st-SFC: Optimizing Dynamic Deployment of Stateful SFCs on P4-based PDP Switches

Tingyu Li , Zhihuang Ma and Zuqing Zhu, *Fellow, IEEE*

***Abstract*—With the rapid development of network function virtualization (NFV), there is an increasing trend of implementing virtual network functions (vNFs), especially the stateful ones, on high-performance programmable data plane (PDP) switches (*e.g.*, the P4-based PDP switches based on Tofino ASICs), and forming stateful service function chains (SFCs) with them. However, the capability of PDP switches on supporting stateful SFCs is still restricted by the limited hardware resources in them. In this work, we study how to optimize the deployment of stateful SFCs in P4-based PDP switches and design the system of st-SFC, so as to not only utilize the hardware resources on switches efficiently but also minimize the overhead of interactions between control and data planes. We first consider the deployment of stateful SFCs on a single PDP switch. Specifically, we propose to abstract each stateful vNF as a state machine and design a stateful SFC building algorithm to merge the state machines of vNFs for reducing redundant resource usages, and for the vNFs whose operations involve interactions with the control plane, we develop a *PktIn-Table* to reduce the resource usage in PDP switches and the interaction latency. Then, we propose an SFC deployment algorithm that realizes stateful SFCs on PDP switches on demand, aiming to optimize the resource usages across all the switches in runtime. We prototype st-SFC with PDP switches based on Tofino ASICs and demonstrate its effectiveness experimentally.**

***Index Terms*—Service function chain (SFC), Stateful packet processing, Programmable data plane (PDP), P4.**

I. Introduction

NOWADAYS, network function virtualization (NFV) [1] has become one of the key technologies to accommodate the ever-increasing traffic, users, and applications in the Internet [2, 3]. Specifically, NFV addresses the shortcomings of the traditional way of deploying network services with special-purpose middle-boxes (*e.g.*, low cost-effectiveness, complex maintenance, and long time-to-market) by decomposing network services into the virtual network functions (vNFs) that can be instantiated on general-purpose software and hardware platforms. Therefore, service providers (SPs) can deploy network services in a much more cost-efficient and timely way: instantiating the vNFs of a network service on general-purpose platforms on demand and steering traffic through the vNFs in sequence to form a service function chain (SFC) [4–8].

Note that, vNFs can be instantiated on both software platforms (*e.g.*, virtual machines and containers) and hardware platforms (*e.g.*, bare metal servers and programmable data plane (PDP) switches) [9]. Among the NFV platforms, the PDP switches based on Tofino ASICs [10] are gaining more and more attention recently, due to their high-throughput packet processing at line-rates of up to 100 Gbps, low latency in sub-microseconds, and the protocol-independent switch architecture (PISA) that benefits from P4 programmability [11]. Their packet processing capability can perfectly adapt to the recent advances on fiber-optic networking [12–16]. Hence, people have leveraged such P4-based PDP switches to develop various network services with vNFs and demonstrated superior packet processing performance [17–23]. However, the PDP switches' capability of supporting NFV is still restricted by the limited hardware resources in them. Specifically, if we want to deploy an SFC on one PDP switch, the number of pipeline stages in the switch restricts the length and logical complexity of the SFC, while the memory resources in each stage, which can be 1.28 MB of static random-access memory (SRAM) and 67.6 KB of ternary content-addressable memory (TCAM) [24], limits the number of flows that can use the SFC. This makes it critical to optimize the resource allocation in the switch's pipeline when deploying SFCs there.

Previous studies have tackled the problem of allocating memory in PDP switches to deploy various SFCs in their pipelines [25–29]. Nevertheless, they only considered stateless SFCs, which may suffer from the latency and bandwidth overheads caused by frequent interactions with the control plane when the network environment is highly dynamic. Therefore, researchers have tried to retain certain state information on PDP switches and realize stateful SFCs[1] with P4 programming primitives and registers to achieve ultra-low processing latency and high throughput [30–34]. Although the registers in a PDP switch can be leveraged to realize stateful packet processing [35], the number of registers and their actions are so limited that a wide range of stateful vNFs can hardly be supported simultaneously. For example, the actions of the registers in certain Tofino ASICs can only support two or fewer conditional statements [10], which makes it infeasible to fully accommodate some classic stateful vNFs, *e.g.*, the TCP-based stateful firewall, leading to inevitable interactions with the control plane for state updates [20, 21]. Meanwhile, there are other types of stateful vNFs (like heavy hitter), which may consume many registers to store state information. Hence, it is relevant to study how to successfully deploy various stateful vNFs in PDP switches and optimize their resource allocations, to not only assign hardware resources to vNFs efficiently but also minimize the interactions between control and data planes.

T. Li, Z. Ma 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).

Manuscript received on November 14, 2023.

[1]In this work, we distinguish stateful and stateless vNFs based on whether they store per-flow/per-packet information in the data plane, and thus stateful load balancer and network address translator (NAT) are also considered as stateful, not only the vNFs that rely entirely on the data plane for state updates.

To this end, it is of great importance to design a widely-applicable approach that can orchestrate various stateful vNFs to realize SFCs with them in PDP switches. First of all, in the face of a wide variety of stateful vNFs, we need to define a generic way to classify them and decompose each vNF in fine-grained logic unit combinations accordingly. Note that, due to the variety and amount of vNFs that need to be supported in an NFV system, treating each vNF as atomic during SFC deployment (*e.g.*, the scheme in [36]) might not be resource-efficient, and thus we have to consider merging vNFs to reduce the redundant logic units. Second, we need to clarify how to leverage P4 programming primitives and hardware resources to realize the logic units in the pipelines of PDP switches. Lastly and most importantly, a highly-efficient vNF merging method should be designed to merge various vNFs based on their logic unit combinations and the available hardware resources in each PDP switch, to minimize resource usage. Here, we hope to point out that it is not good enough to only merge the same tables in vNFs (like the method proposed in [35]) since it is also possible to merge tables of different types.

In this work, we propose, implement, and demonstrate an aforementioned widely-applicable approach, namely, st-SFC, which can deploy stateful SFCs in P4-based PDP switches to not only utilize the hardware resources on switches efficiently but also minimize the overhead of interactions between control and data planes. We first leverage the idea of improved state machine [37] to abstract each stateful vNF as a state machine that consists of a state transition module and an action module. Then, we propose a stateful SFC building algorithm to merge the state machines of vNFs of different types for reducing redundant resource usages in PDP switches. Meanwhile, for the vNFs whose operations involve interactions with the control plane, we design a *PktIn-Table* to reduce the resource usage in PDP switches and the interaction latency. Next, we develop an SFC deployment algorithm that realizes stateful SFCs on PDP switches on demand, aiming to optimize the hardware resource usages across all the switches in runtime. Finally, we prototype st-SFC with PDP switches based on Tofino ASICs and commodity servers and demonstrate its performance experimentally. Our results verify that st-SFC can provision dynamic SFC requests with high resource-efficiency, and outperform benchmarks in terms of resource utilization and interaction latency between control and data planes.

The rest of the paper is organized as follows. We briefly survey the related work in Section II. Section III describes our proposal of abstracting stateful vNFs as state machines and building SFCs accordingly. The algorithm for deploying stateful SFCs across PDP switches is explained in Section IV. In Section V, we discuss the implementation and demonstration of st-SFC. Finally, Section VI summarizes the paper.

II. Related Work

Previously, there have been a number of studies on how to deploy stateful vNFs on PDP switches. Some of them focused on realizing a specific type of vNFs, such as heavy hitter [17], stateful firewall or packet filter [20, 21], stateful load balancer [18, 22, 23], fast rerouting engine [19], and stateful NAT [23]. As they only considered a single type of vNFs, their solutions were neither generic enough to support a wide variety of vNFs nor capable of deploying an SFC that consists of multiple vNFs. In order to find a generic way of deploying various vNFs in a PDP switch, the proposals like OPP [30], FlowBlaze [32], and SDPA [37] considered various system architectures to define the improved state machines for stateful vNFs, and researchers also tried to develop customized-designs of PDP switches specifically for supporting stateful packet processing [31, 33, 34], *e.g.*, the authors of [31] proposed a PDP switch that can realize line-speed stateful packet processing and be programmed with DOMINO, which is a C-like high-level language. However, all these approaches were not based on P4-based PDP switches that equip Tofino ASICs.

