P4SC: Towards High-Performance Service Function Chain Implementation on the P4-Capable Device

Xiang Chen*, Dong Zhang*, Xiaojun Wang†, Kai Zhu‡, Haifeng Zhou‡
*College of Mathematics and Computer Science, Fuzhou University, Fuzhou, China
†R&D Institute, Ruijie Networks Co., Ltd., Fuzhou, China
‡College of Computer Science and Technology, Zhejiang University, Hangzhou, China

***Abstract*—Most Service Function Chains (SFCs) in Network Function Virtualization (NFV) are realized on the software or offloading to the network interface card (NIC) and FPGA. However, the software introduces significant performance overhead while the NIC and FPGA suffer from the limited processing capability and the development complexity, respectively. In response, we present P4SC, a system for implementing SFCs on the P4-capable device. P4SC provides the high-performance and flexible SFC implementation by combining the hardware capability and the P4 programmability. It offers some high-level primitives for operators to build the SFC requests and converts the requests to a corresponding P4 program. Moreover, P4SC merges several SFCs to implement them on the same target while observing the P4 grammar. Besides, P4SC provides the runtime management of SFCs by wrapping the low-level device APIs. We evaluate the P4SC performance on various P4 devices, including a Barefoot Tofino switch. Experimental results show that compared to a state-of-the-art NFV framework, P4SC can achieve a 96.98% SFC processing delay decrease on the P4-capable device.**

***Index Terms*—Service Function Chain, P4, Network Function Virtualization, Software-Defined Networking.**

I. Introduction

Network Function Virtualization (NFV) implements Network Functions (NFs) on low-cost servers and provides the flexibility of NF management, while Software-Defined Networking (SDN) decouples the control plane and the data plane and offers unprecedented network programmability. Their combination introduces efficient deployment of NFs in Service Function Chains (SFCs) [1, 2]. However, software-based SFCs suffer unacceptable performance overhead in terms of limited processing capability and high processing latency [3, 4, 5]. For example, Ananta Software Muxes introduces from 200 μs to 1 ms latency at 100 Kilo packets per second (Kpps) due to heavy processing bottleneck [3, 5], which is unsuitable for many low latency applications.

Recently, some research efforts choose to exploit advanced technologies like Data Plane Development Kit (DPDK) [6] to improve the performance of software-based SFCs. However, our experiment results in Section IV reveal that the processing latency of DPDK-based solution is 33× of the hardware-based solution in the worst-case scenario, which is still unacceptable. Besides, some recent works have been devoted to accelerate SFCs by offloading SFC operations to network interface card (NIC) [7] or the Field Programmable Gate Array (FPGA) [8]. Nevertheless, the limited processing capability of NIC [9] cannot meet the performance requirements of SFCs. Moreover, although state-of-the-art FPGA devices can achieve considerable performance compared to dedicated Application-Specific Integrated Circuit (ASIC) hardware [10], operators need to master a Hardware Description Language (HDL) (e.g., Verilog), which exposes the low-level hardware architecture and the digital logic design, to implement their SFCs on the FPGA devices [8, 11]. Such unwelcome development complexity brings the inflexible and inefficient SFC implementation.

978-3-903176-15-7

Therefore, in this work, we exploit the benefit of P4 [12], a domain-specific language for data plane programming, to provide the high-performance and flexible SFC implementation on the P4-capable device. P4 exposes the inner packet processing pipeline of the data plane device and enables operators to customize the behavior of the programmable data plane based on their policies. Unlike the development of FPGA, operators can describe the SFC features in a target-independent P4 program, which can be compiled to various P4 devices, in a few hours. Meanwhile, the P4-capable hardware device, such as Tofino [13], can achieve up to 6.5 Tbps line rates while offering extra-low packet processing latency [14]. Together, the hardware capability and the P4 programmability conform to the requirements of implementing SFCs, which need the high processing performance and the deployment flexibility.

We present P4SC (the abbreviation of "P4 Service Chaining"), a system for implementing SFCs on the P4-capable device. We conclude the following challenges in our design: (1) To avoid the development complexity introduced by substrate details, we are challenged to enable operators to describe the SFC features with the high-level policies; (2) To implement several SFCs on the same target, we are challenged to observe the P4 grammar [15] when maintaining all the SFC features on the output program; (3) For the runtime management of SFCs, we are challenged to provide operators with a convenient way to control the SFC behaviors. In response, P4SC provides some high-level primitives for operators to construct the SFC requests, meanwhile, converts the input SFC requests to a P4 program. It leverages a Longest Common Subsequence (LCS) [16] - based algorithm to merge several SFCs on the output P4 program, while introducing a small program overhead in terms of duplicate P4 tables for

observing the P4 grammar. P4SC also provides the runtime management of SFCs, which enables the implementation of SFC policies and the scheduling of SFCs. Besides, due to the target-independent feature of P4, P4SC can be easily accommodated to various P4 devices, ranging from the software switch to the ASIC-based switch.

In this paper, we make the following contributions:

- We introduce P4SC, a system for implementing SFCs on the P4-capable device that combines the hardware capability and the P4 programmability to provide the high-performance and flexible SFC implementation.
- We present the design and implementation of P4SC. We design several high-level primitives for operators to construct SFC requests. We design a converter and a generator in P4SC to generate P4 program according to input SFC requests. We design a LCS-based algorithm in the converter to merge several SFCs in a P4 program with a small program overhead. We design the generator to wrap device APIs for providing the runtime management of SFCs.
- We implement three real-world SFCs on various P4-capable devices to validate P4SC. Experimental results demonstrate that by integrating the hardware capability into the SFC implementation, P4SC achieves significant SFC performance improvement, including a 96.98% delay decrease compared to the DPDK-based solution.

The remainder of this paper is organized as follows. Section II elaborates the background of implementing SFCs on the P4-capable device and the design challenges of P4SC. The design of P4SC is articulated in Section III. We present the implementation of P4SC and evaluations in Section IV. We summarize the related work in Section V, and conclude this paper in Section VI.

II. Background and Design Challenges

In this section, we start with the background of implementing SFCs on the P4-capable device, and then we reveal the design challenges, which guide the design of P4SC.

