodity servers. The pipeline design of P4SFC used serially connected and runtime-configurable modules to realize NFs, which would introduce dependencies not only within the modules due to the requirements on configurability but also between tandem modules, exhausting the stages in a PDP switch quickly. Moreover, when a large number of NFs need to be offloaded, P4SFC will have to use the recirculation that makes packets pass through the pipeline repeatedly.

In summary, Table I compares our SFCache with several representative NFV acceleration systems in the literature, and thus highlights the novelty and contribution of this work.

III. System Design

In this section, we first present the overall design of SFCache, then describe the three data paths in the data plane, and finally explain the detailed designs of key components.

A. System Overview and Workflow

Fig. 1 shows the overall system design of SFCache, which consists of a PDP switch and a commodity server in terms of physical devices. Then, SFCache can implement SFCs with hardware- and software-based processing, respectively. The software-based SFC deployment platform on the commodity server is developed by modifying OpenNetVM [18], and can carry SFCs independently. As shown in Fig. 1, the software system follows a multi-process design based on DPDK, where the processes are NFs and an NF Manager. The NF Manager is in charge of initializing shared memory, managing the life-cycle of each NF, and transferring packets between NFs and network interfaces. The communications between NFs and the NF Manager are accomplished through shared memory.

The NF Manager collaborates with the runtime systems of SFCache, which are also exposed to NFs, allowing SFCache to cache the SFC-level packet processing rules on the PDP switch. The receiving (RX) thread of the NF Manager classifies the packets received from the PDP switch based on whether any of their processing rules have been cached in the PDP switch[1]. The packet whose rules have not been cached completely will be forward to the ingress NF of its SFC on the server, while that has already been processed by certain rules cached on the PDP will be steered solely through the remaining NFs of its SFCs. As for the NFs on the server, they can be programmed with a high-level programming language such as C, and be instantiated as processes or containers. Therefore, flexible SFC deployment with runtime readjustment capability can be achieved by the software system, which also contains the user interface of SFCache, to enable SPs to implement, deploy and adjust SFCs. With SFCache, SPs only need to instantiate NFs as software on commodity servers, while offloading the NFs adaptively to explore the benefits of PDP switches is handled automatically, ensuring the transparency of SFC deployment.

On the other hand, the PDP switch carries a packet processing pipeline for instantiating SFCs too. We design a P4 runtime agent and leverage it to accomplish two tasks: 1) processing the *CacheREQ* messages from the NF Manager to cache the corresponding NFs as SFC-level match-rewrite rules in the PDP switch's pipeline, and 2) removing the cached rules that have been idle for a specified time period. The workflow of SFCache is summarized as follows, where the bullet numbers correspond to the steps marked with red circles in Fig. 1.

1) An SP develops NFs and deploys an SFC with them on the commodity server, using a set of application programming interfaces (APIs) provided by SFCache. We design the APIs to enable registering the match-rewrite pattern of each NF, looking up match-action tables to collect packet statistics, and generating *CacheREQs* for caching NFs' match-rewrite rules on the PDP switch.
2) SFCache uses a runtime NF synthesis method to merge all the matches and rewrites that a packet will encounter when traversing its SFC into an SFC-level match-rewrite rule. Meanwhile, it runs count-min selection in the SFCache runtime of each NF to identify hot rules.
3) The caching helper in SFCache generates a *CacheREQ* upon receiving a hot rule message from the SFCache runtime of an NF, and sends it to the P4 runtime agent.
4) The P4 runtime agent on the PDP switch processes the *CacheREQ*, retrieves a set of table entries, and inserts them into the NF-destructed P4 pipeline. Once a rule has been cached, the traffic associated with it will be either fully or partially processed by the PDP switch.

B. Data Paths in Data Plane

SFCache is designed to provide three types of data paths, which are the hardware-only, software-only, and hybrid ones and are explained as follows, to realize generic support of NFs.

- *Hardware-only Data Path*: This type of data path is realized with the PDP switch (*i.e.*, the SFC-level rules have been cached on there), where packets first undergo the corresponding processing of their SFCs and are then sent out directly, bypassing the server. Note that, SFCache only handles the processing rules of the NFs that only involve 5-tuple operations in this data path, while the NFs that need to process packet payloads or complex state transitions (*e.g.*, the deep packet inspection (DPI) and stateful firewalls) are considered as uncacheable and are placed in the software-only data path of SFCache.
- *Software-only Data Path*: This type of data path takes place completely in the server. The packets of an SFC can be processed in this data path for three reasons: 1) all the NFs in the SFC are uncacheable, 2) the processing rules of the SFC are not popular enough for being cached, and 3) there are no enough resources in the hardware-only data path to cache the rules of the SFC.
- *Hybrid Data Path*: This type of data path combines the server and PDP switch in SFCache to address scenarios where an SFC consists of both cacheable and uncacheable

[1]When a packet enters, the PDP switch checks whether its rules have been cached. If not, the PDP switch changes its `EtherType` to $0x2023$ before sending it to the server, and restores the field to $0x0800$ when it is sent back.

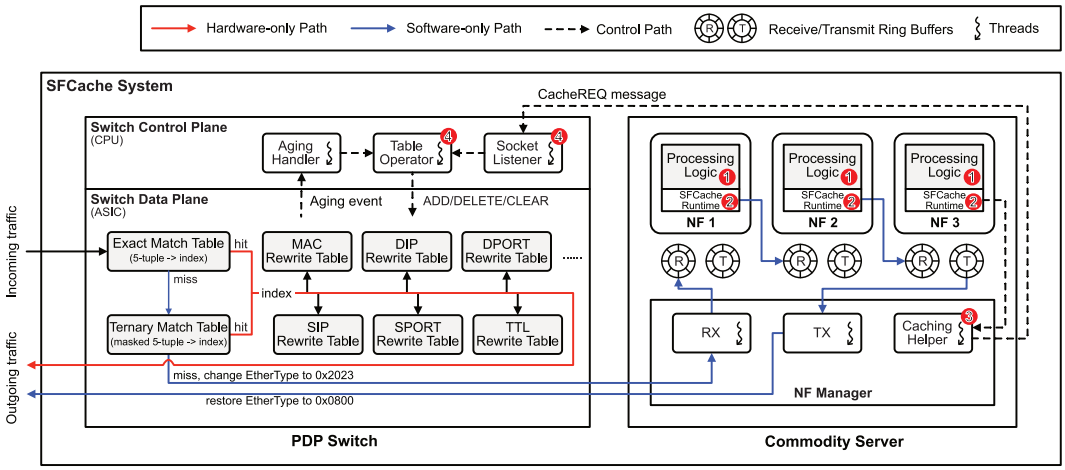

Fig. 1. Overall system design of SFCache.

NFs. In these cases, SFCache is designed to segment the SFC into the parts before, containing, and after the uncacheable NFs. Then, SFCache independently synthesizes and caches the cacheable segments of the SFC on the PDP switch, and leaves the uncacheable NFs on the server.

