P4SFC: Service Function Chain Offloading with Programmable Switches

Junte Ma*†, Sihao Xie*†, Jin Zhao*†
*School of Computer Science, Fudan University, China
†Shanghai Key Laboratory of Intelligent Information Processing, Shanghai, China
{jtma19, xiesh19, jzhao}@fudan.edu.cn

***Abstract*—A Service Function Chain (SFC) is an ordered sequence of network functions (NFs). Software-based NFs in Network Function Virtualization (NFV) could introduce significant performance overhead. In this paper, we present P4SFC, a high-performance SFC system that leverages P4-capable switches to accelerate packet processing by offloading proper NFs to the switches. First, considering the current limitations of P4, we analyze the offloadability of NFs and divide them into three categories: fully offloadable, partially offloadable, and non-offloadable. Second, when deploying new SFCs, P4SFC automatically offloads proper NFs to switches based on their position and offloadability. To deploy new SFCs at runtime, we design a dynamic P4 data plane, whose execution logic can be reconfigured at runtime without interrupting the current execution logic. Finally, to maintain state consistency between the server and the switch for partially offloaded NFs, we design a state library to automatically synchronize states between servers and switches. Experimental results show that P4SFC achieves significant performance improvement for real-world SFCs.**

***Index Terms*—Network Function Virtualization, Service Function Chain, P4, High Performance.**

I. Introduction

Network flows are often required to be processed by multiple network functions (NFs) in an ordered sequence. Moreover, different tenants have different requirements in terms of the chaining sequence of NFs. This notion is known as Service Function Chaining (SFC) [1]. Network Functions Virtualization (NFV) addresses the problems of traditional proprietary middleboxes by leveraging virtualization technologies to implement NFs on commodity servers [2]. However, the benefits of NFV come with considerable compromises especially on packet processing latency. For example, Ananta Software Muxes running on commodity servers can increase the latency from 200µs to 1ms at 100 Kpps [3]. Such high latency may be unacceptable for latency-sensitive network applications, such as algorithmic stock trading. To overcome the limitations of software packet processing while retaining flexibility, some recent efforts are proposed to accelerate SFCs by offloading (a part of) NF operations to high-performance hardware, including Graphics Processing Units (GPUs) [4], Network Interface Cards (NICs) [5], and Field Programmable Gate Arrays (FPGAs) [6].

Emerging programmable data plane and data plane programming languages such as P4 [7] offer new opportunities to accelerate SFCs. There have been pioneering efforts in offloading SFCs into P4-enabled switches [8], [9]. However, existing offloading strategies simply try to offload entire SFCs into the switch. Unfortunately, due to the current limitations of P4, some NFs cannot be implemented in P4. For example, the Intrusion Prevention System (IPS) that needs to access packet payload is not supported by P4 currently. Thus, the above systems can only deploy fully offloadable NFs.

In this paper, we present P4SFC, a high-performance SFC system which leverages P4-capable switches to accelerate SFCs by intelligently offloading appropriate NFs in an SFC into switches (Hereinafter, all switches are P4-capable unless otherwise specified.). We adopt the popular modular-based method [10] to construct NFs in our system. Considering the current limitations of P4, we divide NFs into three categories in terms of their offloadability, i.e., fully offloadable, partially offloadable, and non-offloadable. Through the above classification, when deploying an SFC, we can offload some NFs in an SFC from commodity servers into high-performance switches to accelerate packet processing. Since SFC requests can come at any time, we elaborately design a P4 program whose execution logic can be reconfigured at runtime to deploy or destroy SFCs. Since the states generated and maintained by each NF are essential for its functionality, we specially design a state library that can automatically maintain NF state consistency between servers and switches.

Our main contributions are:

- We propose a high-performance SFC system named P4SFC, which offloads proper NFs to P4-capable devices to accelerate SFCs.
- We analyze the offloadability of NFs by considering the limitations of P4. In particular, we design a state library for partially offloadable NFs to automatically maintain state consistency between the switch and the server.
- We design a dynamic P4 data plane in P4SFC to support the reconfiguration of switches at runtime, without interrupting current execution of other SFCs.
- We implement a prototype of P4SFC. Experimental results show that P4SFC gains a latency reduction of around 20%, and a 46.5% processing rate gains in throughput compared with pure Click [10], for realistic SFCs.