A. Implementing SFCs on the P4-Capable Device

A SFC enables the high-level creation and composition of network services and applies value-added services to selected flows [1]. The graph structure of a SFC may be composed of many branches, corresponding to different Service Function Paths (SFPs). In a SFC, a SFC classifier identifies incoming flows and distributes them to different SFPs. A NF of a SFC executes dedicated operations on flows, for example, a firewall validates the flow security to prevent malicious intrusions. Moreover, the metadata is used to exchange the processing information between NFs. At runtime, operators can dynamically select the SFP to process flows based on their policies. Thus, a system for implementing SFCs on the target device is supposed to shield low-level details when describing SFCs, maintain the SFC features, including the SFC classifier, SFPs and the metadata, on device configurations, and enable the runtime management of SFCs.

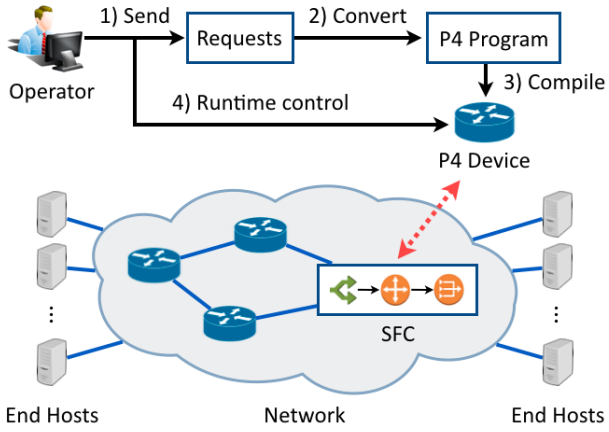


Fig. 1: Operator issues the SFC requests, which are converted to the corresponding P4 program. After compiling and implementing the P4 program, P4SC provides operators with a convenient way to control the SFCs running on the P4-capable device at runtime.

Simultaneously, P4 [12] is a domain specific language, which describes the packet processing pipeline of the data plane device. A P4 program is composed of headers, parsers, metadata, match-action tables, actions, and control flows. A match-action table matches the packet and selects an action to execute that can be used as a part of NF operations or a flow classifier. The metadata is used to exchange the processing information between tables, and a control flow realizes NFs by ordering tables and the processing logic. Therefore, a P4 program is capable of describing a SFC by using a P4 table as the SFC classifier, composing NFs with tables and control flows, exploiting metadata to exchange the processing information between NFs, and defining SFPs in control flows. A P4 compiler loads the program to target devices and exposes the device APIs for operators to populate rules at runtime.

Fig. 1 presents an overview of our P4SC system, which implements SFCs on the P4-capable device. Operators describe the SFC features in the high-level SFC requests. After that, P4SC converts these requests to the corresponding P4 program and implements it on the target device. At runtime, operators manage SFCs by controlling the device behaviors via this system.

B. Design Challenges

We reveal three major challenges in the design of P4SC.

SFC development complexity avoidance: An efficient SFC development requires minimizing the development complexity. However, the substrate details, such as the P4 grammar, introduce the non-trivial complexity when implementing SFCs on the P4-capable device. Therefore, we are supposed to provide a simple approach for operators to implement SFCs without involving any complex substrate knowledge. In response, we design several high-level primitives in P4SC,

TABLE I: The primitives for constructing the SFC requests

Primitives	Descriptions
NF_1 *before* NF_2	The execution priority of NF_1 is higher than NF_2.
NF_1 *then* NF_2 *or* NF_3 [*or* NF_i]	There are two (or more) branches after the processing of NF_1. The first branch processes the packets with NF_2 while the second branch processes the packets with NF_3.
NF_1 *then* NF_2 *or* NF_3 [*or* NF_i] *and branch_end*=NF_N	Unlike the second primitive, these branches are ending with the processing of NF_N.
NF_1 *first*	The processing of NF_1 is placed in the beginning of SFC.
NF_1 *last*	The processing of NF_1 is placed in the end of SFC.
NF_1 *loop*	After the execution of NF_1, the processing restarts from the beginning of SFC.
End_of_Request	This primitive marks the end of a request. It is used as the delimiter between two individual SFC requests.

in order to shield substrate details and enable operators to describe the SFC requests based on their high-level policies. (Section III-B1)

Correct and efficient conversion mechanism: The conversion mechanism, which converts the input SFC requests to the corresponding P4 program, should precisely merge all the SFC features and correctly express them on the output program. However, this conversion may violate the P4 grammar due to the dependency conflicts between different NFs and multiple NF invocations. The strawman solution for merging SFCs introduces lots of duplicate P4 tables for ensuring the program correctness, which is inefficient and unacceptable. Therefore, care must be taken in the conversion mechanism of P4SC. To this end, P4SC ensures the correctness of the conversion from the SFC requests to the P4 program, while using a LCS-based algorithm to merge SFC requests efficiently. (Section III-B2)

Convenient runtime management: At runtime, operators need to control the SFCs running on the P4-capable device. Therefore, we are challenged to provide the runtime management for operators to control the SFC behaviors. However, the device APIs used to populate the control rules are coupled with the details of the P4 program, which brings unwelcome management difficulties. In response, P4SC encapsulates the low-level device APIs to provide a convenient way for the runtime management of SFCs. (Section III-C2)

III. Design of P4SC

In this section, we describe the architecture of P4SC and elaborate two key components of P4SC, the converter and the generator.

A. Overview

Fig. 2 plots the architecture components of P4SC. The converter of P4SC enables operators to describe the SFC features in high-level requests without involving any substrate details. It extracts the SFC features from input requests and convert them to an intermediate representation (IR). Moreover, in the scenario of implementing several SFCs, the converter uses a LCS-based algorithm to merge SFCs while observing the P4 grammar by introducing a small number of duplicate P4 tables. The generator of P4SC generates the output P4 program based on the IR. Meanwhile, it provides the runtime management for operators to control SFCs on the P4-capable device.