C. NF-destructed P4 Pipeline

To realize the hardware-only and hybrid data paths, we need to solve two questions: 1) how to make the packet processing pipeline in a PDP switch carry multiple NFs to form an SFC? and 2) how to run multiple SFCs simultaneously in the pipeline of a PDP switch for flexible SFC deployment? Previously, to address the first question, the studies in [30, 31] proposed to implement the required NFs one by one in the pipeline of a PDP switch. However, it is known that supporting many NFs with the limited memory resources in a PDP switch is difficult. As for the second question, the approaches designed in [40, 41] tried to virtualize the memory resources in the pipeline of a PDP switch to accommodate various P4 programs for NFs, but they still bear the issues of degraded performance on bandwidth and latency and extra resource overheads.

Based on the observation that most NFs process packets by the actions of matching and rewriting based on the 5-tuple [20], we propose a pipeline design that ignores individual NFs but focuses on the process of matching and rewriting, as shown in Fig. 1. The pipeline (namely, NF-destructed P4 pipeline) decomposes the match-action tables of NFs into match and rewrite tables for distinct header fields. Ultimately, different processing rules can be realized by combining these field-grained tables, replacing the sequential execution of NFs in an SFC with an equivalent match-rewrite rule at the SFC-level.

To realize the NF-destructed P4 pipeline, we divide the pipeline's main body into sections specifically for matching and rewriting purposes. The matching section uses two tables for ternary and exact matching of the 5-tuple, respectively. The ternary matching is realized by using the ternary content-addressable memory (TCAM) on the PDP switch, to support masked matching for adapting to various matching patterns. The exact matching is accomplished with the static random-access memory (SRAM) on the PDP switch, and specifically, we design the NF-destructed P4 pipeline to use SRAM to accomplish exact 5-tuple matching for certain types of NFs that need to differentiate flows (*e.g.*, the stateful load-balancer). A successful match in the matching section assigns a unique index to a packet, and then each table in the rewriting section matches against the index, where the related packet fields are rewritten according to the table entry upon a successful match.

Meanwhile, in the control plane of the PDP switch, P4 runtime is used to modify the entries in each table, providing the flexibility of adjusting the rules (*i.e.*, matching and rewriting behaviors) in the pipeline. Note that, these rules are not restricted to a single NF but can also be at the SFC-level.

D. Runtime NF Synthesis Method

In this subsection, we explain our design of converting an SFC deployed on the server into an equivalent set of match-rewrite rules to adapt to the NF-destructed P4 pipeline.

Previously, a number of studies have tackled how to synthesize an SFC by leveraging the directed acyclic graph (DAG). Specifically, they first traverse all the NFs in an SFC, abstracting the entire packet process as a DAG, and then iteratively derive a set of equivalent SFC-level processing rules based on the DAG. However, the time needed to build a DAG and perform synthesis can be relatively long, especially when there are many NFs to process (*e.g.*, for the traffic classifier that needs to distinguish $4,000$ traffic classes, the synthesis time can be as long as 10 seconds [36]). Moreover, the DAG-based approach can hardly convert stateful NFs into the processing rules that can be accommodated in a P4 pipeline. This is because certain stateful NFs, like the stateful load-balancer, need to rewrite the destination IP address of every packet, resulting in each flow matching to a specific processing rule. Then, to minimize the size of a DAG, we have to consolidate the rules by representing their matches with a range that can cover a variety of potential flows. However, this type of rules can hardly be executed on a P4 pipeline, which needs precise definition of each entry of a match-action table.

To address this issue, we design a runtime synthesis method whose procedure is depicted in Fig. 2. When a packet enters an SFC on the server, the ingress NF of the SFC initializes

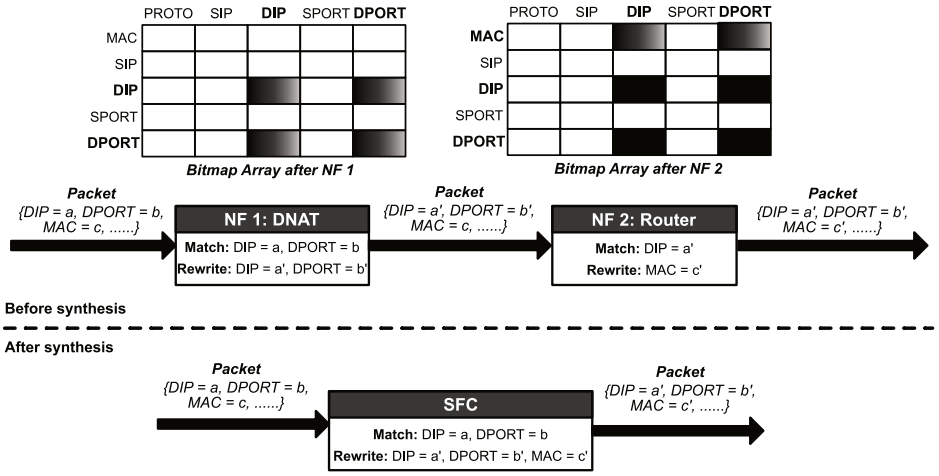

Fig. 2. Procedure of our runtime NF synthesis method.

the metadata for the packet in the shared memory to enable the subsequent synthesis process. The metadata contains the original 5-tuple of the packet and a bitmap array that tracks all the matching and rewriting operations that the packet will experience. Each field of interest (A), such as the source IP address, is assigned a bitmap, where each bit in it corresponds to a specific field (B), and we set the bit as 1 if B can impact the rewriting of A and 0 otherwise. After completing the required match-rewrite operations on a packet, an NF updates the bitmap array. Specifically, the bitmap that corresponds to each field rewritten by the NF is combined with the bitmaps of all the fields involved in the NF by using the logical OR. If a field's bitmap has an initial value of 0's, indicating that it has not yet been used for matching in this SFC, the bit corresponds to the field itself in its bitmap has to be set to 1 before we combining it with the bitmaps of other fields.

After the packet exits the SFC, a bitmap array is obtained, which has been iteratively updated by each NF in the SFC. This bitmap array indicates which field has been rewritten and identifies the fields that are responsible for the rewriting, essentially mapping the original 5-tuple to the processed 5-tuple after executing the SFC. Then, a match-rewrite rule at the SFC-level can be obtained, which is equivalent to all the match-rewrite rules encountered by the packet at the SFC's NFs. This can be accomplished by utilizing the bitmap array, the initial 5-tuple, and the processed 5-tuple of the packet.

Considering the fact that most NFs have fixed match-rewrite patterns, we further optimize the NF synthesis method. Specifically, we make each NF register its match-rewrite pattern to the NF Manager to get a unique ID for the pattern. For instance, in the example shown in Fig. 2, *NFs* 1 and 2 (*i.e.*, for DNAT and router, respectively) have fixed match-rewrite patterns as {(DIP, DPORT), (DIP, DPORT)} and {(DIP), (MAC)}, respectively. The synthesized match-rewrite rule at the SFC-level for *NFs* 1 and 2 is {(DIP, DPORT), (MAC, DIP, DPORT)}, corresponding to the two unique IDs assigned to *NFs* 1 and 2. Then, when a packet of the two NFs arrives, we store the IDs in the metadata for the packet. Hence, the SFC-level match-rewrite patterns can be retrieved with the IDs, avoiding the need of updating the bitmap array for each packet going through an NF of the SFC.