II. Background and Design Challenges

A. Background

P4 [7] is a programming language specially designed to program data plane packet processing pipelines based on

978-1-7281-9829-3/20/$31.00 ©2020 IEEE

a match-action architecture. The packet processing pipeline mainly consists of four parts - the parser, the ingress pipeline, the egress pipeline, and the deparser. The parser recognizes and extracts user-defined headers from packets. Both ingress pipeline and egress pipeline consist of a sequence of match-action stages which are used to perform user-defined actions on packets (e.g., modify, forward, or drop). However, P4 only supports limited operations currently. For example, P4 does not support loop operation and cannot access payload when processing packets.

P4 has gained ever-growing interests from the networking community thanks to its flexibility and general-purpose interface. Recent work has shown that many NFs, such as Load Balencer [11] and K-V caching [12], can be accelerated by implementing them in switches. Therefore, some efforts have devoted to accelerating SFC by simply offloading all NFs into underlying high-performance switches [8], [9], [13]. However, there exist NFs that cannot be offloaded to switches due to the limited operations supported by P4.

B. Design Challenges

We encounter three major challenges in designing P4SFC.

Identifying NF's offloadability: Since not all operations are supported by P4, the first challenge is to judge whether an NF can be offloaded to P4-capable switches. To overcome it, we use the widely adopted Click [10] elements to construct NFs. Then, we analyze each element's read/write behaviors to figure out whether it can be implemented in P4. Finally, the offloadability of an NF can be derived by analyzing the offloadability of its elements altogether. More details are introduced in Section III-D.

Maintaining NF state consistency between servers and switches for partially offloaded NFs: The key to ensure NFs processing packets correctly is that NFs can access all states related to the packet being processed. When offloading NFs to switches, offloading the functionality of NFs only is not enough. We also need to install their states into switches at runtime and ensure each packet is processed with the correct states. To achieve this, we carefully design a state library that can automatically maintain NF state consistency between servers and switches when the NF is partially offloaded to a switch. The details are discussed in Section III-D.

Reconfiguring P4-capable switches at runtime: Generally, a P4-capable switch is exclusively occupied by a P4 program and cannot be reconfigured at runtime without interrupting the running P4 program. However, we need to deploy new SFCs or destroy SFCs at runtime dynamically. Therefore, we are challenged to reconfigure the P4-capable switches at runtime with ultra-low overhead. To this end, we design and implement a P4 program whose execution logic can be reconfigured at runtime by modifying its table entries. Details are presented in Section III-F.

III. System Design

In this section, we describe the architecture of P4SFC and elaborate all the components of it.

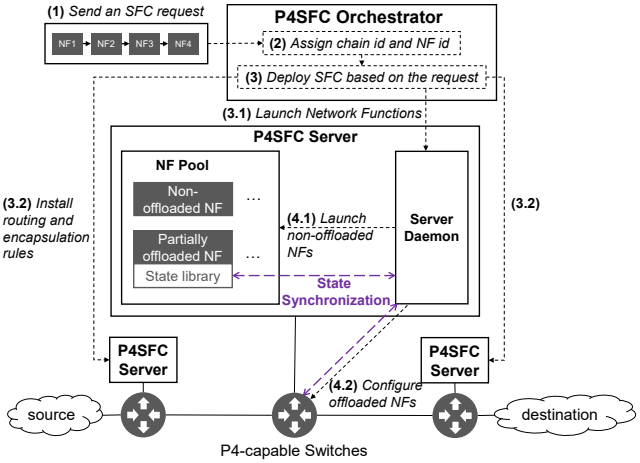


Fig. 1. P4SFC overview using an example of an SFC.

