FlexNF: Flexible Network Function Orchestration for Scalable On-Path Service Chain Serving

Jingyu Xiao, Xudong Zuo, *Member, IEEE*, Qing Li, *Senior Member, IEEE*, Dan Zhao, Hanyu Zhao, Yong Jiang, *Member, IEEE*, Jiyong Sun, Bin Chen, Yong Liang, and Jie Li, *Member, IEEE*

***Abstract*— Programmable Data Plane (PDP) has been leveraged to offload Network Functions (NFs). Due to its high processing capability, the PDP improves the performance of NFs by more than one order of magnitude. However, the coarse-grained NF orchestration on the PDP makes it hard to fulfill the dynamic service chain demands and unreasonable network function deployment causes long end-to-end delays. In this paper, we propose the Flexible Network Function (FlexNF) deployment on the PDP. First, we design an NF Selection Framework, leveraging the service selection label and re-entering operations for flexible NF orchestration. Second, to support runtime NF reconfiguration to meet the dynamic flow demands, we propose the Per-Flow On-Demand servicing mechanism, where one Match-Action Table with multiple mixed NFs works as different NFs for different flows. Third, to ensure the QoS of flows, on the one hand, we design an SP-aware NF Placement Algorithm to find a near-optimal placement solution that accommodates peak traffic volume while minimizing the overall routing path lengths of all the requests, on the other hand, we design a Two-Stage Service Path Construction Algorithm to provide on-path service while considering load balancing. We implement 15 types of network functions on the P4 switch, based on which we construct the comprehensive experiments. FlexNF reduces the traffic delay by 42.6% while increasing the service chain acceptance rate by five times compared with current solutions. Besides, when switching functions, the FlexNF improves the throughput by 2.04Gbps and reduces the packet loss by 8.269% compared with current solutions.**

***Index Terms*— Service chain, network function, programmable data plane.**

Manuscript received 15 April 2023; revised 7 September 2023; accepted 13 November 2023; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor K. Chen. This work was supported in part by the National Key Research and Development Program of China under Grant 2022YFB3105000, in part by the Major Key Project of Peng Cheng Laboratory (PCL) under Grant PCL2023A06, and in part by the Shenzhen Key Laboratory of Software Defined Networking under Grant ZDSYS20140509172959989. *(Jingyu Xiao and Xudong Zuo contributed equally to this work.) (Corresponding author: Qing Li.)*

Jingyu Xiao, Xudong Zuo, Hanyu Zhao, and Yong Jiang are with the Tsinghua Shenzhen International Graduate School, Shenzhen 518055, China, and also with the Peng Cheng Laboratory, Shenzhen 518066, China (e-mail: jy-xiao21@mails.tsinghua.edu.cn; zuoxd20@mails.tsinghua.edu.cn; zhao-hy18@mails.tsinghua.edu.cn; jiangy@sz.tsinghua.edu.cn).

Qing Li and Dan Zhao are with the Peng Cheng Laboratory, Shenzhen 518066, China (e-mail: liq@pcl.ac.cn; zhaod01@pcl.ac.cn).

Jiyong Sun, Bin Chen, Yong Liang, and Jie Li are with China Mobile Communication Group Guangdong Company Ltd., Guangdong 510030, China (e-mail: sunjiyong@gd.chinamobile.com; chenbin3@gd.chinamobile.com; liangyong8@gd.chinamobile.com; lijie2@gd.chinamobile.com).

Digital Object Identifier 10.1109/TNET.2023.3334237

I. INTRODUCTION

NETWORK Functions (NFs), such as Load Balancers (LBs), Firewalls, Failure Detectors, and Network Address Translators (NATs), are widely deployed in data centers [1], [2], mobile networks [3], and cloud environments [4], etc. NFs have a long history of being implemented as expensive and proprietary hardware middleboxes, with problems of high management complexity and operational cost [5]. Therefore, Network Function Virtualization (NFV) [6] has been proposed to revolutionize the way that traditional networks are managed and operated. By decoupling NFs from the underlying hardware and running them as software-based virtual devices in the cloud environments, NFV offers great flexibility [7] and cost-effectiveness for provisioning network services. However, the benefits of NFV are accompanied by performance penalties. The software-based NFs could introduce significant performance overheads, resulting in prolonged packet processing delay and compromised packet processing throughput [7].