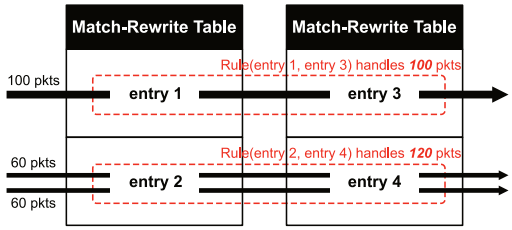

Fig. 3. Example on the mismatch between heaviest flow and hottest rule.

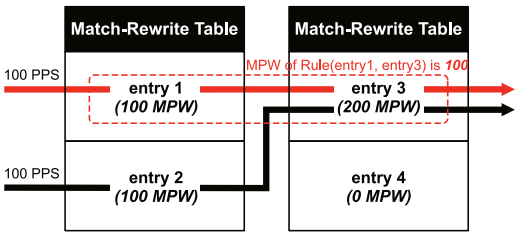

Fig. 4. Count-min selection strategy with 1-second *EWindow* length.

E. Count-Min Selection Strategy

This subsection describes the strategy by which SFCache determines whether the match-rewrite rules at the SFC-level should be cached on the PDP switch.

At first glance, it appears possible to leveraging existing methods such as the heavy-hitter detection [26, 27] to determine which of the match-rewrite rules should be cached on the PDP switch. However, we hope to point out that heavy-hitter detection usually operates on the per-flow basis and can only find the flows whose packet counts are the highest. As we opt to cache packet processing rules, a flow with a high packet count does not necessarily mean that its processing rules are the hottest. Specifically, as shown in Fig. 3, it is possible for multiple small flows to collectively make one or more rules the hottest. In such cases, heavy-hitter detection may not be a suitable solution to address the problem.

Hence, we propose to evaluate the cache-worthiness of an SFC-level match-rewrite rule based on the usage frequencies

Algorithm 1: Count-min Selection Strategy

State: wid: index of current *EWindow*, $te.mpw$: MPW of table entry te, $te.wid$: index of the most recent *EWindow* when $te.mpw$ was updated.

```
1  while a packet p arrives do
2      store wid in p.wid and set p.mpw = UINT32_MAX;
3      for each table entry te of SFC of p in sequence do
4          if p.wid > te.wid then
5              if p.wid = te.wid + 1 then
6                  p.mpw = min(p.mpw, te.mpw);
7              else
8                  p.mpw = 1;
9              end
10             te.wid = p.wid, te.mpw = 1;
11         else
12             p.mpw = 0, te.mpw = te.mpw + 1;
13         end
14     end
15     if (p.mpw ∈ (M_th, UINT32_MAX)) then
16         synthesize table entries experienced by p to get
           an SFC-level match-rewrite rule r;
17         generate a CacheREQ to cache r in PDP switch;
18     end
19 end
```

of the table entries that are related to it. To achieve this, we first consider fixed-duration time slots, each of which serves as an *evaluation window* (*EWindow*). Then, we leverage the well-known count-min selection strategy to find the smallest "matches per *EWindow*" (MPW) among all the match-rewrite table entries that a packet encounters in an SFC, as shown in Fig. 4. The rationale behind choosing the count-min selection strategy is twofold. First, as an SFC-level match-rewrite rule is synthesized with related match-rewrite table entries, using the minimum of the entries' MPWs as the MPW of the SFC-level match-rewrite rule can reduce the overestimation errors that arise from multiple flows referring to the same table entries. Second, the low complexity of the count-min selection strategy ensures the packet processing performance of SFCache.

In each *EWindow*, the MPW of a table entry accumulates, which makes the MPWs of different *EWindows* independent, ensuring their timeliness. When an *EWindow* ends, the MPWs of table entries are collected by the control plane to determine the rule caching scheme. In general, a longer *EWindow* implies a more conservative way of choosing hot rules, and *vice versa*.

Algorithm 1 explains the count-min selection strategy, which obtains the MPW of an SFC-level match-rewrite rule r and decides whether to cache it on the PDP switch based on the MPW. Here, for each table entry te, we assign two variables: its current MPW $te.mpw$, and the index $te.wid$ to record the index of the most recent *EWindow* when $te.mpw$ was updated. When a packet p arrives, we update the variables of all the table entries that it is processed by in its SFC, and finally get the MPW of the packet's SFC-level match-rewrite rule r, using the while-loop of *Lines* 1-19.

Line 2 initializes the variables associated with packet p. Here, $p.wid$ stores the index of the *EWindow* that p experiences, and it will be compared with the $te.wid$ of a table entry te to ascertain the timeliness of $te.mpw$. We use $p.mpw$ to denote the estimated MPW of the packet's SFC-level match-rewrite rule, which is initialized as a relatively large value to facilitate subsequent updates. Then, when packet p is processed by each table entry te of its SFC in sequence, we update $te.wid$, $te.mpw$, and $p.mpw$ according to the difference between $p.wid$ and $te.wid$, which implies the timeliness of $te.mpw$ (*Lines* 3-14). Specifically, if p is the first packet processed by te in the current *EWindow* and te has processed packet(s) in the previous *EWindow*, we update $p.mpw$ as the minimum of $p.mpw$ and $te.mpw$ (*Lines* 5-6). Otherwise, if te has not processed packet(s) in the previous *EWindow* (*i.e.*, the table entry has been idle for more than one *EWindows*), we set $p.mpw = 1$ (*Line* 8). On the other hand, if p is not the first packet processed by te in the current *EWindow*, we set $p.mpw = 0$ and increment $te.mpw$ by 1 (*Line* 12). Finally, after packet p has been processed by all the table entries in its SFC, we compare $p.mpw$ with a preset threshold M_{th} to determine whether the SFC-level match-rewrite rule r should be cached on the PDP switch or not (*Lines* 15-18).

The complexity of *Algorithm* 1 is $O(N \cdot M)$, where N is the maximum number of table entries of the SFC of a packet and M is the number of packets that need to be processed. As N can be treat as a constant in SFCache, the actual complexity of *Algorithm* 1 is $O(M)$. Meanwhile, in the worst case, each incoming packet can lead to a *CacheREQ*.

IV. Implementation of SFCache

In this section, we describe the implementation of SFCache, including the P4 pipeline and P4 runtime agent running on the PDP switch and the software SFC platform on the server.

A. P4 Pipeline on PDP Switch

We implement the NF-destructed P4 pipeline of SFCache on a PDP switch based on Intel Tofino ASIC, which consists of the following components (as shown in Fig. 5).

1) Ingress Pipeline: After experiencing the ingress parser, each packet moves on to the matching section. The packet first seeks a match in the exact match table and then proceeds to the ternary match table. A hit in either table grants the packet a unique index and specifies the egress port for it. If no match is found in both tables, the packet is designated to the server. Following the matching section, the packet drop module decides whether to drop the packet based on its egress port, and this is needed for certain types of NFs such as firewalls. Next, the egress redirection module checks whether the packet is from the server. If yes, the packet is redirected to the external network. This step is essential when the packet has to undergo NFs running on the server, because in this case, the egress port specified in the matching section is connected to the server, even when the packet just left the server. Specifically, the redirection is achieved by the bypass table, which can be configured to bypass packets from the egress pipeline (*i.e.*, sending them directly to the server or to external network). The `EtherType` rewrite module modifies

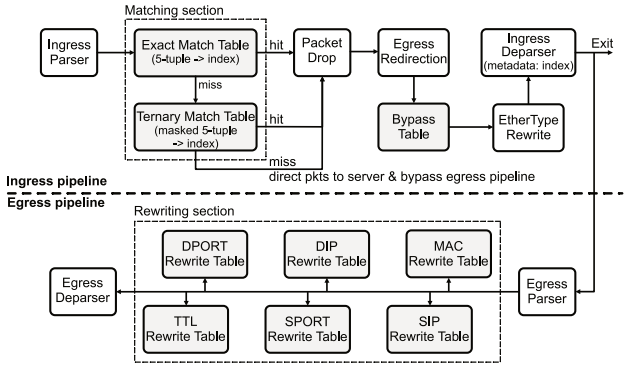

Fig. 5. Implementation of NF-destructed P4 pipeline of SFCache.