A. Overview

Fig. 1 illustrates the architecture of P4SFC. It is composed of four components: the orchestrator, the server daemon, the NF pool, and the dynamic P4 data plane. To understand how P4SFC works, consider a simple network consisting of three P4-capable switches connected to three servers as shown at the bottom of Fig. 1. Assume that an operator wants to deploy an SFC, as shown in step 1 of Fig. 1. In step 2, the P4SFC orchestrator assign a chain_id to the chain and an nf_id to each NF instance. Afterwards, based on the request, the orchestrator informs related server daemons about how to deploy the NFs (step 3.1) and instructs server daemons to install the routing and encapsulation rules in switches (step 3.2). Then the server daemon will generate the offloading policy for the NFs placed on its server. According to the policy, the server daemon will launch the non-offloaded NFs and partially offloaded NFs in the NF pool (step 4.1), and configure the partially offloaded NFs and fully offloaded NFs in the corresponding switches (step 4.2). At runtime, the state of partially offloaded NFs are synchronized through the state library and the server daemon.

B. Packet Classification and Distribution

One of the most important issues in our system is to distinguish packets that belong to different SFCs, as multiple SFCs may co-exist in the system. Therefore, we tag each packet with its corresponding sfc_id to uniquely identify it. However, knowing that which SFC a packet belongs to is not enough. We also need to know which NF in this chain should process this packet next, so that we can distribute the packet to the correct NF. To achieve this, we also carry the not yet passed NFs' id in the order of SFC in the packet header. In this way, we can distribute packets according to the sfc_id and the first nf_id in the packet header. We show the structure of our customized packet header in Fig. 2.

C. Orchestrator

The P4SFC orchestrator is the entrance for our system. It exposes the functionality of our system to network operators.

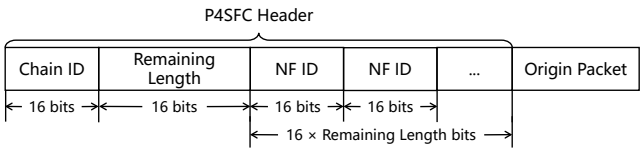


Fig. 2. Structure of P4SFC customized packet header.

TABLE I
COMMON ELEMENTS AND THEIR OFFLOADABILITY

Element	Offloadability
Classifier, IPClassifier, IPFilter	Fully
IPEncap, EtherEncap	Fully
DPI (Deep Packet Inspection)	No
Drop, Alert, Forward (Actions)	Fully
Counter, AverageCounter	Fully
IPRewriter, TCPRewriter	Partially
IPSecDES (encrypting packet using DES-CBC)	No

One can describe his service function chaining intent in JSON format and send it to the orchestrator. The JSON object must have the following three attributes: 1) ***NFs***: describe all NFs that will be used in this chain; 2) ***Links***: describe the links between NFs; 3) ***Route***: describe the route information for this chain. For each NF, three attributes are necessary: *name* is used to identify itself, *type* represents which type this NF belongs to, and *location* is used to describe which server should the NF be deployed.

On deploying a new SFC, the orchestrator needs to complete the following five steps. 1) Assign a unique *chain_id* to the chain. 2) Assign a unique *nf_id* to each NF. 3) Add the attribute *offloadability* to each NF according to the *type* of the NF. The offloadability of each NF type will be analyzed in Section III-D. 4) Group NFs according to their *location* and send each group to the corresponding server daemon running in *location*. 5) Configure the route for the chain according to *Route*.

D. Network Function in P4SFC

The key difference between P4SFC and the previous Click-based SFC systems is that for each SFC we offload proper NFs to P4-capable switches to improve performance. In this subsection, we analyze the offloadability of NFs. As defined in Click [10], an NF is a directed graph composed of packet processing elements. Thus, in the following, we will introduce the offloadability of elements first, then that of NFs.

1) The Offloadability of Packet Processing Elements: Since not all processing logic of elements are supported by P4-capable switches, we surveyed a wide range of elements and classify them into three categories: fully offloadable, partially offloadable, and non-offloadable.