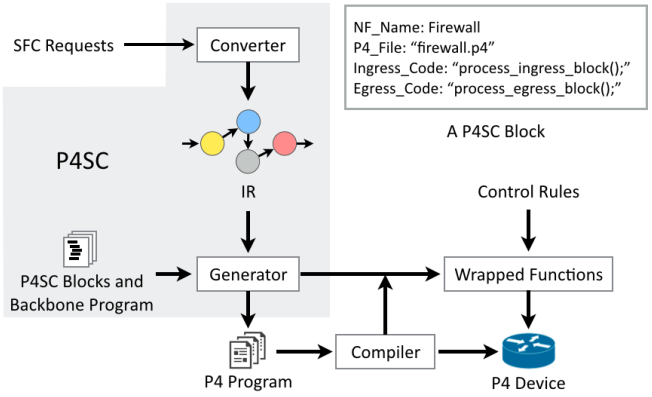


Fig. 2: P4SC architecture components

We further define a *P4SC block*, which is composed of the NF name, the P4 file that stores P4 codes, and the codes of P4 ingress/egress control flows, as shown in Fig. 2. A P4SC block corresponds to a specific NF. Instead of writing a P4 program, operators only need to write their NFs using P4 and import them into P4SC. P4SC will automatically convert the P4 NFs to reusable P4SC blocks. Besides, operators can also utilize 20+ built-in P4SC blocks that are extracted from *switch.p4* [17] to build SFCs.

B. The Converter of P4SC

The converter of P4SC offers some high-level primitives for constructing the SFC requests. A request, which is composed of NF names and primitives, describes the SFC features. The converter converts each input request to a directed acyclic graph (DAG). It uses a LCS-based algorithm to merge these DAGs to an IR, and delivers the IR to the generator of P4SC.

1) ***Converting the SFC Requests to DAGs:*** The converter provides the primitives listed in Table I and available NFs for operators to construct the SFC requests. The converter checks the structure of SFC indicated by a request and rejects non-DAG SFCs with the construction failure. Note that a request can indicate a non-DAG SFC for two reasons: (1) A NF can appear multiple times in a request, or (2) the SFC has loop conditions. In response, the converter requires operators to rename the NF, which is invoked again with a serial number as writing requests, and uses a node attribute to indicate the start of a loop. It allocates an unique SFC ID to each request and creates NF nodes. Each NF node is associated with some

Algorithm 1 Merging two DAGs

1: **function** MERGING($DAG1, DAG2$)
2: Get *order1* and *order2* by sorting *DAG1* and *DAG2*
3: $sharedOrder \leftarrow LCS(order1, order2)$
4: **if** *sharedOrder* is None **then**
5: **return** $AND(DAG1, DAG2)$
6: **end if**
7: Get *Base* and *Attach* from *order1* and *order2*
8: Split *Attach* to *mainSgmt*, *first*, *follow*
9: Insert *first* and *follow* to *Base*
10: $p \leftarrow$ the place after *first* on *Base*
11: **for each** *node* in *mainSgmt* **do**
12: **if** *node* in *sharedOrder* **then**
13: $p \leftarrow$ the place of *node* on *Base*
14: $id1,\ id2 \leftarrow Base[p].sfcID,\ node.sfcID$
15: $Base[p].sfcID \leftarrow MERGE(id1, id2)$
16: **else**
17: Insert *node* to *Base*
18: **end if**
19: $p++$
20: **end for**
21: $IR \leftarrow ADDLINK(Base, DAG1, DAG2)$
22: **return** IR
23: **end function**

attributes, including the NF name, the node length, which is equal to the number of P4 tables occupied by the NF, a SFC ID array for identifying the DAGs that utilize this node, and a pointer list used to connect to other nodes. Meanwhile, the converter strips out serial numbers from NF names and connects the NF nodes based on the node order acquired from the input request to produce the DAG.

*2) **Merging DAGs to the IR**:* When merging DAGs to the IR, we consider two problems of maintaining the correctness: (1) *NF successor dependency conflicts:* As specified in [12, 15, 18], there may exist NF successor dependency conflicts between two different DAGs. For example, "NF1 *before* NF2" in DAG A violates "NF2 *before* NF1" in DAG B. Such situation brings the failure of SFC implementation. (2) *Multiple NF invocations:* The P4 grammar forbids the multiple invocations of a P4 table. In this case, a NF node can only be accessed once in a P4 program. However, different DAGs may visit the same NF node, which violates the P4 grammar when merging DAGs.

The parallelism method: The strawman solution for merging DAGs called *the parallelism method* introduces a pre-visiting node to distribute flows and connects this node with original DAGs in parallel. To resolve the above-mentioned problems when merging DAGs, this solution creates duplicate P4 tables for each DAG to observe the P4 grammar. However, as a compromise, lots of resources are wasted due to the exponential number of P4 tables.

Our solution: In response, to ensure the program correctness while avoiding the heavy P4 program overhead, we develop a LCS-based algorithm in the converter of P4SC, as described in Algorithm 1. The converter iterates this algorithm to merge DAGs and produces an IR in the end. We compare our solution with the parallelism method in Section IV-E.

The converter takes two DAGs as the input of Algorithm 1. First, it acquires the topological sequences of the DAGs (line 2). By referring to NF node length, the LCS produces

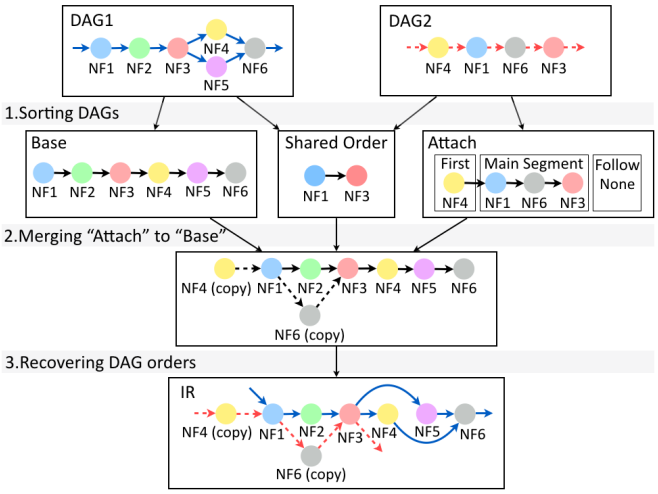
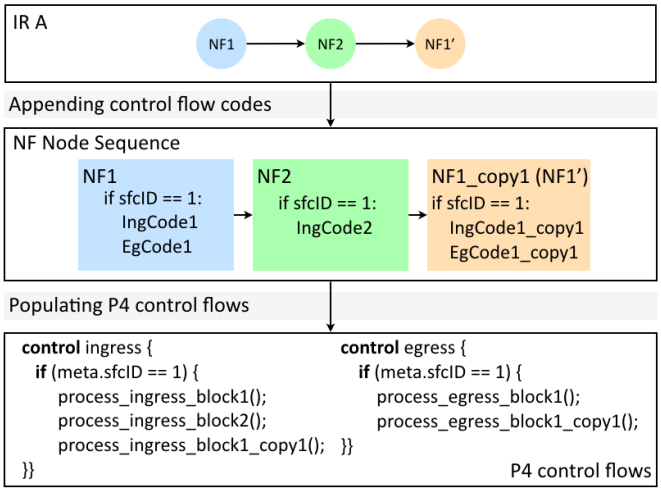