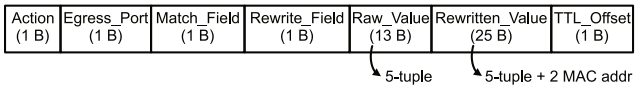

Fig. 6. Packet format of *CacheREQ* message.

the `EtherType` field of a packet if necessary. For the packets whose rules have not been cached in the PDP switch, it changes their `EtherType` fields to $0x2023$ (*i.e.*, signaling SFCache that each of the packets should be evaluated to see whether its processing rule needs to be cached in the PDP switch). Otherwise, when it sees a packet with `EtherType` as $0x2023$, it will restore the `EtherType` to $0x0800$.

2) Egress Pipeline: The egress pipeline is where packet header is modified. The rewriting section includes several match-action tables that use the packet's index obtained from the matching section to rewrite certain fields in its header (*e.g.*, the 5-tuple and the time-to-live (TTL) field). Finally, the egress deparser updates the checksum of the packet.

3) Resource Usage: In the PDP switch, the Tofino ASIC has 4 independent pipelines, each of which contains 12 match action units (MAUs) that each has 10 Mbits of SRAM and 528 Ktrits of TCAM. With $15,000$, $5,000$, and $20,000$ entries in the exact match table, ternary match table, and other tables, respectively, the pipeline implementation of SFCache uses 7 MAU stages, 13.5% of SRAM, and 10.4% of TCAM.

B. P4 Runtime Agent

In the control plane, we develop a P4 runtime agent, which is a multi-threaded program based on C++ and runs on the CPU of the PDP switch. The agent handles *CacheREQs* from the server, extracts SFC-level rules from them to install in the PDP switch, and removes outdated rules when necessary.

Specifically, the agent consists of three threads, *i.e.*, the socket thread, the table thread, and the aging thread. The socket thread parses each *CacheREQ* with the packet format in Fig. 6. The `Action` field denotes the rule cache action encoded in the *CacheREQ*, which can be either caching a new rule (`ADD`) or removing existing rules (`CLEAR`). Then, the `Egress_Port` field specifies the egress port of the packets associated with the corresponding SFC-level rule. If the packets will only be processed by the PDP switch, this field port stores their egress port to the external network. Otherwise, if the packets still require processing on the server, their egress port is the one that connects to the server. Next, `Match_Field` and `Rewrite_Field` are two bitmaps, each bit in which indicates whether a header field is used for matching and rewriting in this cached rule, respectively.

The table thread translates the parsed fields into P4 table entries. Specifically, it first checks `Match_Field` and `Raw_Value` in the *CacheREQ* to identify the fields and values for matching, and then acquires an available ID for this match-rewrite rule from a thread-safe ID manager. Following this, it uses `Rewrite_Field` and `Rewritten_Value` to determine the fields to rewrite and their new values, finalizing the table entries in the rewriting section accordingly.

The aging thread employs the table entry aging mechanism of Tofino ASIC to clear inactive processing rules via a callback function. Each processing rule in SFCache correlates with multiple table entries. Specifically, each rule ID maps to an entry in the matching section and may be related to multiple entries in the rewriting section. We use the ID as a clue for when a processing rule should be deleted. The ID manager keeps track of all the table entries linked to a particular ID. When the callback function begins to process aging events from the pipeline, it consults the ID manager using the ID tied to the event to identify which table entries should be removed.

C. Software SFC Platform on Server

In SFCache, the software SFC platform on the server is an extension of OpenNetVM [18], to enable SFCache features. With our modifications, the SFC platform offers a comprehensive API suite within the DPDK framework, enabling SPs to create NFs with SFC-level rule caching capabilities. The APIs support NF match-rewrite pattern registration, as well as the generation and dispatch of *CacheREQs*. We also introduce an extended table lookup API to collect MPW statistics. We allocate a dedicated core to handle *CacheREQs* solely.

V. Experimental Demonstrations and Evaluations

In this section, we evaluate our SFCache experimentally on a real-world network testbed for the following purposes:

- Assessing the overheads imposed by the runtime NF synthesis method and the count-min selection strategy on the commodity server (Section V-A).
- Measuring the time needed for cached rules to become operational on the PDP switch (Section V-B).
- Examining the performance enhancements that SFCache achieves for various SFC types, with a particular focus on packet processing throughput and latency (Section V-C).
- Evaluating the service disruption during switching the processing of SFCs within the SFCache (Section V-D).
- Analyzing the effect of the length of *EWindow* on the rule caching in the PDP switch and the overall packet processing performance of SFCache, and assessing the scalability of SFCache roughly (Sections V-E and V-F).
- Comparing SFCache to P4SFC [38], which is the existing approach that also combines PDP switches and servers for generic and flexible SFC deployment (Section V-G).

The testbed for experimental evaluation consists of a PDP switch with 40 Gbps ports, which are connected to the server

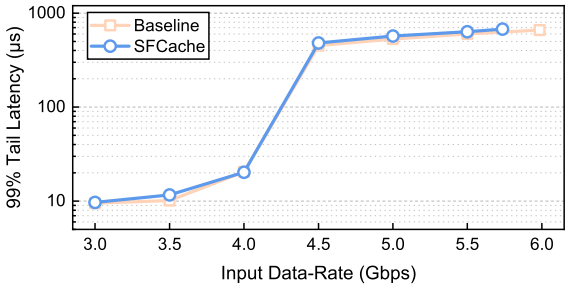

Fig. 7. Per-packet overheads of SFCache in terms of 99% tail latency.

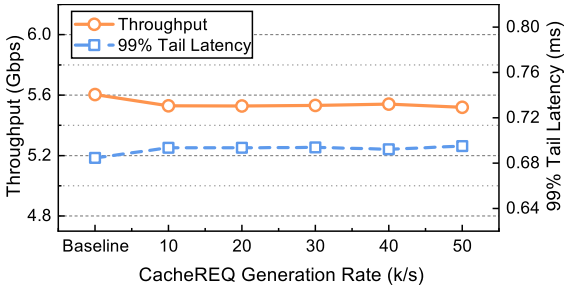

Fig. 8. Overheads of *CacheREQ* generation in SFCache.

and an external packet processing performance test system. The server is equipped with two Intel Xeon Silver 4210 CPUs running at 2.20 GHz with 64 GB of memory. We let SFCache use the CPU cores of a single non-uniform memory access (NUMA) node in the server exclusively, and reserve 16 hugepages, each of which is 1 GB, there for DPDK. The performance testing system consists of a PDP switch for time-stamping packets, a hardware traffic generator, a server that runs pktgen-dpdk [42]. The hardware traffic generator and the server running pktgen-dpdk are used to replay traffic traces at 40 Gbps, which include both synthetic and real-world traces. Specifically, the real-world trace is taken from the WIDE project [43], and it contains the statistics of 2.97 million flows.