The element that reads or writes packets' payload is non-offloadable, because accessing the payload is not supported by P4-capable switches. The element that updates its table entries (a.k.a., states) dynamically during processing packets is *partially offloadable*, such as the IPRewriter element in NATs and Load Balancers. These elements cannot be fully offloaded because P4 does not support updating its table entries by itself. However, the performance of these elements can still be improved by offloading a part of the element to switches. We extract a new fully offloadable element which does not update its table entries (i.e., it is stateless), from the origin element. The new element only handles the static packet processing procedures. Due to the incompleteness of processing logic, the original element still needs to be run in server to handle the stateful logic. All the remaining elements are *fully offloadable*, such as IPClassifier. We conclude the popular elements and their categories in Table I.

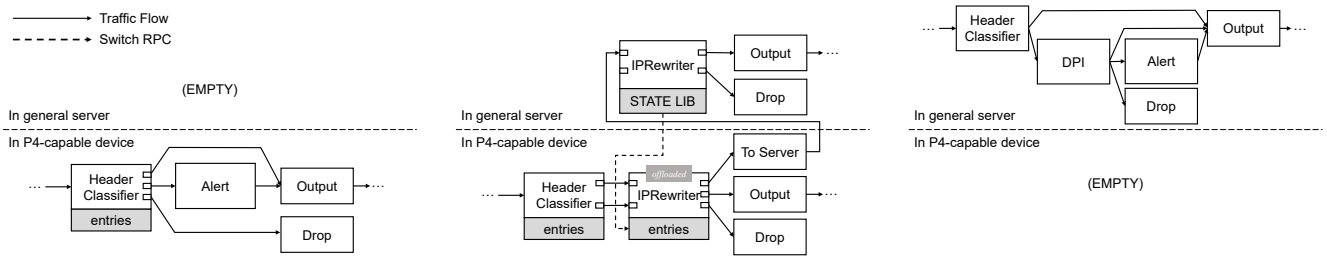
TABLE II
P4SFC ELEMENT STATE PROGRAMMING API

Category	API	Description
Method	*blocking*	Blocking update
	non-blocking	Non-blocking update
P4SFCTable	*init(KeyNames, ValNames)*	Initialize the table
	create(KeyArgs&, ValArgs&)	Create a table entry
	get(KeyArgs&)	Retrieve a table entry
	remove(KeyArgs&)	Remove a table entry
P4SFCCounter	*init(Name)*	Initialize the counter
	get(void)	Retrieve the counter
	set(Val&)	Set the counter

2) State Consistency for Partially Offloadable Elements: For the partially offloaded elements, the states updated in original element need to keep consistent with the offloaded element. Similarly, counters in P4 updated by the offloaded elements also need to be perceived by the original element in servers. Thus, for the ease of development, we design a state library to handle the state consistency issues and hide the communication logic between servers and switches. Table II summarizes the P4SFC state APIs.

We provide two consistent state data structures in our state library for partially offloaded elements. P4SFCTable is a mapping data structure which uses a list of arguments as keys and a list of arguments as values. P4SFCCounter holds an unsigned integer and it corresponds to the counter variable in P4 language. Using this library, developer can update the counter and the table data structure in both server side and switch side, without considering the value synchronization. The library supports both blocking and non-blocking updates.

3) The Offloadability of Modular Network Functions: According to the categories of elements, the NFs composed of these elements can also be classified into three categories: fully offloadable, partially offloadable, and non-offloadable. The sample element graphs for each kind of NFs are illustrated in Fig. 3. A fully offloadable NF is all composed of fully offloadable elements (e.g., Firewall), and it can run completely in P4-capable switches. An NF is partially offloadable only if, in the elements that compose the NF, there is only one partially offloadable element and others are all fully offloadable. (e.g., NAT). For this kind of NF, we extract a fully offloadable element from the partially offloadable element, and replace the origin element in switch side. Then a fully offloadable NF is constructed (in the bottom of Fig. 3(b)). Also, a subgraph starting from the partially offloadable element needs to run in



(a) A fully offloadable example, firewall. (b) A partially offloadable example, NAT. (c) A non-offloadable example, IDS.