To accelerate NFV service, significant research efforts [8], [9], [10], [11], [12] have been dedicated to hardware-offloading of network functions (NFs) using general hardware platforms, which preserves the openness of NFs. With the advent of Programmable Data Plane (PDP) [13], [14], data-plane offloading is envisioned as an effective approach for NFV acceleration [15]. The PDP provides high-speed processing capability that can suffice the line rate (e.g., the programmable Tofino switch supports a port bandwidth of 12.8Tbps [16]). Moreover, PDP can flexibly accommodate advanced network functions and applications through customized packet processing logics. By installing customized NF programs on the PDP, existing works [8], [9], [17], [18], [19], [20], achieve full line-rate processing, with no performance loss compared to the forwarding-only PDP. However, there are still three challenges when offloading NFs to the PDP.

First, **each programmable switch only provides fixedly deployed service of network functions installed in a predetermined order.** This inflexibility causes a dilemma when offloading NFs to the PDP. On the one hand, we can offload a single NF instance to a programmable switch and enable service chaining by routing network flows across different switches. But such an exclusive deployment forces flows to go through more hops, leading to the increased end-to-end delay. On the other hand, we can deploy a sub-chain, consolidating

multiple NFs, on a switch so that service chaining can potentially be fulfilled within fewer switches, as will be detailed in Section II. However, this approach limits the scalability of the composable service chains on the PDP, making it infeasible for dynamic NFV management.

Second, **changing the programs installed on switches would cause service disruption**. As the demands of network traffic are constantly changing, the network functions installed on the switches may need to be changed frequently. However, the programmable switches do not support hot swapping of the deployed NFs. Changing the packet processing logic of a switch requires stopping the device and installing a newly compiled configuration, costing more than 10 seconds, which significantly affects the passing traffic on the switch.

Third, **enabling on-path serving is challenging.** Usually, the network uses the shortest path to forward traffic, which we call on-path servicing. However, due to unreasonable network function deployment, the traffic would endure an extra forwarding delay than the shortest path due to the off-path service chaining, resulting in poor routing performance. As such, it is critical to carefully plan the placement of NFs and construct the forward path of service chain demands to reduce the end-to-end delay experienced by traffic.

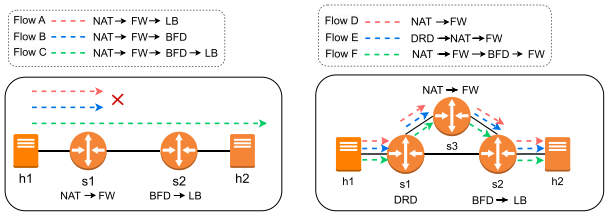
In this paper, we propose FlexNF,[1] a flexible and efficient PDP-based NFV framework. We conquer the above challenges with the following key ideas:

- We design an **NF Selection Framework (NSF)**, leveraging labels to instruct flows to run or skip NFs, which enables flexible and fine-grained NF orchestration.
- We propose a **Per-Flow On-Demand (PFOD)** servicing mechanism, which deploys several different NFs on the same Match-Action Table (MAT) of the P4 switch. For different flows, the MAT works as different NFs. In this way, an NF service can be enabled or disabled by adjusting the corresponding rules in a hot-swapping way.
- To reduce the off-path service delay, we design an **SP-aware NF Placement Algorithm (SNPA)** and a **Two-Stage Service Path Construction Algorithm (TSPC)**. Specifically, SNPA is executed during the initial deployment stage for the optimal NF placement and TSPC optimizes NF service performance with the shortest response delay based on dynamic network statistics, with load balancing requirements taken into account.

To sum up, in the offline mode, SNPA finds an NF deployment solution that minimizes the total route lengths of all historical requests as much as possible, with memory constraints on all switches along the paths satisfied. Then in the online mode, when flows enter the ingress switch, the NSF will flexibly select or skip some NF instances and the PFOD will dynamically switch NFs based on the demands of flows. Finally, the TSPC will assign service paths to serve the flows while considering both the route length and load balancing.