For deploying stateful vNFs on P4-based PDP switches with high resource-efficiency, one needs to merge vNFs to reduce redundant resource usages. Previous investigations in [25–29] have considered how to merge P4 programs. Nevertheless, they did not address the P4 programs for stateful vNFs, and thus their schemes were not fine-grained enough for merging stateful vNFs. To realize an SFC that consists of multiple vNFs on a P4-based PDP switch, people have tried to pre-deploy various vNFs in the switch's pipeline and steer traffic through them by leveraging recirculation [38, 39]. However, using recirculation in a PDP switch degrades the performance of line-speed packet processing in its pipeline, especially when an SFC includes many vNFs whose sequence cannot be determined in advance. Although recirculation can be avoided if the SFC is deployed on a "big switch" that groups multiple PDP switches [29], resource-efficiency will be degraded. There were also other studies on deploying SFCs on P4-based PDP switches [36, 40–44], but they focused on enabling runtime reconfigurability of SFCs deployment. Hence, they did not optimize the tradeoff between resource-efficiency and packet processing performance, and thus are irrelevant to this work.

pSFC [35] considered fine-grained table entry merging to reduce the redundant logic units in stateful SFCs, but it only tried to merge the same tables in stateful vNFs. Moreover, it only addressed the stateful vNFs that fully rely on the registers in PDP switch for state updates. These two issues limit its universality, and our st-SFC can address them properly.

III. Strategies for Building Stateful SFC

In this section, we design the strategies for st-SFC to decompose stateful vNFs, merge their state machines, and update their states by interacting with the control plane.

A. Pipeline Model for P4-based PDP Switch

To explain our proposal clearly, we establish the following model for the pipeline in a PDP switch and the SFCs deployed in it. We denote an SFC that consists of n stateful vNFs as $S = \{V_1, \cdots, V_n\}$, where each vNF V_i can be decomposed into m_i logical units as $V_i = \{L_{i,1}, \cdots, L_{i,m_i}\}$, which can be match action tables (MATs) $\{t_{i,u}\}$, register actions $\{R_{i,u}\}$, individual actions $\{A_{i,u}\}$, and conditional nodes $\{c_{i,u}\}$. The relation between logic units $L_{i,u}$ and $L_{i,v}$ is represented by a binary variable $d_{(v,u)}$, which equals 1 if $L_{i,v}$ depends on $L_{i,u}$

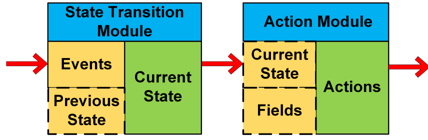


Fig. 1. Example on abstracting a stateful vNF as a state machine.

(*i.e.*, $L_{i,u}$ should be executed before $L_{i,v}$), and 0 otherwise. Each MAT $t_{i,u}$ can be denoted as a tuple $t_{i,u} = \{t^M_{i,u}, t^A_{i,u}\}$, where $t^M_{i,u}$ and $t^A_{i,u}$ are the sets of match items and actions in the MAT, respectively. Each register action $R_{i,u}$ is also a tuple $R_{i,u} = \{R^I_{i,u}, R^A_{i,u}\}$, where $R^I_{i,u}$ is the unique index of the register referred by it and $R^A_{i,u}$ is the set of actions that can be performed on the register.

B. Strategy for Decomposing Stateful vNFs

To deploy SFCs that consists of stateful vNFs in the pipeline of a PDP switch with high resource-efficiency, we need to find a generic way to decompose various stateful vNFs into fine-grained logical units, so as to facilitate the subsequent merging of vNFs. Hence, we propose to abstract each stateful vNF as a state machine that consists of a state transition module (ST-M) and an action module (A-M), as shown in Fig. 1. Each packet entering the vNF is first processed by the ST-M, which maintains the state information of the vNF (*e.g.*, per-packet state, per-flow state, and link state) based on internal (captured locally by the PDP switch) or external (informed by the control plane) events, to obtain the current state of the vNF. Next, the A-M matches to certain field(s) in the packet and applies proper action(s) to it according to the matching result and the current state. We implement the ST-M and A-M with the MATs and registers in the switch pipeline. Note that, certain stateful vNFs may also include pre-processing (*e.g.*, hash), and thus a more generic model of stateful vNF should include the ST-M, A-M and a preprocessing module (P-M). Based on their operation principles and the involvement of the control plane in them, we classify stateful vNFs into three categories.

The first category is for the vNFs that can solely rely on the switch pipeline to realize stateful packet processing (*e.g.*, the heavy hitter detector). In this case, as shown in Fig. 2(a), the ST-M can be realized with a register action, where the register index can be determined by the flow characteristics of a packet, such as the hash value of its 5-tuple, and the set of actions defines the state transitions. The A-M can be realized with an MAT that matches to the current state of the vNF (and certain field(s) in each packet) to obtain the actions to apply.

The second category is for the vNFs whose state transitions need the assistance from the control plane, like a stateful firewall. As shown in Fig. 2(b), the ST-M of a vNF in this category can still be realized with a register action, but the state updates in it are driven by the control plane. Therefore, the ST-M also includes a digest module that works as the interface to interact with the control plane. The implementation of the A-M is the same as that in the first category.

The third category is for the vNFs that use the entries of MATs instead of registers to store their state information, such as stateful load balancer and stateful NAT. As shown in Fig.

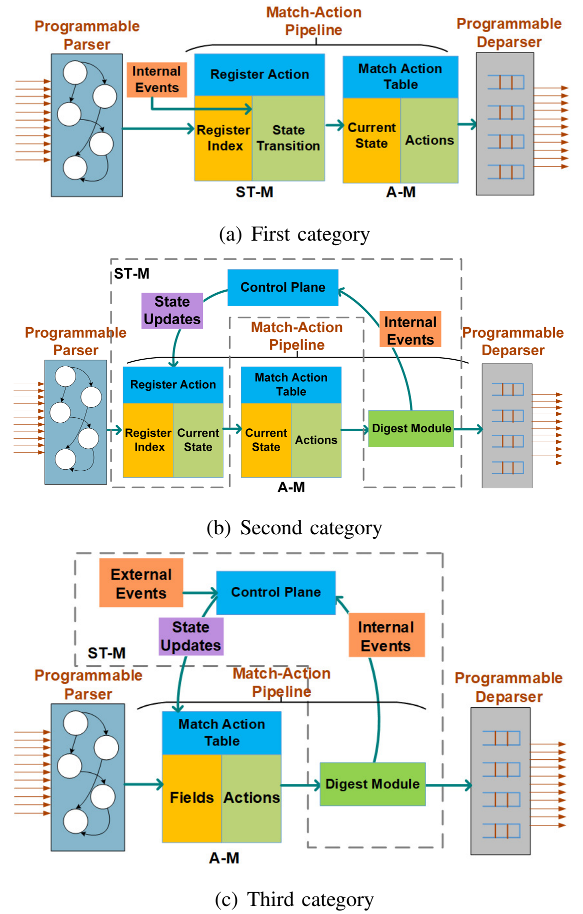


Fig. 2. State machine implementations for three categories of stateful vNFs.

2(c), the ST-M is realized solely in the control plane this time, while the A-M is an MAT that matches to certain packet field(s). When the state of the vNF changes, the control plane updates the MAT to modify its match items or/and actions.

C. Strategy for Merging Stateful vNFs

After abstracting each vNF as a state machine and decomposing it into the ST-M and A-M, we merge the resulting logic units (MATs and register actions) to reduce redundant resource usages, with the following three types of merging principles.

- *Exact Merge*: If two logic units in two vNFs are exactly the same, we can merge them [35]. Here, we say two logic units are the same if they are of the same type and have the same elements. For example, two MATs $t_{i,u}$ and $t_{j,v}$ are the same, if we have $t^M_{i,u} = t^M_{j,v}$ and $t^A_{i,u} = t^A_{j,v}$.
- *Same Action Merge*: If two logic units only contain the same actions, we can also merge them. This principle is usually used to merge the A-Ms in different vNFs. For instance, two MATs $t_{i,u}$ and $t_{j,v}$ can be merged with this principle if we have $t^M_{i,u} \neq t^M_{j,v}$ but $t^A_{i,u} = t^A_{j,v}$.
- *Same Match Merge*: With this principle, we can merge the logic units that only have the same match items.

To the best of our knowledge, the last two merging principles have not been explored in the literature. Therefore, we introduce two illustrative examples in Fig. 3 to explain them and show why resource-efficiency can be improved by them.