Fig. 3. Different NFs with different offloading strategies

TABLE III
POPULAR NETWORK FUNCTIONS AND THEIR OFFLOADABILITY

Network Function	Offloadability
Firewall/Access Control List(ACL)	Fully
Flow Monitor	Fully
Network Address Transmission (NAT)	Partially
IP Security, Virtual Private Network	No
L2/L3 Switch, Router	Fully

the server side to handle the stateful logic (in the top of Fig. 3(b)). When the offloaded NF meets the situation that it can not handle, i.e. new table entry needs to be created, it will send the packet to the NF running in the server. Finally, the remaining NFs are non-offloadable (e.g., IPS), and these NFs can only run in the server. We conclude the offloadability of the common NFs in Table III.

E. Server Daemon

In P4SFC, the server daemon runs in every server and plays a key role in connecting the NFs offloaded to the switch and the NFs in the server. It has three main functional modules - NF deployment module, packet distribution module, and P4 controller module.

1) NF Deployment Module: The NF deployment module is responsible for deploying NFs in this server and offloading proper NFs to the P4 switch that the server is connected to. The NF deployment module first divide these NFs into three sets: *pre_server_NFs*, *server_NFs*, and *post_server_NFs* according to the offloadability of NFs. More specifically, the *pre_server_NFs* contains those NFs that appear before the first non-offloadable NF in the chain, and they will be offloaded to the switch. Similarly, the *post_server_NFs* contains those NFs that appear after the last non-offloadable NF in the chain, and they will also be offloaded. All the other NFs are in the *server_NFs*. After dividing the NFs, the NF deployment module configures the offloaded NFs in the switch and starts all necessary NFs in the server. The NFs started in the server include all the NFs in *server_NFs* and all the partially offloadable NFs in *pre_server_NFs* and *post_server_NFs*.

2) Packet Distribution Module: The packet distribution module is responsible for distributing packets among NFs running in the server. When receiving a packet, The packet distribution module distributes the packet according to our custom header. If the next NF is in this server, the packet is distributed through a virtual NIC bound to the NF; Otherwise, the packet is distributed to the switch.

3) P4 Controller Module: This module acts as a controller for the switch the server directly connects to. All interactions with the switch are done through this module. For example, when deploying an SFC, new configuration rules are installed through this module, and when a partially offloaded NF updates its table entries, the state library accomplish state synchronization through this module.

F. Dynamic P4-Capable Data Plane

Ordinarily, a P4-capable switch is exclusively operated by a program once configured, which means that different programs cannot run simultaneously in a P4-capable switch. Additionally, there is no way to dynamically reconfigure the execution logic of a P4-capable switch without interrupting its execution according to the P4Runtime Specification [14]. When deploying a new SFC in P4SFC, we may need to offload some NFs to switches, which means that the execution logic of those target switches should be reconfigured. Therefore, we are challenging to design a dynamic P4 data plane which can be reconfigured in runtime with ultra-low overhead.

We show our dynamic P4 data plane design in Fig. 4. Firstly, we divide the processing logic in the P4-capable switch into stages. In each stage, one element logic will be executed. Then, configure the execution logic of the switch dynamically is to configure the element to be executed in each stage for each packet dynamically. To achieve this, we apply a match-action table (i.e., **element_control_table**) at the beginning of each stage. The **element_control_table** matches on the packet header and invoke the **set_control_data** action with different parameters based on the matching result. In the **set_control_action**, we will set the control metadata according to the incoming parameters, including: **element_id** that is used to specify which element to be executed in this stage; **is_NF_complete** that is used to indicate whether the element executed in this stage is the last element of the current NF; **next_stage** that is used to specify which stage should this packet enter after this stage. According to the **element_id**, we can first execute the required element logic for this packet. Then, if the executed element is the last element of the current NF, we need to remove the first NF_id from the packet header. Finally, we pass the packet to the next stage as specified in the