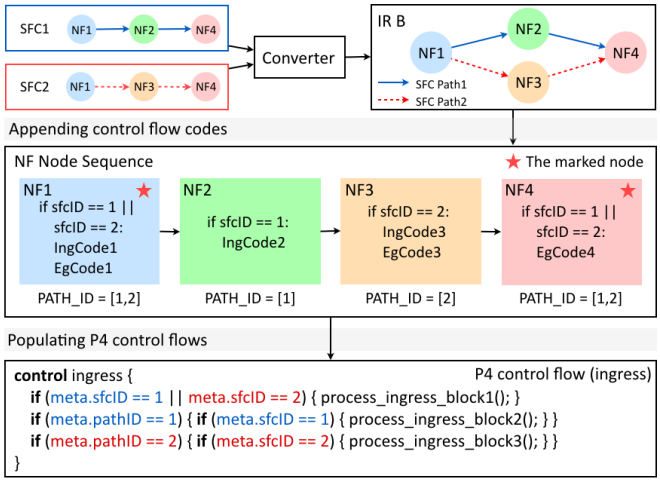
Fig. 3: An example of Algorithm 1

"sharedOrder", which is a NF node sequence that occupies the maximum P4 tables (line 3). If "sharedOrder" is empty, the converter connects the two DAGs to a pre-visiting node and ends up the procedure (lines 4-6). Otherwise, it combines the shorter node sequence "Attach" to another longer sequence "Base". The node sequence between the first node and the last node of "sharedOrder" on "Attach" is named "main segment". Meanwhile, "first" is the node sequence before "main segment" and "follow" is the node sequence after "main segment" (line 8). The converter copies "first" and inserts the replica before the first NF node of "sharedOrder" on "Base". Similarly, the replica of "follow" is placed after the last NF of "sharedOrder" (line 9). The converter uses a pointer "p" to point to the place after "first" on "Base" (line 10). For each node on "main segment", it determines if this node exists in "sharedOrder" or not. If true, the converter combines the SFC ID array of this node with that of the same node on "Base" (lines 12-15). If false, the replica of this node is inserted to the place indicated by "p" (lines 16-17). Then "p" is moved to the next node on "Base" (lines 19). Finally, the converter recovers the structures of the input DAGs on "Base" (line 21) and produces the IR.

Fig. 3 plots an example that presents the mechanism of Algorithm 1, which is a three step process: (1) Algorithm 1 acquires the topological sequences using the topological sorting, and feeds the sequences to LCS to produce "sharedOrder". (2) By referring "sharedOrder", Algorithm 1 splits the shorter node sequence, "Attach", into three subsequences, "first", "main segment", and "follow" ("follow" is none in this case), and individually merges the three subsequences into the longer node sequence, "Base". For handling the "first" and "follow", Algorithm 1 directly copies them and inserts the replicas into "Base". For handling "main segment", Algorithm 1 iterates each node of "main segment" and determines whether the current node exists in "sharedOrder". If so, Algorithm 1 skips this node, e.g., "NF1" and "NF3" in "main segment". Otherwise, Algorithm 1 copies this node and inserts the replica into the "Base", e.g., "NF2" in "main



(a) The workflow for handling IR A



(b) The workflow for handling IR B

Fig. 4: The generator workflow for handling IR examples

segment". (3) Finally, Algorithm 1 recovers the original DAG structures on the merged sequence and produces the IR.

C. The Generator of P4SC

The generator of P4SC generates the P4 program based on the IR. Meanwhile, it wraps the low-level device APIs to provide the runtime management of SFCs.

1) ***Generating the P4 Program based on the IR****:* Above all, the generator handles the NF nodes that appear multiple times in the IR. It searches the P4SC blocks with the name of duplicate nodes and creates an unique block replica for each duplicate node in the IR. The name of block replica is appended with a serial number to distinguish it from the original name.

Thereafter, the generator produces the *NF node sequence* of the IR using topological sorting and records IR branches in the linked lists. A linked list is assigned a path ID that corresponds to a SFP, and each node in the NF node sequence is assigned the codes of P4 control flows that are recorded in relevant P4SC blocks. The generator uses *if-else* statements for SFC IDs to indicate the boundary between different SFCs

TABLE II: The primitives for SFC management

Primitive names	Descriptions
Show_SFCs()	Show all the SFCs.
Show_SFPs(*sfcID*)	Show all the SFPs of a SFC.
Show_SFC_Configs(*sfcID* [*pathID*])	Show the configuration of a SFC/SFP.
Select_SFC(*sfcID* [*pathID*])	Select a specific SFC/SFP.
Delete_SFC(*sfcID* [*pathID*])	Delete a specific SFC/SFP.

on the codes of P4 control flows. Subsequently, the generator introduces an empty control flow pair for ingress and egress control flows. It selects the nodes that exist in all the linked lists and marks them in the NF node sequence. Then it traverses the NF node sequence and identifies whether a node is marked. If so, the generator directly populates the codes of P4 control flows recorded in this node to the control flow pair. Otherwise, it acquires path IDs of the linked lists in which this node exists, and inserts the codes of P4 control flows into the control flow pair as well as using *if-else* statements for path IDs to set the boundary of SFPs. If a node has an attribute that indicates a loop, the generator adds the loopback action to the last P4 table cited by this node.

Fig. 4 shows the generator workflow for handling two IR examples. Fig. 4a shows that to process the duplicate node "NF1_copy1", the generator creates a block replica, which is associated with "NF1". Fig. 4b shows that the boundary between SFC1 and SFC2 is set by using *if-else* with SFC IDs on the control flows. Fig. 4b also shows that the codes for "NF1" are directly populated because "NF1" exists in all the SFPs, while other codes are limited by path IDs.