A. Overheads Introduced by SFCache

To implement SFCache, we develop additional features on OpenNetVM, which introduces computational overheads. We first design an experiment to assess the overheads on a per-packet basis, where we deploy an SFC consisting of three NFs on the server, each of which is allocated a CPU core. The experiment sends 64-byte packets according to a synthetic trace to the SFC, and measures packet processing latency in terms of the 99% tail latency. We use the original OpenNetVM [18] as the baseline. The experimental results indicate that SFCache results in 4.16% drop in packet processing throughput. Fig. 7 shows the results on latency, suggesting that compared with the baseline, SFCache only induces an average increase of 5.44% in 99% tail latency across various input rates.

According to our design, when an SFC-level processing rule becomes sufficiently hot, SFCache will generate a *CacheREQ* to the PDP switch, which brings in overheads. Therefore, we design another experiment to measure the overheads, by changing the rate of *CacheREQ* generation on the server. Note that, to emulate the worst case scenario, the experiment does not let the P4 runtime agent handle *CacheREQs* or offload any SFC-level rules to the PDP switch, *i.e.*, all the traffic (64-byte packets) is still processed by the SFCs on the server by the same CPU core that is used to generate *CacheREQs*.

The experimental results are plotted in Fig. 8. We can see that compared with the baseline (*i.e.*, the case in which the feature of *CacheREQ* generation is removed from the server part of SFCache), there is no significant decrease in throughput when *CacheREQ* generation rate increases, and among all the scenarios, the maximum throughput reduction is only 1.52%. On the other hand, the results on 99% tail latency in Fig. 8 indicate that the maximum increase on latency is only 1.54%, when comparing the cases with *CacheREQ* generation with the baseline. Meanwhile, we also observe that the results on throughput and latency generally do not change when the *CacheREQ* generation rate increases. This is due to the low complexity of the procedure of processing packets to generate *CacheREQs* and the fact that the *CacheREQ* generation rates in Fig. 8 are the way below the threshold above which the generation of *CacheREQs* can impact the performance of the server. Specifically, the maximum *CacheREQ* generation rate of 50,000 per second in Fig. 8 is chosen according to the intrinsic table entry insertion capacity of the PDP switch.

TABLE II
LATENCY FOR CACHED RULES TO BECOME OPERATIONAL

Minimum (msec)	Maximum (msec)	Average (msec)
0.70	1.09	0.85

B. Latency for Cached Rules to become Operational

After the server generates a *CacheREQ* to cache an SFC-level rule on the PDP switch, it takes certain time for the rule to become operational. In order to measure this latency, we conduct 20 independent tests and the results are listed in Table II. It can be seen that the average latency is only 0.85 msec. Note that, the rules cached in SFCache are primarily for elephant flows, which are typically long-lasting and bandwidth-intensive, a latency at the sub-millisecond level will be acceptable and will not affect their performance noticeably.

C. Performance of SFC Deployment

In the following, we discuss two experiments to evaluate the performance of SFCache on SFC deployment.

1) Stateless SFC with Restricted CPU Core Usage: This experiment utilizes SFCache to deploy a stateless SFC that incorporates a firewall followed by a DPI. According to the operation principle of SFCache, the DPI can only be carried by the software SFC platform on the server, while the firewall is stateless and thus its match-action rules can be fully cached on the PDP switch. Note that, when the stateless SFC runs completely on the server, its two NFs should be assigned with dedicated CPU cores to avoid frequent CPU context switching and cache pollution. However, after the stateless firewall has been offloaded to the PDP switch, the cores assigned to it will become idle, making the dedicated core assignment inefficient.

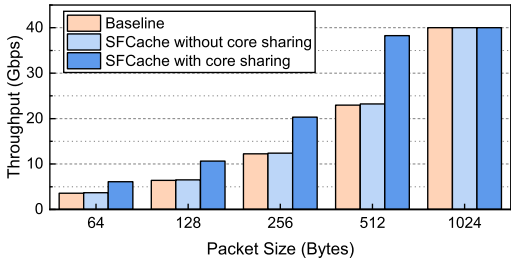

Fig. 9. Results on throughput of experimental scenario with stateless SFC.

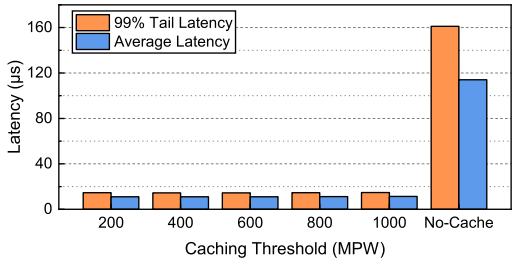

Fig. 10. Results on latency of experimental scenario with stateful SFC.

TABLE III
LATENCIES FROM EXPERIMENTAL SCENARIO WITH STATELESS SFC

	99% Tail Latency (μs)	Average Latency (μs)
Baseline	162.76	137.03
SFCache without core sharing	163.73	128.77
SFCache with core sharing	167.45	70.01

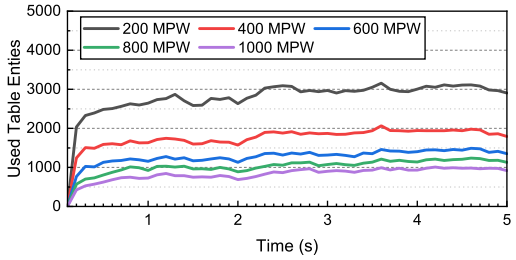

Fig. 11. Changes of table entry usage on PDP switch over time.

Therefore, the experiment considers three scenarios, which are the original OpenNetVM (the baseline), and the SFCache implementations that enable and disable CPU core sharing in the server. All the scenarios are configured to utilize 5 CPU cores for the two NFs, but the way of allocating the CPU cores is different per scenario. Specifically, for the baseline and SFCache without core sharing, we allocate 3 and 2 cores to the DPI and stateless firewall, respectively, while the SFCache with core sharing assigns 3 cores to the DPI and makes it share the remaining 2 cores with the stateless firewall.

Fig. 9 compares the throughput results of the three scenarios, which shows that the SFCache without core sharing provides similar throughput as that of the baseline. This is because even though SFCache can offload the stateless firewall to the PDP switch, the DPI is still heavily loaded with only three cores and thus remains the bottleneck. This issue is alleviated by SFCache with core sharing, leading to significantly higher throughput in Fig. 9 when the packet size is 512 bytes or less. For example, when the packet size is 64 bytes, SFCache with core sharing improves the throughput by 41.32% over the baseline. This verifies the advantage of SFCache on adjusting resource allocation in the software SFC platform adaptively in run-time, for optimizing SFC performance.