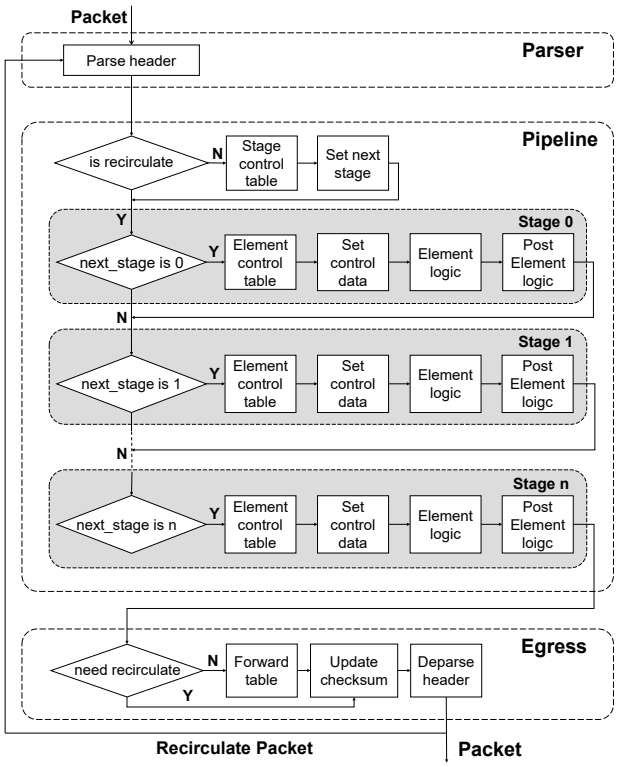


Fig. 4. Design of dynamic P4 data plane.

next_stage field. As P4 does not allow loops in the program, we have no choice but to write a fixed number of stages in the implementation. If an SFC has more elements that need to be executed in a switch than stages, we use the P4 primitive *recirculate* to implement loops, so that any number of elements can be executed in a switch. In order to decide the initial stage for a packet that enter the switch for the first time, we add a **stage_control_table** before the start of stages.

Isolation in the Dynamic P4 Data Plane. A necessary property of the dynamic P4 data plane is isolation. More specifically, at each stage, different configurations (in the form of table entry) belonging to different SFCs use the same **element_control_table**. Meanwhile, if different NFs need to execute the same element at the same stage, they will use the same table of that element. Therefore, we need to ensure that configurations performed for a certain NF or a certain SFC cannot affect any other NFs or SFCs. To achieve the isolation, we add the chain_id and NF_id to the match key of each match-action table. Therefore, each table entry belongs to a specific SFC or a specific NF instance only, which ensures the isolation.

IV. Implementation and Evaluation

Implementation: The implementation of P4SFC is composed of three parts: 1) The orchestrator is implemented in Python and connects to each server daemon using an out-of-band control network. 2) In each server, we run a server daemon that is mainly implemented in Python. The interactions between server daemon and the switch are through the P4Runtime [14] interface. The packet distribution sub-module in the server daemon is implemented using DPDK [15]. We use Click [10] as our NF execution engine in the server and we assign a virtual NIC to each NF. 3) The dynamic P4 data plane is composed of P4-capable switches with a configurable P4 program which is written in P4_16. Moreover, we implement five NFs with different offloadability to evaluate P4SFC.

Testbed: At the time of writing this paper, we have difficulties in deploying the testbed with hardware switches due to various constraints in the COVID-19 pandemic. Therefore, we perform our evaluations based on the standard P4 BMv2 switch and use Mininet to build the topology. We evaluate P4SFC based on a virtual machine, which is equipped with two Intel(R) Xeon(R) E5-2620 v2 CPUs (2.10GHz, 6 physical cores) and 8G RAM. The server runs Linux kernel 4.15.0-101.

A. Performance Improvement

1) NF: To evaluate the performance gains of a single NF, we use packets from 64B to 1500B to measure the throughput and latency of a most common NF (Firewall). The results in Fig. 5(a), Fig. 5(b) show that Firewall in P4SFC stably holds a latency reduction of near 20% compared with Click, while the processing rate gap between them is very small.