Fig. 3(a) shows an example on the same action merge. In this work, to save hardware resources, we define the state of each stateful vNF as a bitmap. For instance, the binary state values of the first MAT are $\{0001, 0000\}$, where the lowest bit is the significant bit in the MAT's state value, while the state values of the third MAT are $\{0100, 1000, 0000\}$, where the two highest bits are the significant bits. We can see that the three MATs use the same action "output to *Port* 1", and thus we can merge the entries in them to get a common one with a state value as $0111 = 0001+0010+0100$. Meanwhile, the state value of the original entry for "output to *Port* 2" in the third MAT gets changed to $1011 = 0001 + 0010 + 1000$, to avoid the drop cases in the first and second MATs. This is because in a PDP pipeline, the action of "drop" has the highest priority when a positive match is encountered, *i.e.*, a packet will be dropped immediately if it matches to any entry for dropping in an MAT. Therefore, in Fig. 3(a), when merging the MATs to get the entry for "output to *Port* 2", we merge the entry of "output to *Port* 2" in the last MAT with those of "output to *Port* 1" in the first two MATs, to avoid dropping packets incorrectly. If we assume that the sizes of the match, state and action in the MATs are 16, 8 and 8 bits, respectively, the table entries in the three MATs before merging consume a total of 128 bits of SRAM, while the merging reduces SRAM usage to 64 bits[2]. On the other hand, if we only merge the first two MATs, the saved SRAM will be reduced to 32 bits.

Fig. 3(b) explains the same match merge. Here, two MATs t_{1,u_1} and t_{2,u_2} contain the same match, *i.e.*, $t^M_{1,u_1} = t^M_{2,u_2}$. Then, we can merge the MATs to generate a new one that performs both actions $\{t^A_{1,u_1}, t^A_{2,u_2}\}$ after a positive match of t^M_{1,u_1} has been obtained. In this way, if we assume that the sizes of the match and action in the MATs are 16 and 8 bits, respectively, the merging saves 16 bits of SRAM.

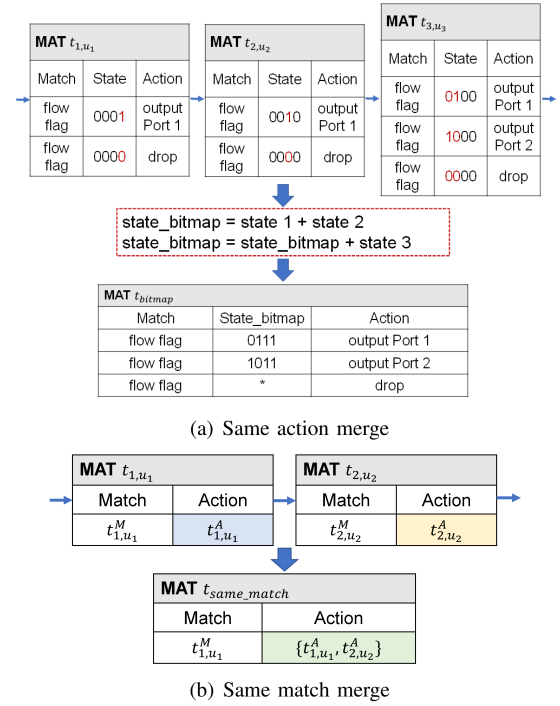


Fig. 3. Examples of two proposed merging principles.

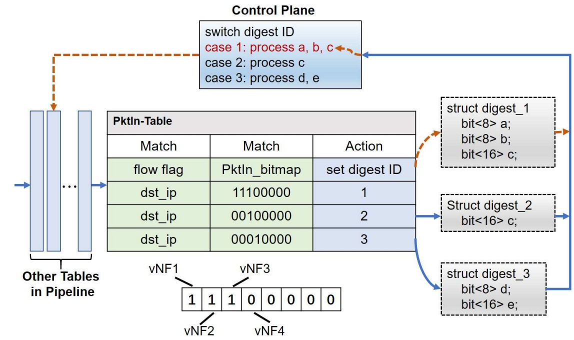


Fig. 4. Example on interacting with control plane with *PktIn-Table*.

D. Strategy for Interacting with Control Plane

As we have explained before, certain stateful vNFs cannot accomplish their state updates solely in the pipeline of a PDP switch, and thus need to interact with the control plane. The conventional way of interacting with the control plane in a PDP switch based on Tofino ASICs is to specify a unified digest structure for uploading state information, which includes all the information that can be uploaded by vNFs in an SFC. This scheme is not efficient because an interaction for state update might not need to upload all the state information of the vNFs, and the resulting redundant bandwidth and processing overheads will degrade the performance of the interaction between the control and data planes, making it the bottleneck of stateful packet processing.

To address the issue above, we design a *PktIn-Table* to reduce the overheads due to interacting with the control plane. As shown in Fig. 4, we denote each stateful SFC with a bitmap and use each bit in it to represent a vNF that is in the SFC and needs to upload state information to the control plane. Then, the digest module checks the bitmap to encode a digest ID and only upload the state information that is truly necessary.

[2]Note that, the entries for "drop" are default actions in the MATs and thus do not occupy any memory.

For instance, in Fig. 4, the first entry in the *PktIn-Table* uses a bitmap of 11100000 to denote the SFC that consists of *vNFs* 1-3, and assigns its digest ID as 1 to denote that its state information to the control plane contains three fields, *i.e.*, a and b in 8 bits each and c in 16 bits. After receiving the digest message for state information, the control plane parses its digest ID to find the right procedure to process the state information in it. For example, in Fig. 4, after receiving a message with its digest ID as 1, the control plane invokes the procedure to process the state information of a, b and c.

E. Algorithm for Building a Stateful SFC

With the strategies discussed above, we design *Algorithm* 1 to merge stateful vNFs in an SFC to generate a stateful SFC. *Line* 1 first decomposes each vNF in a stateful SFC $S = \{V_1, \cdots, V_n\}$ into three modules (*i.e.*, P-M, ST-M and A-M) according to the strategy discussed in Section III-B. Each module m contains a set of logical units, denoted as $\Lambda_{n,m}$. Then, for each vNF V_i in S, we check the vNFs after

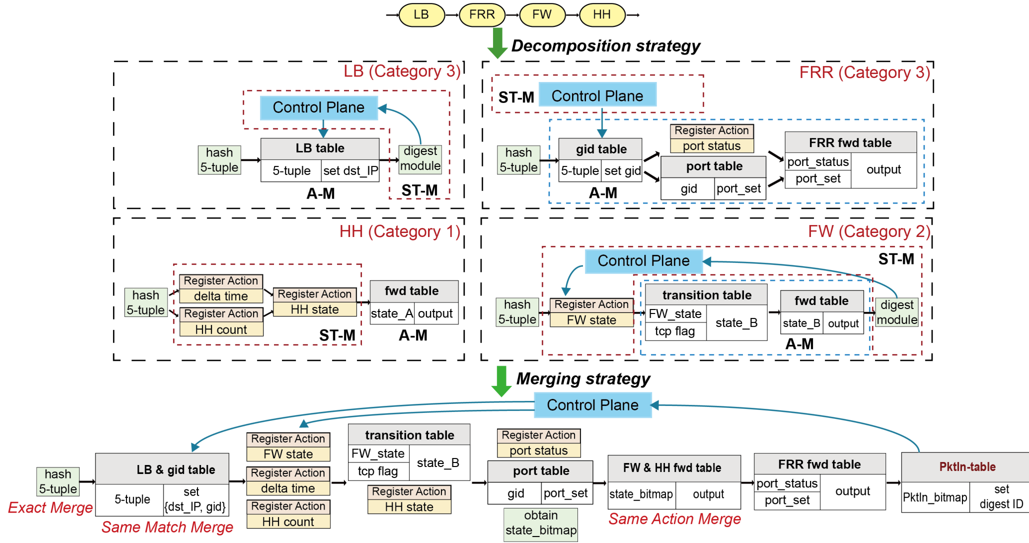


Fig. 5. Example of an overall st-SFC building strategy consisting of 4 st-vNFs.

it. Specifically, we consider modules of the same type in V_i and V_j and to see whether the logical unit $L_{i,u}$ in $\Lambda_{i,m}$ can be merged with $L_{j,v}$ in $\Lambda_{j,m}$ (*Lines* 2-14). Here, we say two logic units $L_{i,u}$ and $L_{j,v}$ are mergeable if the following two conditions are satisfied: 1) one of the three merging principles designed in Section III-C can be applied to merge them, and 2) merging $L_{i,u}$ and $L_{j,v}$ will not affect the operations of vNFs V_i and V_j (*i.e.*, the relations between logic units in them will not be impacted). Hence, if two logic units $L_{i,u}$ and $L_{j,v}$ are mergeable, we merge them to $\tilde{L}$ and replace them with it to save hardware resources (*Lines* 7-9). Finally, we find the vNFs that need to interact with the control plane for state updates and design *PktIn-Table* for them accordingly (*Line* 15).