As the throughput results in Fig. 9 indicates that all the three scenario can achieve the line-rate of 40 Gbps with the packet size of $1,024$ bytes, we fix the packet size as $1,024$ bytes to measure the packet processing latencies of the scenarios. This not only ensures apple-to-apple comparisons but also tells the smallest performance gaps on latency between SFCache with core sharing and the other two scenarios. The latency results are listed in Table III. We can see that the baseline and SFCache without core sharing perform similarly in terms of both the 99% tail and average values of packet processing latency, but SFCache with core sharing reduces the average latency by 48.91% over the baseline. Meanwhile, it is interesting to notice that the 99% tail latency of SFCache with core sharing is slightly longer (2.88%) than that of the baseline. This can be attributed to the abnormal latencies induced by CPU context switching in some corner cases.

2) Stateful SFC with Long-Tail Distribution Traffic: In this experiment, we consider a stateful SFC that consists of a network address/port translator (NAPT) and a stateful load-balancer, which dynamically generate stateful processing rules in run-time. Therefore, it would be impractical to cache all of their rules on the PDP switch, especially when the input traffic distribution is long-tail. This time, we allocate four CPU cores to each NF in the software SFC deployment platform, and the input traffic is generated according to the real-world trace collected in the WIDE project [43]. Meanwhile, we make sure that the server is capable to process traffic for the NFs at line-rate, and thus latency becomes the only metric of interest.

As for rule caching, we set the length of *EWindow* as 100 msec and gradually increase the threshold M_{th} in *Algorithm* 1 from 200 to $1,000$ MPW, and the no-cache scenario is used as the baseline. Fig. 10 shows the results on latency, which indicate that rule caching on the PDP switch stabilizes the 99% tail and average latencies at 14.56 μs and 11.09 μs, respectively, regardless of the threshold used in SFCache, and it respectively reduces the 99% tail and average latencies by up to 91.03% and 90.37% over the baseline. This is because in SFCache, the NFV acceleration provided by the PDP switch greatly relieves the processing load of the software-only data path on the server. Fig. 11 plots how the number of table entries, which are used for caching SFC-level rules on the PDP switch, changes over experimental time. It can be seen that the table entry usage increases rapidly after the first *EWindow* (*i.e.*, 100 msec in this experiment), regardless of the threshold M_{th}, and a smaller M_{th} leads to a larger table entry usage when the ruling caching has become stable. This confirms the benefit of *Algorithm* 1 for achieving timely and efficient rule caching. Meanwhile, Fig. 11 also suggests that offloading SFC processing rules from commodity servers to the PDP switch is handled automatically by SFCache at runtime, verifying the transparency of SFCache in SFC deployment, *i.e.*, SPs do not

TABLE IV
EFFECT OF EWINDOW LENGTH

Length of *EWindow* (msec)	200	400	600	800	1000
Cache Efficiency (%)	88.52	95.86	98.78	99.35	99.82
Selected Rules (‰)	1.00	0.68	0.54	0.46	0.36
Table Entry Insertion (entries/s)	1854	1083	785	634	532

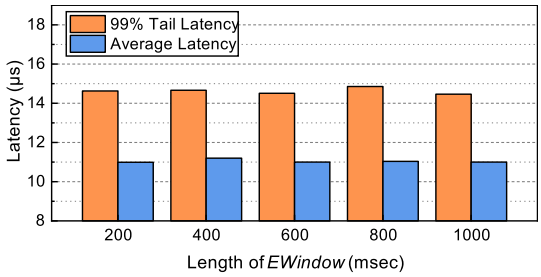

Fig. 12. Impact of *EWindow* length on latency performance of SFCache.

need to worry about the operations related to PDP switches.

D. Switching SFC Processing in SFCache

As one important design goal of SFCache is to support flexible SFC deployment, we allow the existing pipeline in the PDP switch to adapt to different SFCs without reprogramming or reloading. Specifically, the change of SFC processing rules is accomplished by replacing the match-rewrite rules of the old SFCs with those of the new ones. Therefore, to evaluate the service disruption during switching the processing of SFCs within SFCache, we design an experiment with stateless SFCs, and measure the service interruption duration of deploying a new SFC to replace an old one in SFCache. Note that, the old SFC can run on either the server or the PDP switch, we record the longer service interruption between the two cases. Our results indicate that the average service interruption is 3.062 seconds, which is acceptable for most elephant flows.

E. Effect of Evaluation Window Length

As explained in Section III-E, the length of *EWindow* in *Algorithm* 1 determines how aggressive it chooses hot SFC-level match-rewrite rules to cache on the PDP switch. We define a *cache efficiency* as the ratio of the number of flows processed by cached rules to the total times that rules have been selected for being cached by *Algorithm* 1. Note that, if a cached rule has become aged and is then re-cached, we count it being cached twice. Since the rule aging time on the PDP switch is set to the length of an *EWindow*, a shorter *EWindow* means that cached rules will age quickly, reducing the cache efficiency as rules might be re-cached more frequently.

We perform experiments to evaluate the effect of *EWindow* length. This time, we use 8 CPU cores for two NFs, which perform flow table matching operations on the flows that are generated according to the traces in [43] with a constant total throughput of 40 Gbps. Here, the processing rule of each flow consumes 3 table entries in the PDP switch. The preset caching threshold is set as $M_{th} = 1,000 \cdot wlen$ MPWs, where $wlen$ is the length of *EWindow* in seconds.

Table IV shows the effect of *EWindow* length on the cache efficiency, selected rules (*i.e.*, ratio of rules selected for being cached), and insertions of table entries in the PDP switch. We can see that the cache efficiency increases with the length of *EWindow*. This is because a shorter *EWindow* can make *Algorithm* 1 select more rules of short-lived flows. Meanwhile, increasing *EWindow* also reduces the speed of table entry insertion to the PDP switch, which relieves the pressure on the control plane (*i.e.*, the workload of the P4 runtime agent). In order to further confirm the packet processing performance of SFCache when we use a relatively long *EWindow*, we plot the results on per-packet processing latency in Fig. 12, which are obtained by ignoring the latency results in the first second of each experiment (*i.e.*, the transition time before the rule caching has been stable). Fig. 12 indicates that the length of *EWindow* only affects the per-packet processing latency slightly, either in terms of the tail latency or average latency.

F. Scalability of SFCache

When the volume of input traffic increases, the scalability of SFCache might face the challenges from three perspectives: 1) the processing capacity of the software-based data path, 2) the speed of the P4 runtime agent on table entry insertion, and 3) the memory space of the PDP switch for rule caching. As there are numerous techniques in the literature to improve the capacity of software data paths, we will focus on the bottlenecks related to the PDP switch in the following discussions.

We analyze the aforementioned bottlenecks based on the experimental results in the previous subsection, since they are obtained with a real-world traffic trace and thus representative to certain extent. The results in Table IV suggest that when we set the length of *EWindow* as one second and the caching threshold as $M_{th} = 1,000$ MPWs, the average insertion rate of table entries is 532 per second. Meanwhile, in this case, the PDP switch caches an average of 661 processing rules.