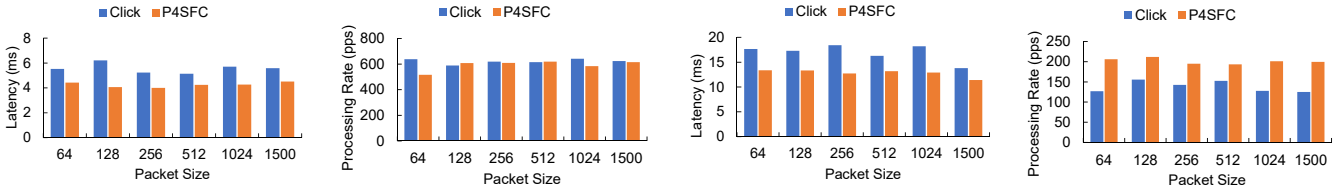
2) SFC: To guarantee the applicability of our P4SFC, we use packets with size 64B to 1500B to evaluate the throughput and latency of a realistic service function chain (IPS $\rightarrow$ Monitor $\rightarrow$ Firewall $\rightarrow$ NAT). The results in Fig. 5(c), Fig. 5(d) show that P4SFC performs better on both the latency (17.5% on average) and the throughput (46.5% on average).

B. Overhead

Introducing dynamic configuration into P4 data plane would lead to performance degradation due to more Match-Action stages. In Fig. 6, compared with native P4 implementation that does not pass through extra Match-Action tables, the latency of elements in our system is slightly higher (about 0.8 ms on average). However, due to the poor performance of Bmv2 switches, the throughput of elements in our system is 50% lower than the native P4 implementation, which can be further reduced in hardware P4-capable switches.

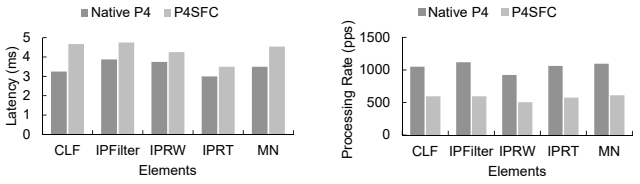
V. Related Work

NFV Acceleration: Many efforts have been devoted to accelerating NFV. Generally, these efforts can be divided into software-based solutions [16], [17] and hardware-based solutions [6], [8], [9], [18]. FastClick [16] integrates both DPDK and Netmap into Click to accelerate Click's packet processing speed. Zhang *et al.* [17] carefully designed efficient data structures and applied various optimizations to achieve better performance. Those software-based acceleration efforts are complementary to our work and can be used to accelerate our server-side NFs. Different with software-based solutions, ClickNP [6] focuses on accelerating NFs with FPGA, while PacketShader [18] uses GPUs in order to alleviate the costly computing need from CPUs. P4SC [8] proposes to use the



(a) Latency of Firewall (b) Processing Rate of Firewall (c) Latency of SFC (d) Processing rate of SFC

Fig. 5. Performance of a single NF Firewall and performance of a realistic sequential chain (IPS → Monitor → Firewall → NAT).



(a) Latency (b) Processing rate

Fig. 6. Overhead of our P4 dynamic data plane (Element CLF stands for classifier, IPRW stands for IPRewriter, IPRT stands for IPRouteTable and MN stands for monitor).

P4-capable device to maintain SFCs for high performance by merging multiple SFCs at code level into one P4 program. Dejavu [9] also tries to accelerate SFCs by P4-capable switches but takes the resource constraints of the switch into consideration. However, they both do not take the limitation of P4 into consideration and only support fully offloadable NFs.

Data Plane Reconfiguration: There has been previous work [19], [20] on how to reconfigure the data plane at runtime. Both Hyper4 [19] and HyperV [20] introduce a hypervisor into the P4-capable device to enable runtime reconfiguration of the switch. Different from the previous work, our dynamic P4 data plane uses elements as execution units, which is more compatible with NF offloading.