Algorithm 1: Building a Stateful SFC by Merging vNFs

1. decompose each vNF in a stateful SFC $S = \{V_1, \cdots, V_n\}$ into P-M, ST-M and A-M, where the set of logical units contained in each module m is $\Lambda_{n,m}$;
2. **for** *each* $i \in [1, n-1]$ **do**
3.   **for** *each* $j \in [i+1, n]$ **do**
4.   **for** *each* $m \in [1, 3]$ **do**
5.   **for** *each* $L_{i,u} \in \Lambda_{i,m}$ **do**
6.   **for** *each* $L_{j,v} \in \Lambda_{j,m}$ **do**
7.   **if** *$L_{j,v}$ can be merged with $L_{i,u}$* **then**
8.   merge $L_{j,v}$ and $L_{i,u}$ to replace them;
9.   **end**
10.   **end**
11.   **end**
12.   **end**
13.   **end**
14. **end**
15. find the vNFs that need to interact with control plane and design *PktIn-Table* accordingly;

Fig. 5 gives an example on building a stateful with merged vNFs. We need to build an SFC that consists of a load balancer (LB), a fast re-routing (FRR), a TCP firewall (FW), and a heavy hitter (HH) in sequence. The vNFs are first decomposed into logic units according to their categories (defined in Section III-B), where LB and FRR belong to the third category, FW and HH are in the second and first categories, respectively. Then, the logic units of the vNFs are merged as follows: 1) as the vNFs all include a logic unit for calculating hash value with 5-tuple, the logic units can be merged with the exact merge, 2) as both LB and FRR contain an MAT whose match item is the 5-tuple of packet, the MATs can be merged with the same match merge, and 3) as both FW and HH use an MAT whose action is to forward packet out, the MATs can be merged with the same action merge. Finally, the configuration of the stateful SFC is shown in the bottom of Fig. 5 with the merged logic units.

IV. Deploying Stateful SFCs over PDP Switches

In the previous section, we explain how to build a stateful SFC by merging its vNFs, with the assumption that the SFC will be deployed on one PDP switch. However, in a real-world NFV environment, various stateful SFCs can be requested by clients in runtime and the hardware resources in a PDP switch might not always be sufficient for SFC deployment. Hence, in this section, we extend st-SFC such that it can deploy stateful SFCs over multiple PDP switches in runtime.

A. System Architecture for Runtime SFC Deployment

Fig. 6 shows the overall system architecture of st-SFC, to support the deployment of stateful SFCs over multiple PDP switches. The data plane is mainly built with P4-based PDP switches, each of which includes a pipeline for SFC deployment and a local controller for control plane operations. Each local controller can communicate with the global controller for network-wide coordination. During system initialization, the global controller downloads specific P4 programs to certain PDP switches according to the output of the st-SFC deployment algorithm that will be discussed in the next subsection. Then, during runtime, when a client requests to be served with a stateful SFC, the global controller finds out which switch pipeline(s) should serve the client flow to realize the requested SFC, by leveraging a client flow routing algorithm.

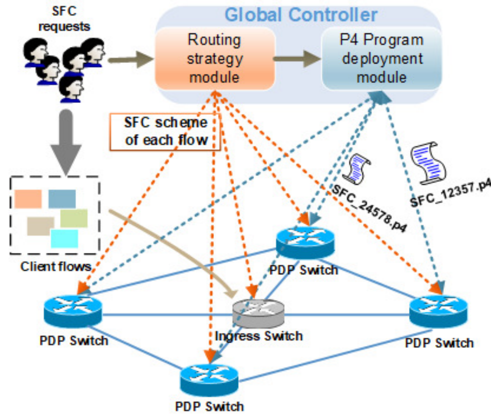


Fig. 6. Overall system architecture of st-SFC with multiple PDP switches.

Then, it installs corresponding table entries in the selected PDP switch(es) to provision the requested SFC to the client, without interrupting any in-service SFCs.

To realize the st-SFC system mentioned above, we first need to accomplish runtime stateful SFC deployment. Due to the variety of stateful SFCs and the limited hardware resources in PDP switches, it is impractical to deploy a corresponding P4 programm for each stateful SFC. Therefore, we try to pre-deploy certain stateful SFCs, each of which is the superset of a number of stateful SFCs, on PDP switches during system initialization. In other words, each superset SFC can support several types of SFCs, each of which uses part of the vNFs in it. Then, in runtime, the client traffic of an SFC can be steered through the SFC's superset SFC by leveraging a dual-match approach that considers both clients and flows. Specifically, we index each client with a matching item, namely, *Client Flag*, which refers to a packet field that can distinguish the clients, *e.g.*, source or destination IP address, and each flow of a client is still be identified by its 5-tuple. Hence, when a packet enters a logic unit in a superset SFC, the logic unit first matches to its *Client Flag* to see whether it should be processed there. If the logical unit is an MAT, a matching entry for *Client Flag* is added. Otherwise, we can add a new MAT in the logic unit with the matching item as *Client Flag*.

Fig. 7 provides an example on the procedure of runtime stateful SFC deployment discussed above. Here, the pre-deployed superset SFC contains LB, FRR, FW and HH in sequence, and we need to provision three SFC requests in runtime, which consist of LB→FRR→FW, LB→FW→HH, and FW→HH, respectively. Then, by matching to the *Client Flag*, the traffic of the three SFC requests can be distinguished at each logic unit in the superset SFC and thus realize the three SFCs simultaneously. As the adjustments related to the *Client Flag* can be simply realized by the global controller (*i.e.*, installing the corresponding table entries in the related PDP switches), they are hitless to in-service SFCs and thus accomplish stateful SFC deployment in runtime.

B. Dynamic SFC Deployment and Traffic Routing

In order to utilize the limited hardware resources in PDP switches efficiently for stateful SFC deployment, we model the network topology as $G(\mathcal{V},\mathcal{E})$, where $\mathcal{V}$ and $\mathcal{E}$ are the sets of PDP switches and links in the network, respectively. Therefore, there are $|\mathcal{V}|$ switch pipelines in the data plane, *i.e.*, $|\mathcal{V}|$ P4 programs can be installed. S_v is the superset stateful SFC that is deployed on a switch $v \in \mathcal{V}$. In the network, N types of stateful vNFs $\{V_1,\cdots,V_N\}$ are supported, *i.e.*, each stateful SFC S_v consists of part/all of the N types of vNFs. To benefit both the SP and clients simultaneously, we set the optimization objective is to provision as many stateful SFC requests from clients as possible, while hoping that the average end-to-end (E2E) latency of all the provisioned SFCs is also minimized. Deploying the longest possible superset SFC on a PDP switch can potentially increase the number of SFCs that can be deployed on the switch. If the vNFs in an SFC cannot be found on a single PDP switch, we have to make the traffic of the SFC to go across switches, but this will increase the E2E latency since the transmission latency incurred across switches is much longer than the processing latency in a pipeline.