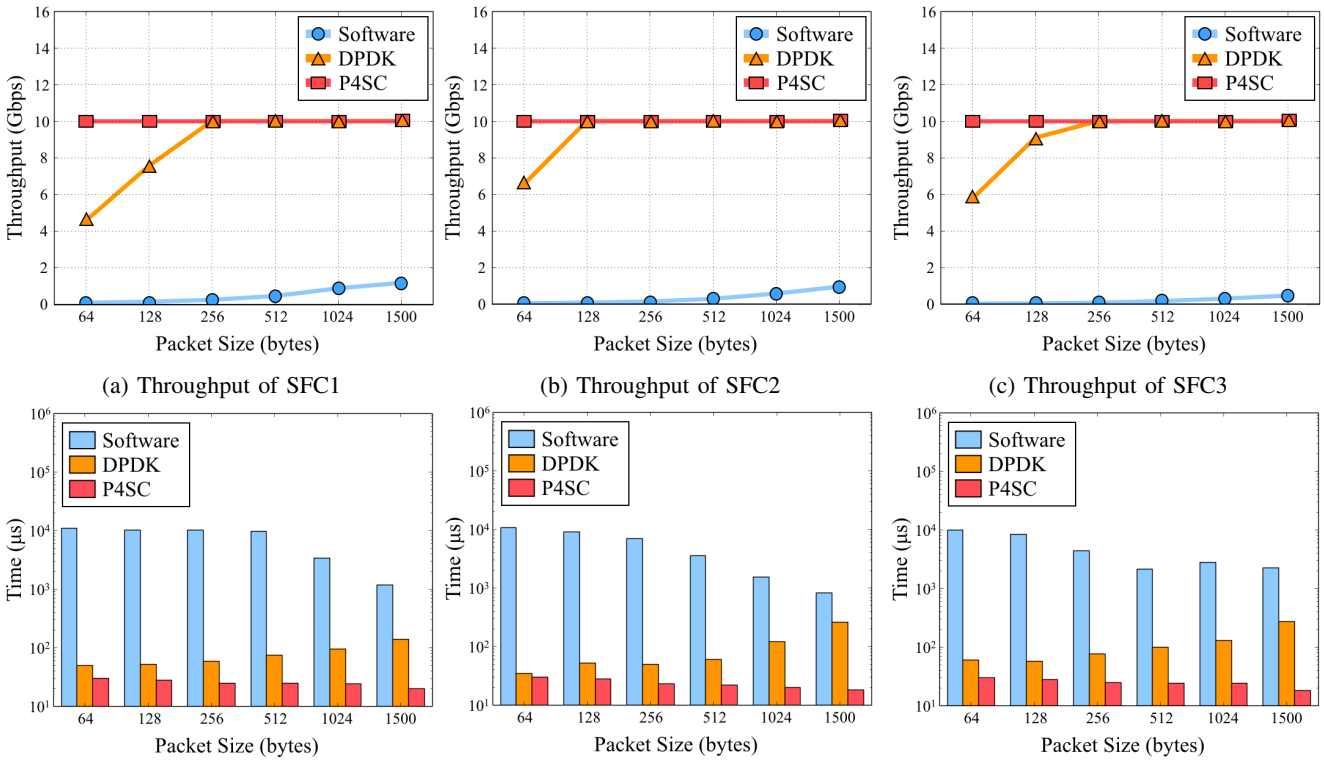
Finally, the generator inserts a P4 table used to classify flows and distribute path IDs and SFC IDs in the beginning of the ingress control flow. It produces the output program by combining the control flow pair with a target-dependent backbone program, which provides the target-dependent definitions, such as the definition of intrinsic metadata. The operators can change this backbone program to accommodate to other P4 devices.

2) ***Runtime Management****:* Another function of the generator is encapsulating the device APIs generated by the P4 compiler, as well as providing wrapped functions for operators to manage the SFCs running on the target device. In the design of the generator, we focus on two aspects of the runtime management, *NF rule management*, and *SFC management*.

NF rule management: The universal approach for populating the NF rules to a P4-capable device is to leverage the device APIs generated by the P4 compiler. However, these APIs are tightly coupled with the details of the P4 program, such as the name of a P4 table. These unwelcome details are supposed to be transparent to operators, who only care about the SFC policies and the NF rules. For example, operators care about the rules of their NF, "Firewall", rather than the table entries of the P4 table "firewall_t". Accordingly, we design the generator to encapsulate the device APIs into the wrapped functions for populating the NF rules without involving any low-level details. Operators can easily invoke these functions in a script written in a high-level program-

TABLE III: Real-world SFCs and the features of output P4 programs

The name of SFC	Scenario	The SFC request	No. of tables	No. of matches	No. of actions
SFC1 for DC	Data Center	IDS *before* Firewall *before* NAT *before* L3fwd	17	47	40
SFC2 for HTTP services	Mobile Network	LB *then* L3fwd *or* Firewall *and branch_end*=Firewall, Firewall *before* NAT	9	48	31
SFC3 for Gi-LAN	Gi-LAN	NAT *before* L2fwd *before* LB *before* L3fwd *before* Firewall	15	61	50



(a) Throughput of SFC1 (b) Throughput of SFC2 (c) Throughput of SFC3

(d) Execution delay of SFC1 (e) Execution delay of SFC2 (f) Execution delay of SFC3

Fig. 6: Performance of real-world SFCs

ming language like Python to populate their NF rules. They can also use the CLI provided by the generator for the NF rule management.

SFC management: In addition, the generator also provides some primitives, as listed in Table II, for managing the SFCs running on the P4-capable device. Unlike the wrapped functions for managing NF rules, these primitives are designed to control SFPs on the target device. For example, it is necessary to provide operators, who determine the SFC to process the packets based on policy intents, with the primitive of selecting SFC running on the target device, i.e., "Choose_SFC".

IV. Implementation and Evaluation

A. Implementation

The implementation of P4SC is composed of two parts: (1) The converter and the generator of P4SC are implemented as a plugin of the P4 compiler, P4C [19]. The compiler compiles the P4 program produced by P4SC and generates the target configurations. And then P4SC configures the target device with the target configurations to accomplish the SFC implementation. (2) The generator of P4SC relies on the Apache Thrift framework [20] and P4Runtime [21], which is a state-of-the-art P4 control framework, to communicate with the SFC target devices and populate control rules at runtime. Moreover, we implement the following NFs to evaluate P4SC:

L2fwd: A packet forwarder that matches the destination MAC address of the packet using exact match to determine the output port with 100 rules.