We implement the prototypes of 15 stateless or stateful NFs on the P4 switch to verify the feasibility of the FlexNF implementation model. To further demonstrate the performance of FlexNF, we construct comprehensive experiments based on the implemented prototypes and the real-world traffic [22]. The experiment results show that: 1) FlexNF improves service chain acceptance rate by five times, while increasing packet forwarding delay by at most 1.7% introduced by the NSF, compared with current solutions; 2) Compared with the normal switch program switching process, under the switching interval of the 60s, the PFOD servicing mechanism improves the throughput by 2.04Gbps and reduces the packet loss rate by 8.269%; 3) SNPA and TSPC can reduce the traffic routing delay by about 42.6%.

[1]This work was presented in part at the 2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS) [21].



(a) Scalability issue. (b) Off-path issue.

Fig. 1. The issue of service chain composition with PDP NFs.

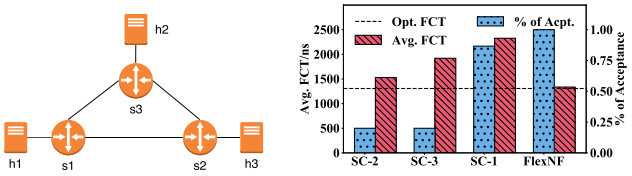
The rest of the paper is organized as follows. We first introduce the motivation in Section II-A and design challenges in Section II-B. The overview of FlexNF is presented in Section III. Then, we elaborate on the design details in Section IV. In Section V, we describe the algorithms for NF placement and orchestration. Section VI details the implementation of FlexNF. In Section VII, the performance of FlexNF is evaluated. Section VIII presents the related works. Finally, we conclude the paper in Section IX.

II. Background and Motivation

A. Issues of Service Chain Composition on the PDP

The PDP-based NF offloading is gradually taken as an effective approach to achieve better performance (i.e., high throughput). To serve the dynamic service chain demands of incoming traffic, it is necessary to effectively orchestrate the NFs on the PDP in real time to form the specific paths. However, the NFs deployed on a single PDP switch must be either completely skipped or triggered in the pre-determined order for the incoming packets. Under such circumstances, a service chain demand has to be fulfilled with the device-level sub-chains (sub-chain for short) as the basic units, causing three practical issues:

- **Scalability issue of Service Chain Composition**. In real networks, there exists a considerable number of dynamically changing service chain demands. The coarse sub-chain deployment granularity limits the number of service chains that a PDP can support. Fig. 1(a) shows a network with two sub-chains (NAT-Firewall (FW) and Big Flow Detector (BFD)-LB) installed. It can accommodate requests of four different service chains: NAT-FW, BFD-LB, NAT-FW-BFD-LB and BFD-LB-NAT-FW. However, NAT-FW-LB and NAT-FW-BFD, cannot be served, even though all the required instances are in fact available in the network.



(a) Topology of the Example (b) FCT and Acceptance Ratio

Fig. 2. Motivating example: performance evaluation on different deployment solutions. SC-n represents installing a sub-chain consisting of n NFs on each switch. FCT denotes flow completion time. Avg.FCT indicates the average value of FCT. Opt.FCT indicates the average FCT when flows go through the shortest path.

- **Network Function Switching Interruption**. A switch has to process different service chain demands of dynamic traffic, which leads to frequent network function switching on the switch. Changing the service of a programmable switch at runtime requires a device interruption of more than 10 seconds due to configuration adjustment, causing severe performance degradation.
- **Off-path Caused Delay**. As each sub-chain is fixedly settled on a specific switch, to traverse the NFs in the service chain order, traffic often has to endure an extra forwarding delay than the shortest path due to the off-path service chaining problem, resulting in poor routing performance. Fig.1(b) shows a network with three sub-chains installed, i.e., DNS Reflection Detection(DRD), NAT-FW and BFD-LB. Suppose there are three requests from h1 to h2 with three different service chains: NAT-FW, DRD-NAT-FW and NAT-FW-BFD-LB. To meet the service chain demands, the network forwards all these requests with the path "s1-s3-s2" rather than the shortest path "s1-s2", causing a longer forwarding delay.