Algorithm 2: Dynamic Stateful SFC Deployment

1 $k=0$, $m=\lceil\frac{|\mathcal{V}|}{2}\rceil$;
2 merge all the N types of vNFs to get a superset stateful SFC $\hat{S}$;
3 **if** *$\hat{S}$ can be accommodated in a PDP switch* **then**
4 select m switches from $\mathcal{V}$ to deploy $\hat{S}$ on;
5 **else**
6 find m combinations of vNFs in $\{V_1,\cdots,V_N\}$, which all contain the most possible vNFs that can be accommodated in a PDP switch and in all cover all the N types of vNFs;
7 merge the vNF combinations to get m superset stateful SFCs and select m switches from $\mathcal{V}$ for 1-on-1 deployment;
8 **end**
9 **while** *st-SFC is operational* **do**
10 **if** $k=|V|-m$ **then**
11 **continue**;
12 **end**
13 **if** *the number of in-service flows exceeds $\alpha\cdot\hat{F}_1$* **then**
14 sort vNFs in descending order of their popularity;
15 $\tilde{S}=\emptyset$;
16 **for** *each vNF $V_i\in\{V_1,\cdots,V_N\}$ in sorted order* **do**
17 **if** *$\{\tilde{S},V_i\}$ can be deployed on a PDP switch* **then**
18 insert V_i in $\tilde{S}$;
19 **else**
20 **break**;
21 **end**
22 **end**
23 merge vNFs in $\tilde{S}$ to get a superset stateful SFC;
24 select a PDP switch without SFC to deploy $\tilde{S}$;
25 $k=k+1$;
26 **if** $k=|V|-m$ **then**
27 **continue**;
28 **end**
29 **end**
30 **if** *the number of in-service flows through the pipeline on v exceeds $\beta\cdot\hat{F}_2$* **then**
31 select a PDP switch without SFC to deploy the superset SFC S_v on v;
32 $k=k+1$;
33 **end**
34 **end**

Algorithm 2 shows our heuristic for dynamic stateful SFC deployment. *Line* 1 is for the initialization, where k is the

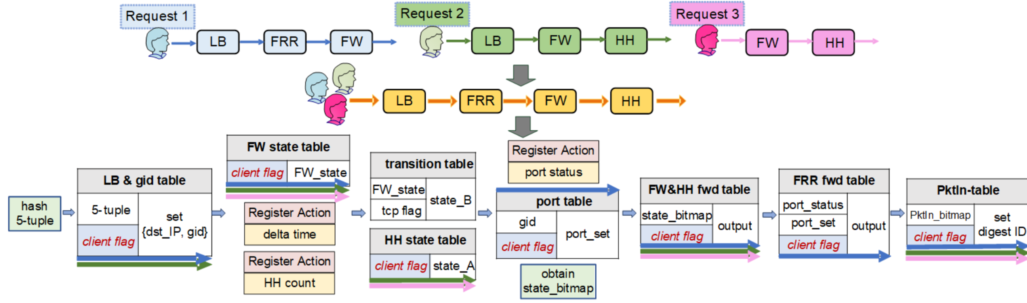


Fig. 7. Example on runtime stateful SFC deployment.

counter for the switches on which superset stateful SFCs have been deployed, and m is the number of switches considered in the initial SFC deployment (*Lines* 3-8). Here, we empirically set $m = \lceil \frac{|V|}{2} \rceil$, and its value can be adjusted according to the characteristics of actual NFV environments. Then, we merge all the N types of vNFs supported in the st-SFC system to get a superset stateful SFC $\hat{S}$ (*Line* 2). If $\hat{S}$ can be accommodated in a PDP switch, we select m switches to deploy it on (*Lines* 3-4). Otherwise, we find m longest sequences of vNFs that each can be accommodated in a PDP switch and in all cover all the N types of vNFs, merge them to get m stateful SFCs, and deploy the SFCs in m switches one by one (*Lines* 5-7).

Algorithm 3: Routing Strategy for New SFC Request

```
1  Ṽ = ∅;
2  for each switch v ∈ V do
3      if the requested SFC S_r ⊆ S_v then
4          add v to Ṽ;
5      end
6  end
7  if Ṽ ≠ ∅ then
8      for each switch v ∈ Ṽ do
9          w_v = N + 1 − len(S_v);
10     end
11     randomly select a switch v* ∈ Ṽ based on {w_v, v ∈ Ṽ};
12     Ṽ = {v*};
13 else
14     sort switches v ∈ V in descending order of the lengths of
       superset stateful SFCs deployed on them;
15     for each switch v ∈ V in sorted order do
16         S'_r = S_r;
17         for each vNF V_i^r ∈ S_r in sequence do
18             if V_i^r ∈ S_v then
19                 S'_r = S'_r \ V_i^r;
20                 add v to Ṽ if v is not already there;
21             else
22                 break;
23             end
24         end
25         if S'_r ≠ ∅ then
26             S_r = S'_r;
27         else
28             break;
29         end
30     end
31 end
32 set up paths to route the request's traffic through switches in Ṽ;
```

The while-loop of *Lines* 9-34 is for runtime operations. First, if all the switches have been deployed with superset SFCs, no operation will be conducted (*Lines* 10-12). Then, we check whether the number of in-service flows in the network exceeds the threshold $\alpha \cdot \hat{F}_1$, where $\alpha \in (0, 1)$ is a ratio and $\hat{F}_1$ is the estimated maximum number of in-service flows that can be provisioned in the network. If yes, we sort the vNFs in descending order of their popularity, greedily merge the most popular vNFs to get the longest superset SFC that can be accommodated in a PDP switch, and deploy the SFC on a switch that does not currently carry a superset SFC (*Lines* 13-29). Next, we check whether the number of in-service flows through each switch v exceeds the threshold $\beta \cdot \hat{F}_2$, where $\beta \in (0, 1)$ is also a ratio and $\hat{F}_2$ is the estimated maximum number of in-service flows that can be carried by the switch. If yes, we select a PDP switch without any superset SFC and duplicate the superset SFC on v onto it (*Lines* 30-33).

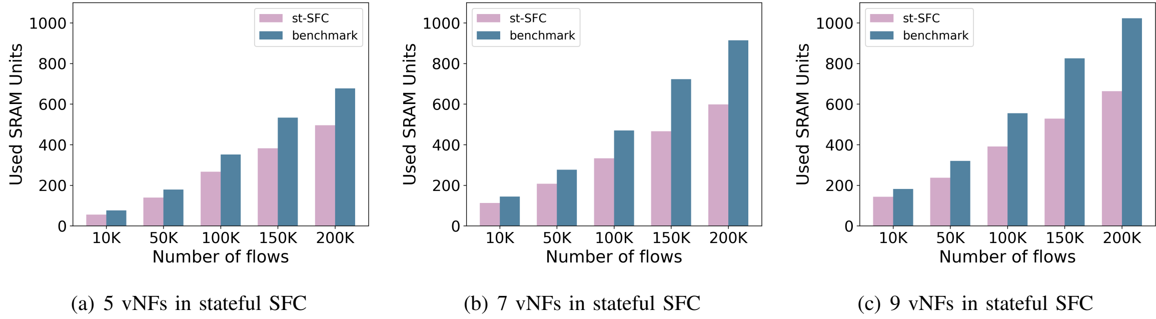
Algorithm 3 explains how to serve a SFC request S_r with our st-SFC system in runtime. *Line* 1 initializes the set of the switches to serve the request as $V = \emptyset$. Then, we check each switch v in the network to see whether the superset SFC deployed on it can cover the requested SFC S_r in whole, and if yes, we add v in $\tilde{V}$ (*Lines* 2-6). Next, if $\tilde{V}$ is not empty, we assign a weight w_v to each switch in it according to the length of the superset SFC on the switch, *i.e.*, the weight w_v decreases with the length of S_r (*Lines* 7-10). *Lines* 11-12 then select a switch v^* from $\tilde{V}$ to provision S_r, with the weighted random selection (*i.e.*, the probability that a switch v is chosen is proportional to its weight w_v). Otherwise, if no switch in V has a superset SFC that can cover S_r in whole, we sort all the switches in descending order of the lengths of superset SFCs deployed on them (*Line* 14). Then, the for-loop of *Lines* 15-30 checks each switch v in the sorted order to see whether vNF(s) in S_r can be served in sequence with the superset SFC on the switch, and if yes, we add v in $\tilde{V}$. Finally, *Line* 32 sets up the paths to route the traffic of S_r through the switches in $\tilde{V}$, and accomplishes the provisioning of the SFC request.

V. Performance Evaluations

In this section, we discuss the implementation of prototype st-SFC and performance evaluations with experiments.

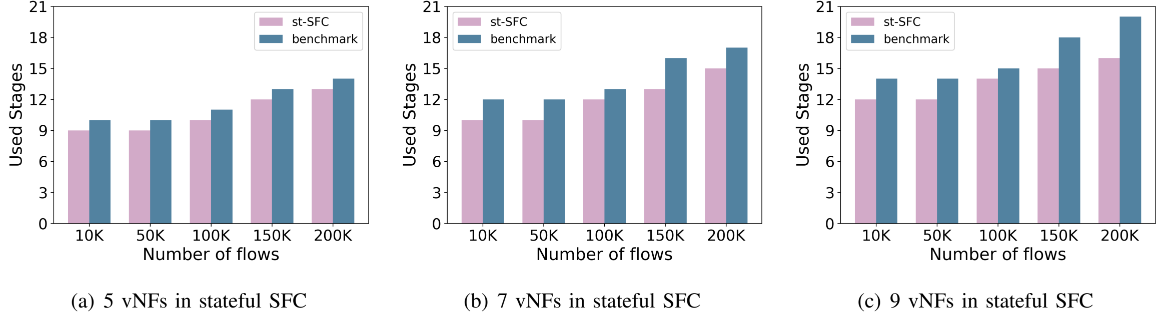
A. Implementation and Experimental Setup

We prototype st-SFC with the P4-based PDP switches that equip Tofino ASICs. On each PDP switch, we program



(a) 5 vNFs in stateful SFC (b) 7 vNFs in stateful SFC (c) 9 vNFs in stateful SFC

Fig. 8. Results on SRAM usage due to stateful SFC deployment.