According to our measurements, the maximum capacity of processing rules of the SFCache pipeline implementation is $\sim$150,000, and the maximum table entry insertion rate supported by the P4 runtime agent is $\sim$50,000 entries per second. Hence, the experimental results on cached processing rules and table entry insertion rate correspond to 0.44% of the processing rule capacity and 1.06% of the maximum table entry insertion rate, *i.e.*, the latter is the actual bottleneck. Then, if we assume that the input traffic follows the statistics of the used real-world trace, the bottleneck in the PDP switch of SFCache would not occur until the total input traffic volume exceeds $40/1.06\% \approx 3770$ Gbps or 3.77 Tbps. Note that, this is just a very approximate analysis, and the actual scalability of SFCache will be explored in future work.

G. Benchmarking against Existing Approach

Finally, we benchmark SFCache against P4SFC [38], which also aims to realize generic and flexible SFC deployment with commodity servers and PDP switches. The primary difference between SFCache and P4SFC lies in their approaches for

TABLE V
RESOURCE UTILIZATION IN PDP SWITCH PIPELINE

	SFCache	P4SFC
MAU stage	7 of 12 (NF-independent)	12 of 12 (two NF modules)
SRAM	13.5%	27.7%
TCAM	10.4%	0%

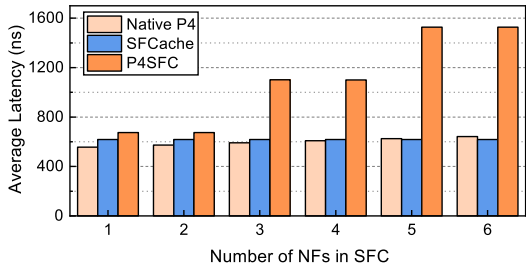

Fig. 13. Average latency that packets encounter in PDP switch pipeline.

achieving versatility in PDP switch pipeline. SFCache uses an NF-destructed P4 pipeline, where the match-rewrite tables are not targeted at individual NFs, but rather at the packet fields. In contrast, the pipeline design of P4SFC focuses on individual NFs by stringing together runtime-configurable "stages", each of which represents a configurable NF module that can act as an NF. Hence, packet recirculation might be needed to reuse certain stages to support a relatively long SFC.

We first perform experiments to compare the resource usages of the two approach, when $20,000$ SFC-level processing rules for two NF modules (*i.e.*, each NF module can support multiple NFs) need to be offloaded to the PDP switch. Note that, the limitation of two NF modules comes from P4SFC, because we find that it can only accommodate two NF modules in the 12 MAU stages of our PDP switch with Tofino ASIC, due to the dependencies between different parts of its pipeline. Table V summarizes the results on the usages of MAU stages, SRAM and TCAM. We can see that P4SFC indeed exhausts all the 12 MAU stages, while SFCache only uses 7 stages and delivers a much more efficient stage usage. Meanwhile, SFCache saves 51.3% SRAM usage over P4SFC, but some of the saving might be attribute to the fact that P4SFC does not use any TCAM while the TCAM usage of SFCache is 10.4%.

Then, we compare the results on the average latency that packets encounter in the pipeline of the PDP switch. Here, as P4SFC can only place two NF modules in the pipeline, it will introduce recirculation if a flow's SFC consists of more than 2 NFs. Fig. 13 shows the latency results. We can see that when the number of NFs increases, the latency of the native P4 pipeline increases slightly, primarily due to the proportional increase in the number of tables that packets need to match to. On the other hand, the latency of SFCache remains unchanged and is even slightly lower than that of the native P4 when the SFC includes 5 or more NFs. This is because our runtime NF synthesis method merges all the cacheable NF rules in an SFC, and thus the number of tables that a packet needs to traverse in SFCache pipeline only depends on the packet fields of interest to the SFC, not the number of NFs. Finally, due to the need of recirculation, the latency of P4SFC increases dramatically when there are 3 or more NFs in the SFC.

H. Discussions and Future Work

1) Adaptive Cached Rule Replacement in the PDP Switch: Currently, we design the cached rule replacement scheme in the PDP switch based on Tofino ASIC's built-in table aging mechanism, which might not be adaptive enough, especially when the traffic is highly dynamic or/and the memory resources are limited on the PDP switch. In our future work, we will try to design more adaptive cached rule replacement scheme to optimize memory allocation more effectively.

2) Static versus *Dynamic Stateful NFs:* Stateful NFs can be categorized into static and dynamic types based on the frequency of their state updates. Static stateful NFs, such as NAPT and load-balancers, only alter their states upon new flow arrivals. In contrast, dynamic stateful NFs, like flow monitors or stateful firewalls, continuously update their states. Within the context of SFCache, caching on the PDP switch is currently limited to the rules associated with static stateful NFs, while dynamic stateful NFs can only be implemented on the server. This stems from SFCache's primary design goal of providing generic SFC deployment support, which leads to the decision of not pursuing the optimizations for specific NFs on a PDP switch. As a result, offloading dynamic stateful NFs in such a generic design would require using many registers to store and frequently update per-flow states, but this can hardly be achieved with current PDP switches.

3) Cache-Hiding Problem: Offloading SFC-level processing rules from the server to the PDP switch can lead to the cache-hiding problem [44, 45], which happens occurs when lower-priority rules are unintentionally given precedence due to the offloading, negating the intended priority of the rules on the server. This problem is particularly noticeable when the matching space of processing rules is overlapped. Hence, the cache-hiding problem can also be an issue in our SFCache. Fortunately, there have already been studies on this problem [44, 45], and their solutions can be leveraged by SFCache.

VI. CONCLUSION

In this paper, we presented SFCache, a flexible SFC deployment system that combines the superior packet processing performance of PDP switches with the software flexibility of commodity servers. By offloading SFC-level processing rules from the server to the PDP switch, SFCache allows the related traffic to benefit from the performance gain offered by the PDP switch. Meanwhile, the rule caching on the PDP switch relieves the processing load on the server, allowing all the SFCs running in SFCache to perform well. Moreover, SFCache supports flexible SFC deployment, meaning that deploying a new SFC does not require reprogramming and reloading the packet processing pipeline in the PDP switch. We prototyped SFCache with a PDP switch based on Tofino ASIC and a commodity server, and demonstrated its performance experimentally. The experimental results obtained with real-world testbed indicated that 1) for a stateless SFC, SFCache increased its throughput by up to 41.32% and reduced its

average latency by 48.91%, and 2) for a stateful SFC, SFCache decreased the 99% tail and average values of its latency by up to 91.03% and 90.37%, respectively.

ACKNOWLEDGMENTS

This work was supported by the NSFC project 62371432.

REFERENCES

[1] P. Lu *et al.*, "Highly-efficient data migration and backup for Big Data applications in elastic optical inter-datacenter networks," *IEEE Netw.*, vol. 29, pp. 36–42, Sept./Oct. 2015.

[2] L. Gong and Z. Zhu, "Virtual optical network embedding (VONE) over elastic optical networks," *J. Lightw. Technol.*, vol. 32, pp. 450–460, Feb. 2014.

[3] Z. Zhu, W. Lu, L. Zhang, and N. Ansari, "Dynamic service provisioning in elastic optical networks with hybrid single-/multi-path routing," *J. Lightw. Technol.*, vol. 31, pp. 15–22, Jan. 2013.

[4] L. Gong *et al.*, "Efficient resource allocation for all-optical multicasting over spectrum-sliced elastic optical networks," *J. Opt. Commun. Netw.*, vol. 5, pp. 836–847, Aug. 2013.