L3fwd: A packet forwarder that matches the source and destination IP addresses of the packet using longest prefix match to determine the output port with 100 rules.

Firewall: A 5-tuple firewall configured with 100 rules.

NAT: A stateless NAT that translates the source IP address of the packet according to 100 rules.

LB: An ECMP-based load balancer that hashes the 5-tuple of the packet to balance the load with 100 rules.

IDS: A simple NF similar to the Bro intrusion detection system [22] with 100 rules.

Fig. 5: The experiment topologic

We have published the source code of P4SC system as well as relevant experimental details including the P4 codes of these NFs, the parameter and the benchmark workload used in our experiments, etc., at [23].

B. Evaluation Overview

Our evaluation answers three important questions: (1) Can P4SC provide a high-performance SFC implementation compared to existing SFC solutions? (Section IV-C) (2) Can P4SC efficiently generate the P4 program and control the SFC target devices? (Section IV-D) (3) Can P4SC merge SFCs in a short time while introducing small program overhead? (Section IV-E)

We run the P4SC system in Ubuntu 16.04 system on a server, which is configured with two Intel(R) Xeon(R) E5-2630 v4 CPUs (2.20GHz, 10 physical cores) and 128GB RAM. The topologic for evaluation is depicted in Fig. 5. On the whole, the experiment results reveal that (1) P4SC provides the high-performance SFC implementation on a P4 hardware target compared to both the software solution and the DPDK-based solution, (2) the converter and the generator can quickly generate the P4 program based on the given SFC requests, while avoiding the heavy P4 program overhead in terms of duplicate P4 tables, and (3) the additional latency introduced by the runtime management of P4SC is acceptable.

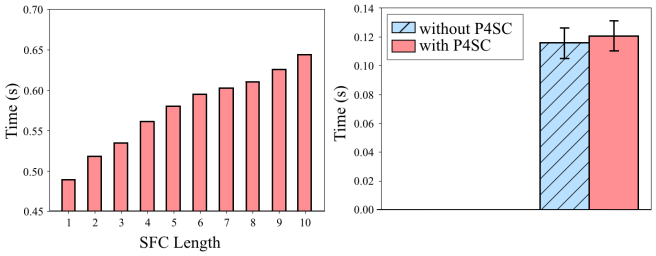
C. Performance of Real-World SFCs

In this experiment, we choose three real-world SFCs to evaluate the performance of P4SC:

SFC1 for Data Center (DC): There are two kinds of traffic in the DC, the east-west traffic between servers, and the north-south traffic from the outside of the DC. We present a SFC that provides security for the north-south traffic [24]. At the beginning of SFC, IDS and Firewall performs security check on incoming traffic to defend against malicious attacks. Thereafter, NAT converts between a public address domain and a private address domain, while L3fwd routes the traffic and connects the DC with the Internet.

SFC2 for HTTP Services: We illustrate a SFC for HTTP services [24]. This SFC is composed of LB, firewall and NAT. At runtime, LB distributes the HTTP traffic and the non-HTTP traffic to two branches. To enhance the performance, one branch forwards the HTTP traffic to go through a performance enhancement proxy (PEP). The non-HTTP traffic in another branch skips the operations of PEP. Thereafter, firewall applies security strategies, and NAT executes the private-to-public address transition.

SFC3 for Gi-LAN: The Gi interface is a major mobile traffic carrier between the external packet data network and the gateway general packet radio service (GPRS) support node



(a) Execution time (b) Runtime overhead

Fig. 7: Efficiency of P4SC

[25]. Considering the requirements of service-level agreement (SLA), the Gi-LAN requires the dynamical deployment of SFCs to accommodate the traffic growth. We extract a SFC for Gi-LAN from [25] that schedules flows to go through NAT, L2fwd, LB, L3fwd, and firewall in this order.

We write corresponding SFC requests of these real-world SFCs, while using P4SC to implement these SFCs on a typical P4-capable device, a Barefoot Tofino switch [13]. Table III presents the SFC requests and the features of P4 programs produced by P4SC. Moreover, we choose the BMv2 switch [26] as the software target to implement the same SFCs as a comparison. We also use a DPDK-based SFC target, Berkeley Extensible Software Switch (BESS) [27, 28] v0.3.0, to implement these SFCs. We use MoonGen [29] to generate the test traffic at 10 Gbps. We use 64B to 1500B packets to evaluate the throughput of SFC and measure the packet processing latency.

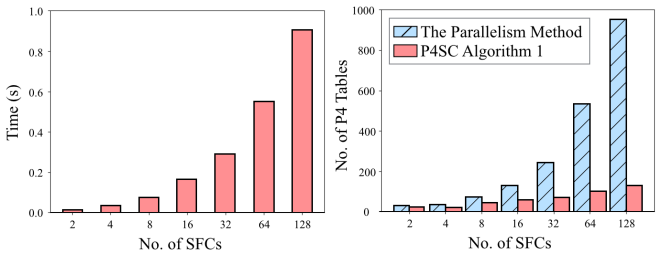
Fig. 6 shows that the processing capability of Tofino can improve the performance of P4SC-based SFCs significantly. Compared to the software-based SFCs, the P4SC-based SFCs achieve orders of magnitude performance improvement on both throughput and packet processing delay. Meanwhile, although the DPDK-based SFCs could achieve as high throughput as the P4SC-based SFCs, the P4SC-based SFCs outperform the DPDK-based SFCs on the packet processing delay with up to 96.98% decrease. These results demonstrate that P4SC is competent to achieve high-performance SFC implementation.

D. Efficiency of P4SC

To qualify the efficiency of P4SC, we write ten different SFC requests and send them to P4SC to generate P4 programs for measuring the execution time of P4SC. The execution time of P4SC depends on the length of SFC so that we vary the length of the SFCs from 1 to 10 in these requests.

At first, we measure the execution time of P4SC when converting the requests to P4 programs. Fig. 7a shows that the execution time increases slowly when the length of SFC increases, which could demonstrate the effectiveness of the conversion mechanism of P4SC.