(a) 5 vNFs in stateful SFC (b) 7 vNFs in stateful SFC (c) 9 vNFs in stateful SFC

Fig. 9. Results on stage usage due to stateful SFC deployment.

TABLE I
INFORMATION ABOUT VNFS SUPPORTED IN OUR ST-SFC PROTOTYPE

Stateful vNFs	Category	Pipeline Stages
Stateful load balancer	3	4
Stateful NAT	3	4
Fast re-routing	3	5
TCP firewall	2	7
Heavy hitter	1	4
DNS reflection attack protection	3	5
SYN flood detection	1	6
DNS request detection	1	6
Super spread detection	1	6
Flow size detection	1	3

a unified Python-based Barefoot Runtime control plane to manage the switch's pipeline that is programmed with the P4 language. We develop the st-SFC prototype to support $N = 10$ types of stateful vNFs, which are all typical ones in the literature. Table I shows the stateful vNFs, their categories according to the classification method discussed in Section III-B, and the number of stages that they will occupy in a switch pipeline. In addition to the PDP switches, we also implement a home-made global controller based on Python for network-wide management (as shown in Fig. 6), *i.e.*, running *Algorithms* 2 and 3 to deploy stateful superset SFCs on PDP switches and provision SFC requests with them in runtime.

In the experiments, we evaluate our st-SFC system to 1) verify its advantages in saving hardware resources and reducing average E2E latency of stateful SFCs, and 2) demonstrate the functionality and performance of stateful SFC deployment with it. We deploy the global controller on an Ubuntu 20.04 Linux server with 40 Intel Xeon Silver 4210 CPUs and 64 GB of memory. The network testbed includes 4 PDP switches, which are all connected to the ingress switch with the topology in Fig. 6. We set $\alpha = 0.6$ and $\beta = 0.9$ in the experiments.

B. Performance of Static Stateful SFC Deployment

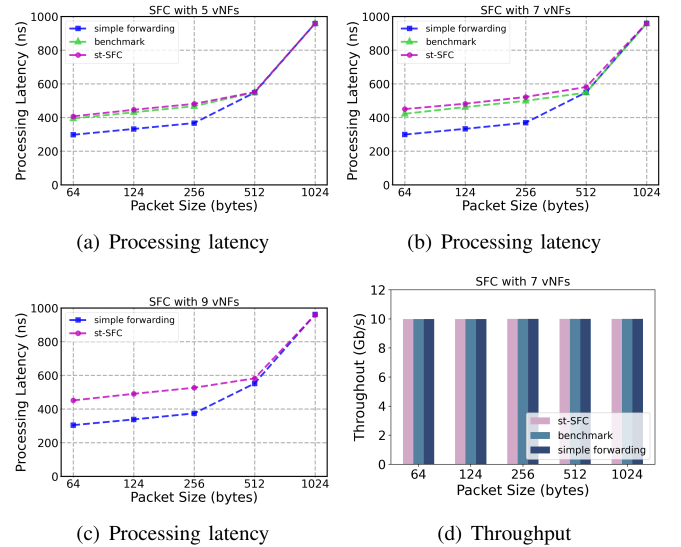
We first evaluate the performance of st-SFC for static stateful SFC deployment (*i.e.*, the one-time initial deployment of stateful SFCs). We also implement a benchmark that only considers the exact merge (*e.g.*, the vNF merging approach used in [35, 36]) and does not leverage the *PktIn-Table* to optimize the interactions with the control plane. We first randomly select $\{5, 7, 9\}$ vNFs from the 10 supported ones to form a stateful SFC and deploy the SFC in PDP switches. The tool of Intel P4 Insight [45] is used to get the usages of SRAM units and stages[3] in PDP switches.

Figs. 8 and 9 respectively show the total numbers of SRAM units and stages used in each experiment to support $[10000, 200000]$ client flows. Here, in each experiment, we first determine the size of each MAT (in table entries) and number of registers based on the number of flows to support, and then write the P4 program for a specific SFC accordingly. Next, we process the P4 program with our st-SFC and the benchmark for table merging, send their outputs to P4 compiler, and get the resource usages with P4 Insight. We can see that st-SFC uses fewer SRAM units and stages than the benchmark, and the resource saving achieved by it increases with the length of a stateful SFC and the number of flows to serve. For example, for the cases with the SFC that consists of 7 vNFs, the SRAM saving achieved by st-SFC over the

[3] A P4 program for stateful SFC can use 12 stages at most in the pipeline of a PDP switch, and the maximum number of SRAM units in a stage is 80.

TABLE II
COMPARISON OF USAGES OF DIFFERENT TYPES OF HARDWARE RESOURCES IN PDP SWITCH

Type of Hardware Resource	5 vNFs		7 vNFs		9 vNFs	
	st-SFC	benchmark	st-SFC	benchmark	st-SFC	benchmark
SRAM Units	267	351	333	470	392	555
Exact Match Input Xbar	109	137	132	166	156	194
Gateway	12	12	12	12	13	13
Hash Bit	460	669	564	819	750	1,007
TCAM Units	17	17	17	17	17	17
Stash	9	20	11	25	14	30
Logical Table ID	25	27	31	33	36	38
Stages	10	11	12	13	14	15
Number of PDP switches	1	1	1	2	2	2



(a) Processing latency

(b) Processing latency

(c) Processing latency

(d) Throughput

Fig. 10. Results on processing latency and throughput of stateful SFCs.

benchmark increases from 21.6% to 34.6% when the number of flows changes from $10,000$ to $200,000$. This is because compared with the benchmark, st-SFC facilitates two more merging strategies (*i.e.*, same action merge and same match merge) to further save SRAM units and stages, and as the resource saving achieved by them is effective for each vNF and each flow, st-SFC saves more resources when the length of a stateful SFC and the number of flows to serve increase.

As for stage usage, Fig. 9 indicates that with a single PDP switch, st-SFC can provision $100,000$ flows in the SFC with 7 vNFs or $50,000$ flows in the SFC with 9 vNFs (*i.e.*, the number of used stages is 12 or less). However, the benchmark can only serve $50,000$ flows for the SFC with 7 vNFs, and it cannot even deploy the SFC with 9 vNFs in one PDP switch. This is because the usage of stages depends on the sizes and dependencies of MATs (*e.g.*, if there are dependencies between two MATs, they can only be placed in two stages), which can both be effectively reduced by the table merging in st-SFC.

In addition to SRAM and stages, there are other types of hardware resources in the pipeline of a PDP switch. For instance, there are also TCAM units, the Exact Match Input Xbars that are used to select data from packet header vectors for field match and hash computation, the Gateways that are for implementing "if" conditional statements, the Hash Bits that are the generators for hash operations, Stashes that provide temporary storage for realizing table entry modifications, and the Logical Table IDs that are used to serialize the processing in MATs. Therefore, for a more comprehensive comparison, Table II lists the usages of various types of resources when there are $100,000$ flows to serve with three types of stateful SFCs. It can be seen that st-SFC saves most types of hardware resource. As the merging strategies of st-SFC cannot reduce conditional statements in a stateful SFC, the Gateway usages of st-SFC and the benchmark are the same. Meanwhile, the table merging considered in this work does not affect TCAM, and thus the TCAM usages in Table II stay unchanged.

Next, we test the throughput and processing latency of a deployed stateful SFC with 7 vNFs, where the number of flows is set as $50,000$, because according to the results in Fig. 9(b), this scenario is the largest scale that can be handled by the benchmark with one PDP switch. This time, we introduce another benchmark, namely, simple forwarding, which only lets each stage match to the destination IP address, and the number of stages is made the same as that of st-SFC. Simple forwarding is introduced to check whether the stateful SFCs deployed by st-SFC will cause abnormally long processing latency, when being compared to the simplest packet forwarding scheme in a PDP switch. Figs. 10(a)-10(c) shows that the processing latency of SFC deployed by st-SFC is only slightly longer than that of SFC deployed by the benchmark when the packet size is less than 512 bytes, while their latencies are almost the same at larger packet sizes (*i.e.*, $\{512,\ 1024\}$ bytes). This is due to the unavoidable increase of logic unit complexity caused by the vNF merging in st-SFC.