B. Design Challenges

To better understand the PDP-based NF orchestration problem, we use a toy example with three programmable switches interconnected as shown in Fig. 2(a). We implement three types of NFs (Network Address Translation, Firewall and Load Balancer) in the network with different sub-chain deployment granularity. As Fig. 2(b) shows, on the one hand, deploying a sub-chain consisting of more NFs on every switch results in a very **low acceptance ratio** ($\frac{\text{No. of serviced requests}}{\text{No. of requests}}$), due to the limited service chain support in the network, but benefits from smaller forwarding delays. On the other hand, deploying a single NF on every switch causes significant forwarding delay for traversing the required NF nodes in the **off-path way**, though enjoying a high acceptance ratio.

To endow scalability and flexibility to the orchestration of PDP-based NFs, some works [23], [24] attempt to virtualize the PDP by leveraging match-action table entries to emulate the control logic at runtime. However, [23] and [24] overconsume memory resources by 7× and 3.2× than the original Tofino switch [16], respectively.

As shown in Table I, all existing methods can only partially solve the problems of service chain composition on the PDP. That is, none of them can achieve scalable service chain provision, on-path traffic serving and cost-effective memory usage at the same time.

TABLE I
COMPARISON OF THE EXISTING SOLUTIONS

Scheme	Scalability	Flexibility	Cost-effective
NF-Granularity	✓	×	✓
Sub-Chain Granularity	×	×	✓
PDP Virtualization [23], [24]	✓	✓	×
FlexNF	✓	✓	✓

In order to solve the above problems, we design the FlexNF architecture, addressing the following challenges:

- **Selective Serving Mechanism on the PDP**. Fixed-node NF orchestration schemes lead to service chain inscalability and extra routing delay caused by off-path service chaining. To orchestrate the NFs on a programmable switch just like the software NFs on the X86 server, it is necessary to replace the consolidated programs with a Selective Serving Mechanism (SSM) on the PDP device to enable fine-grained NF orchestration. To this end, we design a label-based NF Selection Framework (NSF) to support SSM on the PDP, which supports multiple service chain demands in one single PDP device.
- **Real-time Network Functions Switching on the PDP.** Services offered by a single switch are limited by the pre-installed programs. When the demands of the flows passing through switches change drastically, the installed NF needs to be adjusted in time to avoid detours and service quality degradation. However, reconfiguration of a P4 hardware switch incurs a delay of tens of seconds, which causes service interruption. Therefore, we design a Per-Flow On-Demand (PFOD) servicing mechanism to achieve real-time network function switching.
- **On-Path Traffic Serving**. If the traffic service chain demand cannot be completed along the Shortest Path (SP), the QoS of traffic will be damaged. The difficulty is to ensure on-path servicing for all traffic as much as possible, which requires making full use of the resources on the SP to satisfy more functional demands. Therefore, we propose an SP-aware NF Placement Algorithm (SNPA), and a Two-Stage Service Path Construction Algorithm (TSPC) to ensure servicing quality.

III. SYSTEM OVERVIEW

In this section, we provide an overview of FlexNF, which includes the data plane and control plane module components. Fig. 3 illustrates the architecture and the workflow of FlexNF for orchestrating PDP-offloaded NFs.

A. FlexNF Data Plane

The data plane is designed for two goals: 1) orchestrating NFs flexibly to enable arbitrary combinations and orders of NF executions; 2) switching NFs in realtime to adapt to the dynamic demands of flows. The data plane mainly consists of two modules: the NF Selection Framework (§IV-A) and the Per-Flow On-Demand Servicing Mechanism (§IV-B).

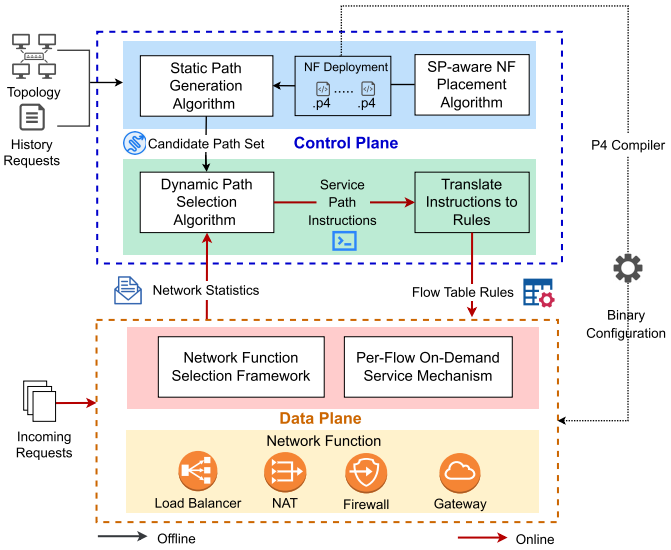


Fig. 3. The architecture of FlexNF.