[5] J. Liu *et al.*, "On dynamic service function chain deployment and readjustment," *IEEE Trans. Netw. Serv. Manag.*, vol. 14, pp. 543–553, Sept. 2017.

[6] Z. Zhu *et al.*, "Impairment- and splitting-aware cloud-ready multicast provisioning in elastic optical networks," *IEEE/ACM Trans. Netw.*, vol. 25, pp. 1220–1234, Apr. 2017.

[7] J. Sherry *et al.*, "Making middleboxes someone else's problem: Network processing as a cloud service," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, pp. 13–24, Oct. 2012.

[8] T. Barbette, C. Soldani, and L. Mathy, "Combined stateful classification and session splicing for high-speed NFV service chaining," *IEEE/ACM Trans. Netw.*, vol. 29, pp. 2560–2573, Dec. 2021.

[9] A. Bremler-Barr, Y. Harchol, and D. Hay, "OpenBox: A software-defined framework for developing, deploying, and managing network functions," in *Proc. of ACM SIGCOMM 2016*, pp. 511–524, Aug. 2016.

[10] M. Chiosi *et al.*, "Network functions virtualisation," 2012. [Online]. Available: https://portal.etsi.org/nfv/nfv_white_paper.pdf.

[11] R. Gandhi *et al.*, "Duet: Cloud scale load balancing with hardware and software," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 27–38, Oct. 2014.

[12] B. Li *et al.*, "Clicknp: Highly flexible and high performance network processing with reconfigurable hardware," in *Proc. of ACM SIGCOMM 2016*, pp. 1–14, Aug. 2016.

[13] J. Cao *et al.*, "CoFilter: High-performance switch-accelerated stateful packet filter for bare-metal servers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, pp. 2249–2262, Dec. 2021.

[14] X. Fei *et al.*, "Paving the way for NFV acceleration: A taxonomy, survey and future directions," *ACM Comput. Surv.*, vol. 53, pp. 1–42, Jan. 2020.

[15] D. Eisenbud *et al.*, "Maglev: A fast and reliable software network load balancer," in *Proc. of USENIX NSDI 2016*, pp. 523–535, Mar. 2016.

[16] P. Patel *et al.*, "Ananta: Cloud scale load balancing," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, pp. 207–218, Oct. 2013.

[17] S. Palkar *et al.*, "E2: A framework for NFV applications," in *Proc. of ACM SOSP 2015*, pp. 121–136, Oct. 2015.

[18] W. Zhang *et al.*, "OpenNetVM: A platform for high performance network service chains," in *Proc. of ACM HotMIddlebox 2016*, pp. 26–31, Aug. 2016.

[19] H. Li *et al.*, "LemonNFV: Consolidating heterogeneous network functions at line speed," in *Proc. of USENIX NSDI 2023*, pp. 1451–1468, Apr. 2023.

[20] C. Sun *et al.*, "NFP: Enabling network function parallelism in NFV," in *Proc. of ACM SIGCOMM 2017*, pp. 43–56, Aug. 2017.

[21] Data Plane Development Kit (DPDK) of Intel. [Online]. Available: https://www.dpdk.org.

[22] L. Rizzo, "netmap: a novel framework for fast packet I/O," in *Proc. of USENIX Security 2012*, pp. 101–112, Aug. 2012.

[23] P4 Specification. [Online]. Available: https://github.com/p4lang/p4-spec.

[24] Intel Tofino Series. [Online]. Available: https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino.html.

[25] Z. Liu *et al.*, "Jaqen: A High-Performance Switch-Native approach for detecting and mitigating volumetric DDoS attacks with programmable switches," in *Proc. of USENIX Security 2021*, pp. 3829–3846, Aug. 2021.

[26] V. Sivaraman *et al.*, "Heavy-hitter detection entirely in the data plane," in *Proc. of ACM SOSR 2021*, pp. 164–176, Apr. 2017.

[27] D. Ding, M. Savi, G. Antichi, and D. Siracusa, "An incrementally-deployable P4-enabled architecture for network-wide heavy-hitter detection," *IEEE Trans. Netw. Service Manage.*, vol. 17, pp. 75–88, Jan. 2020.

[28] R. Miao *et al.*, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching ASICs," in *Proc. of ACM SIGCOMM 2017*, pp. 15–28, Aug. 2017.

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

[30] X. Chen *et al.*, "P4SC: Towards high-performance service function chain implementation on the P4-capable device," in *Proc. of IFIP/IEEE IM 2019*, pp. 1–9, Apr. 2019.

[31] D. Wu *et al.*, "Accelerated service chaining on a single switch ASIC," in *Proc. of ACM HotNets 2019*, pp. 141–149, Nov. 2019.

[32] D. Kim *et al.*, "TEA: Enabling state-intensive network functions on programmable switches," in *Proc. of ACM SIGCOMM 2020*, pp. 90–106, Jul. 2020.

[33] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless network functions: Breaking the tight coupling of state and processing," in *Proc. of USENIX NSDI 2017*, pp. 97–112, Mar. 2017.

[34] M. Scazzariello *et al.*, "A high-speed stateful packet processing approach for Tbps programmable switches," in *Proc. of USENIX NSDI 2023*, pp. 1237–1255, Apr. 2023.

[35] S. Goswami *et al.*, "Parking packet payload with P4," in *Proc. of ACM CoNEXT 2020*, pp. 274–281, Nov. 2020.

[36] G. Katsikas *et al.*, "Metron: High-performance NFV service chaining even in the presence of blackboxes," *ACM Trans. Comput. Syst.*, vol. 38, pp. 1–45, Jul. 2021.

[37] X. Chen *et al.*, "LightNF: Simplifying network function offloading in programmable networks," in *Proc. of IWQOS 2021*, pp. 1–10, Jun. 2021.

[38] J. Ma, S. Xie, and J. Zhao, "Flexible offloading of service function chains to programmable switches," *IEEE/ACM Trans. Serv. Comput.*, vol. 16, pp. 1198–1211, Mar. 2022.

[39] G. Li *et al.*, "Enabling performant, flexible and cost-efficient DDoS defense with programmable switches," *IEEE/ACM Trans. Netw.*, vol. 29, pp. 1509–1526, Aug. 2021.

[40] C. Zhang *et al.*, "Hyperv: A high performance hypervisor for virtualization of the programmable data plane," in *Proc. of ICCCN 2017*, pp. 1–9, Jul. 2017.

[41] D. Hancock and J. Merwe, "Hyper4: Using P4 to virtualize the programmable data plane," in *Proc. of ACM CoNEXT 2016*, pp. 35–49, Dec. 2016.

[42] pktgen-dpdk - Traffic generator powered by DPDK. [Online]. Available: https://git.dpdk.org/apps/pktgen-dpdk/.

[43] MAWI Working Group Traffic Archive (wide.ad.jp). [Online]. Available: https://mawi.wide.ad.jp/mawi/samplepoint-F/2023/202307041400.html.

[44] Y. Liu, S. Amin, and L. Wang, "Efficient FIB caching using minimal non-overlapping prefixes," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, pp. 14–21, Jan. 2013.

[45] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Cacheflow: Dependency-aware rule-caching for software-defined networks," in *Proc. of ACM SOSR 2016*, pp. 1–12, Mar. 2016.