As expected, simple forwarding achieves shorter processing latency than st-SFC and the benchmark when the packet size is less than 512 bytes. Note that, packet receiving, parsing and processing all contribute to the processing latency of a packet. For a smaller packet size, the pipeline needs to process more packets at the same data-rate, and thus the complex processing logic provided by st-SFC can become the bottleneck to cause longer processing latencies than those from simple forwarding for small-sized packets. On the other hand, for a relatively long packet, receiving and parsing it can be time-consuming and contribute to the majority of its processing latency, which explains why the processing latencies of the three schemes are similar at packet sizes of $\{512, 1024\}$ bytes.

As the benchmark cannot deploy an SFC with 9 vNFs in

one PDP switch, Fig. 10(c) only shows the results of st-SFC and simple forwarding, where the gaps between their latencies are similar as those in Figs. 10(a) and 10(b), verifying that st-SFC scales well. Specifically, as the packet size increases, the processing latency of the SFC deployed by st-SFC increases from 451 ns to 961 ns, while that of simple forwarding increases from 304 ns to 960 ns. The packet processing performance of the stateful SFC deployed by st-SFC can be further confirmed with the throughput results in Fig. 10(d), as the line-rate of 10 Gbps can be achieved for all the packet sizes. Note that, the experiments use 10 Gbps as the line-rate for the reason that this is the highest data-rate that we can achieve for all the packet sizes with our software packet generator based on data plane development kit (DPDK) [46].

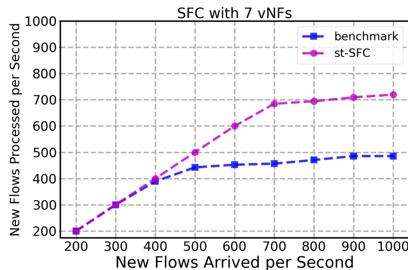


Fig. 11. New flow processing capacity for verifying benefit of *PktIn-Table*.

Finally, we conduct experiments to verify the effectiveness of *PktIn-Table*. We still consider the case of SFC with 7 vNFs, and select two of the vNFs as TCP firewall and DNS reflection attack protection, which belong to the second and third category in Section III-B, respectively, and both need to interact with the control plane for state updates. The experiments generate new flows with a speed of $[200,\ 1000]$ flows per second, and check the capacity of the stateful SFC when interactions with the control plane have to be invoked. As shown in Fig. 11, the stateful SFC deployed by st-SFC can catch up with the arrival rate of new flows until there are more than 700 flows being generated per second, while the one deployed by the benchmark cannot process all the new flows when the arrival rate is higher than 300 new flows per second. This is because without *PktIn-Table*, the benchmark always uploads the state information of TCP firewall and DNS reflection attack protection together, bringing unnecessary message parsing overheads to the control plane. Therefore, by adopting *PktIn-Table*, st-SFC more than doubles the capacity of the SFC on processing new flows, making interactions with the control plane much more efficient.

C. Performance of Dynamic Stateful SFC Deployment

Next, we evaluate our proposed algorithms for dynamic stateful SFC deployment (*i.e.*, *Algorithms* 2 and 3). This time, in order to focus on the performance of the algorithms, we make the benchmark also use the mechanism of st-SFC to merge vNFs to superset SFCs and deploy them on PDP switches. The main difference between it and st-SFC is that it only uses the *Lines* 2-8 in *Algorithm* 2 to find the longest possible superset SFC(s) and deploy them on all the PDP switches in the initial deployment but does not consider the *Lines* 9-34 in the algorithm for adaptive and dynamic superset stateful SFC deployment. Meanwhile, the benchmark also uses *Algorithm* 3) to provision SFC requests in runtime.

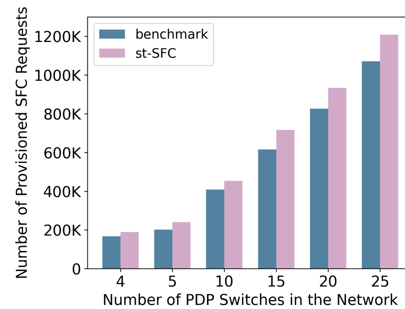
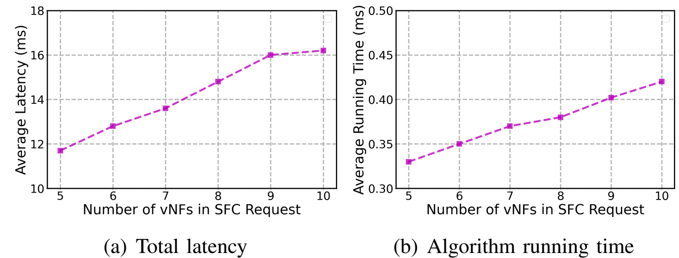


Fig. 12. Results on provisioned SFC requests in dynamic SFC deployment.

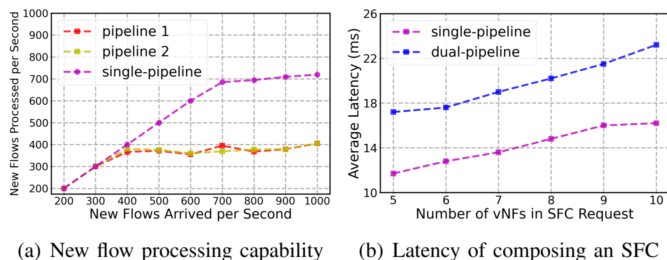


(a) Total latency (b) Algorithm running time

Fig. 13. Results on latency of runtime SFC request provisioning.

The experiments randomly select $[5, 10]$ vNFs from the 10 supported types to generate SFC requests dynamically. Fig. 12 shows the results on the number of SFC requests that can be successfully provisioned in networks with different sizes. Note that, due to our limited budget, only the results for the network with 4 PDP switches are experimental, while the remaining ones are obtained with the simulations based on exactly same resource constraints for PDP switches. Specifically, each simulation considers a network that contains $[5, 25]$ PDP switches, generates SFC requests dynamically in the same way used by the experiments, still uses st-SFC and the benchmark to deploy stateful SFCs in the network, and finally obtains the number of SFC requests that can be provisioned in each case. We observe that related to the benchmark, st-SFC can increase the number of provisioned flows by $[11\%, 19\%]$ with an average ratio of 14.2%, which verifies the effectiveness of *Algorithm* 2. This is because *Algorithm* 2 properly considers the popularity of vNFs to compose and deploy superset stateful SFCs incrementally in runtime, and thus can satisfy the SFC requests of more flows.

Finally, we measure the latency of SFC request provisioning in runtime. Specifically, the latency is the time period from when a new request arrives at the global controller to when the request SFC becomes operational in the network. Fig. 13(a) shows the experimental results on the average latency, which indicates that when the number of vNFs in a request SFC increases from 5 to 10, the latency changes from 11.7 ms to 16.2 ms. We plot the running time of the algorithms in Fig. 13(b), which is within $[0.33, 0.46]$ ms. This suggests that our proposed algorithms are time-efficient and their running time does not make major contribution to the latency. The



(a) New flow processing capability (b) Latency of composing an SFC

Fig. 14. Performance of st-SFC on single- and dual-pipeline Tofino switches.

major part of the latency comes from the interaction between the local control plane and data plane of a PDP switch, for updating table entries for deploying SFC requests.

D. Evaluation of st-SFC in Dual-Pipeline Scenario

Finally, we implement and evaluate st-SFC in a Tofino switch that can support two pipelines simultaneously, *i.e.*, the switch can carry two superset SFCs to process packets in parallel. When the two pipelines are processing stateful vNFs, there will be more interactions with the control plane, which affects the speed of state information update in each pipeline [47]. Fig. 14(a) shows the capabilities of a Tofino switch in single-pipeline and dual-pipeline scenarios on processing new flows for superset SFC(s) deployed with st-SFC. It can be seen that when both pipelines in the switch are activated, the capability of each pipeline on processing new flows is less than 400 flows/sec, which is less than that in the single-pipeline scenario ($\sim$700 flows/sec). We plot the latency of composing an SFC with the deployed superset SFC(s) (*i.e.*, the time used for installing related flow entries in runtime) in Fig. 14(b). We observe that the latency of the dual-pipeline scenario is about $[5.8, 7]$ ms longer than that of the single-pipeline scenario. Fig. 14 suggests that there is a tradeoff between the number of superset SFCs to deploy in a Tofino switch and the state information update speed of each of them.