Moreover, we evaluate the additional runtime overhead introduced by P4SC. We use the wrapped functions to populate a NF rule and measure the latency. We repeat the experiments for 100 times. As shown in Fig. 7b, P4SC introduces an average of 0.05 ms delay, which is acceptable.



(a) Execution time (b) No. of P4 tables

Fig. 8: P4SC's effect of merging SFCs

E. Effect of Merging SFCs

We evaluate P4SC's effect of merging SFCs. We randomly generate the SFC requests with 15 different NFs based on the following criteria: (1) The SFC described in a request has 3 branches at most; (2) The NF number of a SFC is less than 15; (3) A NF occurs only once in a request.

Fig. 8a shows that P4SC is capable of merging a hundred of SFCs in less than a second, which is fast and acceptable. In addition, we compare Algorithm 1 of P4SC with *the parallelism method* that inserts a pre-visiting node to merge SFCs. If one SFC has n NFs while another SFC has m NFs, then the output IR produced by the parallelism method has $n+m+1$ nodes. We individually use the two approaches to merge SFCs. We assume that each NF node of the output IR occupies one P4 table. As Fig. 8b shows, compared to the parallelism method, Algorithm 1 introduces a small number of duplicate P4 tables to observe the P4 grammar when merging several SFCs. For example, in the case of merging 128 SFCs, the program produced by Algorithm 1 only uses 13.15% of the tables of the program produced by the parallelism method.

The experiment results demonstrate that the Algorithm 1 of P4SC can quickly adapt to massive requirements of SFC deployment, while avoiding the heavy P4 program overhead.

V. Related Work

SFC Acceleration: Several recent works have been proposed to improve the performance of SFCs. They focus on NF acceleration [8, 30, 31], packet delivery acceleration [4, 6, 7], NF modularization [32, 33], and NF parallelism [34, 35], respectively. Moreover, some efforts attempt to maintain SFCs on the programmable data plane to achieve SFC acceleration. SLA-NFV [36] and HYPER [37] implement SFCs on the hybrid substrate composed of software targets and hardware targets, in order to support a variety of NFs and enable SLA requirements. And NF-Switch [38] presents a switch architecture that reduces additional match stages and operations between NFs. P4SC is complementary to above research efforts. It combines the hardware capability and the P4 programmability to achieve the high-performance and flexible SFC implementation.

NF Orchestration: Many recent works [35, 39, 40, 41] propose the NF orchestration techniques for SFC implementation. Moreover, some research efforts [42, 43, 44] are proposed to reasonably allocate physical resources to NFs when orchestrating NFs to compose SFC. In addition, some recent works [17, 45, 46, 47, 48, 49] also orchestrate and customize NFs using P4. Hyper4 [46] and HyperV [47] focus on virtualizing the P4 programmable data plane to provide network virtualization services. However, they lack of design considerations for building SFCs on the P4-capable device. P5 [48] identifies the dependencies among the P4 tables and creates an efficient switch pipeline by removing unnecessary features between P4-based NFs. ClickP4 [49] refers to the module design and modularizes P4-based NFs to enable the on-demand orchestration. Unlike P5 and ClickP4, P4SC is a comprehensive system that organizes the P4 program based on the SFC features described in the SFC requests.

Besides, [50, 51] propose policy-based NF orchestration. PGA [50] provides a graph-based abstraction for expressing network policies and makes an attempt to support SFCs. It composes a conflict-free graph by parsing high-level policies and uses the composed graph to generate SFC configurations. Similarly, P4SC also provides several high-level primitives for operators to construct SFCs based on policy intents. However, P4SC focuses on converting the SFC requests to the P4 program and implementing SFCs on the P4-capable device, in order to provide the high-performance and flexible SFC implementation.

VI. Conclusion

In this paper, we presented P4SC, a system for maintaining SFCs on the P4-capable device. P4SC offers some high-level primitives for operators to describe SFCs and converts the SFC requests to a corresponding P4 program. It merges several SFCs while observing the P4 grammar and provides the runtime management of SFCs by wrapping the low-level device APIs. We have implemented some real-world SFCs on various P4 devices via P4SC. Our experiment results show that P4SC can improve the performance of SFC significantly compared to the existing SFC solutions. In the future, we will step further to enrich the primitives for constructing SFCs to support more complex SFC implementation.

Acknowledgement

This work is supported by the National Key Research and Development Program of China (2016YFB0800201, 2017YFB0803205), the Key Research and Development Program of Zhejiang Province (2017C01064, 2018C01008, 2018C03052), Major Scientific Project of Zhejiang Lab (No. 2018FD0ZX01), the Fundamental Research Funds for the Central Universities (2016XZZX001-04), and gifts from Ruijie Networks. Dong Zhang is the corresponding author. The authors would like to acknowledge anonymous reviewers for their useful comments and shepherds, Xiaoyan Hong, Yongfeng Sun, and Zhiyuan Len for valuable comments in improving this paper. Dong Zhang was supported by the China Scholarship Council.

REFERENCE

[1] J. Halpern and C. Pignataro, "Service function chaining (sfc) architecture," Internet Requests for Comments, RFC, 2015.

[2] P. Quinn and T. Nadeau, "Problem statement for service function chaining," Internet Requests for Comments, RFC, 2015.

[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 Computer Communication Review*, vol. 43, no. 4, pp. 207–218, 2013.

[4] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *NSDI*. USENIX Association, 2014, pp. 459–473.

[5] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang, "Duet: Cloud scale load balancing with hardware and software," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 27–38, 2015.

[6] Intel. Data Plane Development Kit. [Online]. Available: http://dpdk.org

[7] J. Hwang, K. K. Ramakrishnan, and T. Wood, "Netvm: high performance and flexible networking using virtualization on commodity platforms," *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, pp. 34–47, 2015.

[8] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen, "Clicknp: Highly flexible and high performance network processing with reconfigurable hardware," in *SIGCOMM*. ACM, 2016, pp. 1–14.

[9] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy, "High performance packet processing with flexnic," in *ASPLOS*. ACM, 2016, pp. 67–81.

[10] G. Brebner, "Softly defined networking," in *ANCS*. ACM, 2012, pp. 1–2.

[11] D. F. Bacon, R. Rabbah, and S. Shukla, "Fpga programming for the masses," *Communications of the ACM*, vol. 56, no. 4, pp. 56–63, 2013.