VI. Conclusion

In this paper, we present P4SFC, a high-performance SFC system that leverages P4-capable switches to accelerate SFCs. We first divide NFs into three categories (i.e., fully offloadable, partially offloadable and non-offloadable) according to the limitations of P4. Then, when deploying new SFCs, we offload proper NFs to the P4-capable switches to accelerate SFCs. To deploy new SFCs at runtime, we design a dynamic P4 data plane whose execution logic can be configured at runtime without interrupting the current execution logic. Besides that, we also design a state library to maintain state consistency between servers and switches for partially offloaded NFs to ensure correctness. Experiments on real-world SFCs show that P4SFC can improve the performance of SFCs significantly compared to the software-based SFC implementation.

Acknowledgments

The work was supported by the National Natural Science Foundation of China under Grant No. 61972101. Jin Zhao is the corresponding author.

References

[1] P. Quinn and T. D. Nadeau, "Problem statement for service function chaining," *RFC 7498*, 2015.
[2] R. Guerzoni *et al.*, "Network functions virtualisation: an introduction, benefits, enablers, challenges and call for action, introductory white paper," in *Proc. SDN and OpenFlow World Congress*, 2012.
[3] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu *et al.*, "Ananta: Cloud scale load balancing," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 207–218, 2013.
[4] K. Zhang, B. He, J. Hu, Z. Wang, B. Hua, J. Meng, and L. Yang, "G-net: Effective gpu sharing in nfv systems," in *Proc. USENIX NSDI*, 2018, pp. 187–200.
[5] J. Hwang, K. K. Ramakrishnan, and T. Wood, "Netvm: High performance and flexible networking using virtualization on commodity platforms," *IEEE Trans. Netw. Service Manag.*, vol. 12, no. 1, pp. 34–47, 2015.
[6] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, and P. Cheng, "Clicknp: Highly flexible and high-performance network processing with reconfigurable hardware," in *Proc. ACM SIGCOMM*, 2016, pp. 1–14.
[7] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014.
[8] X. Chen, D. Zhang, X. Wang, K. Zhu, and H. Zhou, "P4SC: towards high-performance service function chain implementation on the p4-capable device," in *Symp. IFIP/IEEE IM*, 2019, pp. 1–9.
[9] D. Wu, A. Chen, T. S. E. Ng, G. Wang, and H. Wang, "Accelerated service chaining on a single switch ASIC," in *Proc. ACM HotNets Workshop*, 2019, pp. 141–149.
[10] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM TOCS*, vol. 18, no. 3, pp. 263–297, 2000.
[11] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *Proc. ACM SIGCOMM*, 2017, pp. 15–28.
[12] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proc. ACM SOSP*, 2017, pp. 121–136.
[13] M. He, A. Basta, A. Blenk, N. Deric, and W. Kellerer, "P4NFV: an NFV architecture with flexible data plane reconfiguration," in *Proc. IEEE CNSM*, 2018, pp. 90–98.
[14] "P4runtime specification," https://p4.org/p4runtime/spec/v1.1.0/P4Runtime-Spec.html, accessed Apr. 16, 2020.
[15] "Intel dpdk," https://www.dpdk.org/, accessed Jan. 16, 2020.
[16] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *Symp. ACM/IEEE ANCS*, 2015, pp. 5–16.
[17] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. Ramakrishnan, and T. Wood, "Opennetvm: A platform for high performance network service chains," in *Proc. ACM HotMIddlebox*, 2016, pp. 26–31.
[18] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: a gpu-accelerated software router," in *Proc. ACM SIGCOMM*, 2010, pp. 195–206.
[19] D. Hancock and J. Van der Merwe, "Hyper4: Using p4 to virtualize the programmable data plane," in *Proc. ACM CoNext*, 2016, pp. 35–49.
[20] C. Zhang, J. Bi, Y. Zhou, A. B. Dogar, and J. Wu, "Hyperv: A high performance hypervisor for virtualization of the programmable data plane," in *Proc. IEEE ICCCN*, 2017, pp. 1–9.