VI. Conclusion

In this paper, we proposed st-SFC, which is an NFV system that can deploy stateful SFCs in P4-based PDP switches to not only utilize the hardware resources on switches efficiently but also minimize the overhead of interactions between control and data planes. We first abstracted each stateful vNF as a state machine. Then, we proposed a stateful SFC building algorithm to merge the state machines of vNFs of different types for reducing redundant resource usages in PDP switches. Meanwhile, for the vNFs whose operations involve interactions with the control plane, we designed a *PktIn-Table* to reduce the resource usage in PDP switches and the interaction latency. Next, we developed an SFC deployment algorithm that realizes stateful SFCs on PDP switches on demand, aiming to optimize the resource usages across all the switches in runtime. Finally, we prototyped st-SFC with P4-based PDP switches and demonstrated its performance experimentally. Our experimental results verified that st-SFC can provision dynamic SFC requests with high resource-efficiency, and outperform benchmarks in terms of hardware resource utilization and interaction delay between the control and data planes.

Acknowledgments

This work was supported by the NSFC project 62371432.

References

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

[2] 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.

[3] W. Lu *et al.*, “AI-assisted knowledge-defined network orchestration for energy-efficient data center networks,” *IEEE Commun. Mag.*, vol. 58, pp. 86–92, Jan. 2020.

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

[5] W. Fang *et al.*, “Joint spectrum and IT resource allocation for efficient vNF service chaining in inter-datacenter elastic optical networks,” *IEEE Commun. Lett.*, vol. 20, pp. 1539–1542, Aug. 2016.

[6] K. Wu, P. Lu, and Z. Zhu, “Distributed online scheduling and routing of multicast-oriented tasks for profit-driven cloud computing,” *IEEE Commun. Lett.*, vol. 20, pp. 684–687, Apr. 2016.

[7] L. Gong, Y. Wen, Z. Zhu, and T. Lee, “Toward profit-seeking virtual network embedding algorithm via global resource capacity,” in *Proc. of INFOCOM 2014*, pp. 1–9, Apr. 2014.

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

[9] L. Dong, N. L. S. da Fonseca, and Z. Zhu, “Application-driven provisioning of service function chains over heterogeneous NFV platforms,” *IEEE Trans. Netw. Serv. Manag.*, vol. 18, pp. 3037–3048, Sept. 2021.

[10] Tofino switch. [Online]. Available: https://www.barefootnetworks.com/products/brief-tofino/.

[11] P. Bosshart *et al.*, “P4: Programming protocol-independent packet processors,” *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 87–95, Jul. 2014.

[12] Z. Pan *et al.*, “Advanced optical-label routing system supporting multicast, optical TTL, and multimedia applications,” *J. Lightw. Technol.*, vol. 23, pp. 3270–3281, Oct. 2005.

[13] Z. Zhu *et al.*, “Jitter and amplitude noise accumulations in cascaded all-optical regenerators,” *J. Lightw. Technol.*, vol. 26, pp. 1640–1652, Jun. 2008.

[14] 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.

[15] Y. Yin *et al.*, “Spectral and spatial 2D fragmentation-aware routing and spectrum assignment algorithms in elastic optical networks,” *J. Opt. Commun. Netw.*, vol. 5, pp. A100–A106, Oct. 2013.

[16] 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.

[17] V. Sivaraman *et al.*, “Heavy-hitter detection entirely in the data plane,” in *Proc. of SOSR 2017*, pp. 165–176, Apr. 2017.

[18] 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.

[19] M. Chiesa *et al.*, “Fast ReRoute on programmable switches,” *IEEE/ACM Trans. Netw.*, vol. 29, pp. 637–650, Apr. 2021.

[20] 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, Sept. 2021.

[21] L. Teng, C. Hung, and C. Wen, “P4SF: A high-performance stateful firewall on commodity P4-programmable switch,” in *Proc. of NOMS 2022*, pp. 1–5, Apr. 2022.

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

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

[24] K. Qian *et al.*, “FlexGate: High-performance heterogeneous gateway in data centers,” in *Proc. of APNet 2019*, pp. 36–42, Aug. 2019.

[25] D. Hancock and J. Van der Merwe, “Hyper4: Using P4 to virtualize the programmable data plane,” in *Proc. of CoNEXT 2016*, pp. 35–49, Dec. 2016.

[26] 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.
[27] P. Zheng, T. Benson, and C. Hu, “P4visor: Lightweight virtualization and composition primitives for building and testing modular programs,” in *Proc. of CoNEXT 2018*, pp. 98–111, Dec. 2018.
[28] H. Soni, T. Turletti, and W. Dabbous, “P4bricks: Enabling multiprocessing using linker-based network data plane architecture,” Feb. 2018. [Online]. Available: https://inria.hal.science/hal-01632431.
[29] X. Chen *et al.*, “Speed: Resource-efficient and high-performance deployment for data plane programs,” in *Proc. of ICNP 2020*, pp. 1–12, Oct. 2020.
[30] G. Bianchi *et al.*, “Open packet processor: a programmable architecture for wire speed platform-independent stateful in-network processing,” *arXiv preprint arXiv:1605.01977*, 2016.
[31] A. Sivaraman *et al.*, “Packet transactions: High-level programming for line-rate switches,” in *Proc. of ACM SIGBOMM 2016*, pp. 15–28, Aug. 2016.
[32] S. Pontarelli *et al.*, “Flowblaze: Stateful packet processing in hardware,” in *Proc. of NSDI 2019*, pp. 531–547, Feb. 2019.
[33] N. Gebara *et al.*, “Challenging the stateless quo of programmable switches,” in *Proc. of HotNets 2020*, pp. 153–159, Nov. 2020.
[34] V. Shrivastav, “Stateful multi-pipelined programmable switches,” in *Proc. of ACM SIGCOMM 2022*, pp. 663–676, Aug. 2022.
[35] X. Zhang, L. Cui, F. Tso, and W. Jia, “Compiling service function chains via fine-grained composition in the programmable data plane,” *IEEE Trans. Serv. Comput.*, pp. 1–13, Feb. 2023.
[36] X. Chen *et al.*, “P4SC: Towards high-performance service function chain implementation on the P4-capable device,” in *Proc. of IM 2019*, pp. 1–9, Apr. 2019.
[37] C. Sun *et al.*, “Toward a stateful data plane in software-defined networking,” *IEEE/ACM Trans. Netw.*, vol. 25, pp. 3294–3308, Dec. 2017.
[38] D. Wu *et al.*, “Accelerated service chaining on a single switch ASIC,” in *Proc. of HotNets 2019*, pp. 141–149, Nov. 2019.
[39] J. Lee, H. Ko, H. Lee, and S. Pack, “Flow-aware service function embedding algorithm in programmable data plane,” *IEEE Access*, vol. 9, pp. 6113–6121, Dec. 2020.
[40] M. He *et al.*, “P4NFV: An NFV architecture with flexible data plane reconfiguration,” in *Proc. of CNSM 2018*, pp. 90–98, Nov. 2018.
[41] T. Osiski, H. Tarasiuk, L. Rajewski, and E. Kowalczyk, “DPPx: A P4-based data plane programmability and exposure framework to enhance NFV services,” in *Proc. of NetSoft 2019*, pp. 296–300, Jun. 2019.
[42] Y. Zhou *et al.*, “Flexmesh: Flexibly chaining network functions on programmable data planes at runtime,” in *Proc. of Networking 2020*, pp. 73–81, Jun. 2020.
[43] D. Moro, G. Verticale, and A. Capone, “Network function decomposition and offloading on heterogeneous networks with programmable data planes,” *IEEE Open J. Com. Soc.*, vol. 2, pp. 1874–1885, Aug. 2021.
[44] J. Ma, S. Xie, and J. Zhao, “Flexible offloading of service function chains to programmable switches,” *IEEE Trans. Serv. Comput.*, vol. 16, pp. 1198–1211, Mar. 2022.
[45] Intel p4 insight. [Online]. Available: https://www.intel.cn/content/www/cn/zh/products/details/network-io/intelligent-fabric-processors/p4-insight.html.
[46] Dpdk. [Online]. Available: https://www.dpdk.org.
[47] M. Chiesa and F. Verdi, “Network monitoring on multi-pipe switches,” *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 7, pp. 1–31, Mar. 2023.