[12] 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 Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[13] Barefoot Network. Barefoot Tofino. [Online]. Available: https://www.barefootnetworks.com/technology/#tofino

[14] "Barefoot: The World's Fastest and Most Programmable Networks." [Online]. Available: https://barefootnetworks.com/media/white_papers/Barefoot-Worlds-Fastest-Most-Programmable-Networks.pdf

[15] P4 Language Consortium, "The p4-14 language specification, version 1.0.4," 2017. [Online]. Available: https://p4lang.github.io/p4-spec/p4-14/v1.0.4/tex/p4.pdf

[16] D. S. Hirschberg, "Algorithms for the longest common subsequence problem," *Journal of the ACM (JACM)*, vol. 24, no. 4, pp. 664–675, 1977.

[17] P4 Language Consortium. switch.p4. [Online]. Available: https://github.com/p4lang/switch

[18] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling packet programs to reconfigurable switches," in *NSDI*. USENIX Association, 2015, pp. 103–115.

[19] P4 Language Consortium. P4C: A new, alpha-quality reference compiler for the p4 programming language. [Online]. Available: https://github.com/p4lang/p4c

[20] Apache Software Foundation, Thrift. [Online]. Available: http://thrift.apache.org/

[21] P4 Language Consortium. P4runtime: A control plane framework and tools for the p4 programming language. [Online]. Available: https://github.com/p4lang/pi

[22] The bro network security monitor. [Online]. Available: https://www.bro.org/

[23] The code of p4sc. [Online]. Available: https://github.com/P4SC

[24] W. Liu, H. Li, O. Huang, B. M., L. N., C. Z., and H. J., "Service function chaining use cases," *IETF*, 2013.

[25] K. Smith, D. Zhou, D. Nehama, G. F. Hyatt II, J. Oliver, T. Cooper, and R. Browne, "Evaluating dynamic service function chaining for the gilan," *Intel Corporation*, 2016.

[26] P4 Language Consortium. Behavioral-Model: A p4 software switch. [Online]. Available: https://github.com/p4lang/behavioral-model

[27] BESS: Berkeley Extensible Software Switch. [Online]. Available: https://github.com/NetSys/bess

[28] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "Softnic: A software nic to augment hardware," *Technical Report, UC Berkeley*, 2015.

[29] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," in *IMC*. ACM, 2015, pp. 275–287.

[30] Z. Zheng, J. Bi, C. Sun, H. Yu, H. Hu, Z. Meng, S. Wang, K. Gao, and J. Wu, "Gen: A gpu-accelerated elastic framework for nfv," in *APNet*. ACM, 2018, pp. 57–64.

[31] K. Zhang, B. He, J. Hu, Z. Wang, B. Hua, J. Meng, and L. Yang, "G-net: Effective gpu sharing in nfv systems," in *NSDI*. USENIX Association, 2018, pp. 186–200.

[32] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: a framework for nfv applications," in *SOSP*. ACM, 2015, pp. 121–136.

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

[34] Y. Zhang, B. Anwer, V. Gopalakrishnan, B. Han, J. Reich, A. Shaikh, and Z.-L. Zhang, "Parabox: Exploiting parallelism for virtual network functions in service chaining," in *SOSR*. ACM, 2017, pp. 143–149.

[35] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu, "Nfp: Enabling network function parallelism in nfv," in *SIGCOMM*. ACM, 2017, pp. 43–56.

[36] C. Sun, J. Bi, Z. Zheng, and H. Hu, "Sla-nfv: an sla-aware high performance framework for network function virtualization," in *SIGCOMM*. ACM, 2016, pp. 581–582.

[37] ——, "Hyper: A hybrid high-performance framework for network function virtualization," *IEEE Journal on Selected Areas in Communications*, vol. 35, no. 11, pp. 2490–2500, 2017.

[38] C.-L. Hsieh and N. Weng, "Nf-switch: Vnfs-enabled sdn switches for high performance service function chaining," in *ICNP*. IEEE, 2017, pp. 1–6.

[39] O. Sefraoui, M. Aissaoui, and M. Eleuldj, "Openstack: toward an open-source solution for cloud computing," *International Journal of Computer Applications*, vol. 55, no. 3, pp. 38–42, 2012.

[40] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "Simple-fying middlebox policy enforcement using sdn," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 27–38.

[41] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags." in *NSDI*, vol. 14, 2014, pp. 543–546.

[42] H. Moens and F. De Turck, "Vnf-p: A model for efficient placement of virtualized network functions," in *CNSM*, 2014, pp. 418–423.

[43] M. C. Luizelli, L. R. Bays, L. S. Buriol, M. P. Barcellos, and L. P. Gaspary, "Piecing together the nfv provisioning puzzle: Efficient placement and chaining of virtual network functions," in *IM*. IEEE, 2015, pp. 98–106.

[44] J. G. Herrera and J. F. Botero, "Resource allocation in nfv: A comprehensive survey," *IEEE Transactions on Network and Service Management*, vol. 13, no. 3, pp. 518–532, 2016.

[45] A. Sivaraman, C. Kim, R. Krishnamoorthy, A. Dixit, and M. Budiu, "Dc.p4: Programming the forwarding plane of a data-center switch," in *SOSR*. ACM, 2015, pp. 2:1–2:8.

[46] D. Hancock and J. van der Merwe, "Hyper4: Using p4 to virtualize the programmable data plane," in *CoNEXT*. ACM, 2016, pp. 35–49.

[47] 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 *ICCCN*. IEEE, 2017, pp. 1–9.

[48] A. Abhashkumar, J. Lee, J. Tourrilhes, S. Banerjee, W. Wu, J.-M. Kang, and A. Akella, "P5: Policy-driven optimization of p4 pipeline," in *SOSR*. ACM, 2017, pp. 136–142.

[49] Y. Zhou and J. Bi, "Clickp4: Towards modular programming of p4," in *SIGCOMM*. ACM, 2017, pp. 100–102.

[50] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang, "Pga: Using graphs to express and automatically reconcile network policies," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 29–42, 2015.

[51] E. J. Scheid, C. C. Machado, R. L. dos Santos, A. E. Schaeffer-Filho, and L. Z. Granville, "Policy-based dynamic service chaining in network functions virtualization," in *ISCC*. ACM, 2016, pp. 340–345.