NF Selection Framework (NSF). To satisfy diverse service chain demands, different NF instances within one single switch or across multiple switches should be dynamically composed together, with some instances selected and others skipped along the forwarding paths. To this end, we design an NF Selection Framework (NSF) with a *Flow Classifier* to allocate service selection labels and a *Service Selection Process* to flexibly select or skip some NF instances. Moreover, we propose a *Multi-Level Re-enter Mechanism* based on the pipeline re-enter mechanism (i.e., resubmit and recirculation) [25], [26] provided by the P4 switch ASIC to offer more possible ways of NF orchestration, bringing higher flexibility and scalability.

Per-Flow On-Demand (PFOD) Servicing Mechanism. To support non-disruptive NF switching to adapt to dynamic changes in traffic demands, we propose a PFOD servicing mechanism. In the PFOD, multiple NFs are deployed in one Match-Action Table. Different NFs can be assigned to different flows and they can be switched dynamically by installing flow table rules.

B. FlexNF Control Plane

The control plane has two tasks: 1) deciding the NF instance placement strategy, based on which NF instances are installed on switches; and 2) finding the best service path for each flow to fulfill the requested NFs with low overheads.

To fulfill the first task, we propose an **SP-aware NF Placement Algorithm** (§V-A), which is called offline when the topology is constructed or changed. It aims at finding an NF deployment solution that minimizes the total route lengths of all requests as much as possible, with memory constraints on all switches along the paths satisfied. To this end, the proposed algorithm analyzes historical requests and prioritizes the deployment of popular network functions by placing them on the nodes that cover the most shortest paths.

To fulfill the second task, we design a two-stage service path construction strategy to quickly find a low-overhead service path to serve each new flow. The first stage, i.e., a **Static Path Generation Algorithm** (§V-B1), is executed offline only when the topology is constructed or changed. It takes each flow's historic demands as inputs and calculates a candidate path set which contains the top-k shortest paths to fulfill the requested NFs of the flow. Whenever a new flow request arrives, the second stage of the strategy, i.e., the **Dynamic Path Selection Algorithm** (§V-B2), works out the optimal paths with minimum link loads from the candidate set generated by the first stage, while considering realtime load balancing.

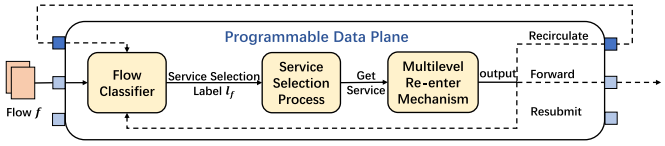


Fig. 4. NF Selection Framework.

C. FlexNF Workflow

Assuming that a flow f with the service chain demand "*NAT→BFD*" arrives at an ingress switch, FlexNF processes it through four steps. First, as shown in Fig. 3, when the first packet of f enters the ingress switch, a match table miss occurs in the Flow Classifier, as the label entries of this flow are not installed yet, then the flow is sent to the controller. Second, after receiving the *PacketIn* event, the controller runs the Dynamic Path Construction Algorithm to choose the optimal path with the requested NFs of this flow from the candidate path set generated by the Static Path Generation Algorithm. Third, for each switch along the selected path, the controller installs the label entries in the Flow Classifier, the forwarding table and the NF table for f. Last, the subsequent packets of f will match the installed rules on the switches along the chosen optimal path, and be forwarded to the destination with the service chain demand fulfilled.

IV. Data Plane Design

The FlexNF data plane consists of an NF Selection Framework and a Per-Flow On-Demand Servicing Mechanism, which are designed to improve the scalability of the service chain and enable non-disruptive NF switching.

A. NF Selection Framework

As shown in Fig. 4, to achieve fine-grained NF orchestration on the PDP, we propose the NF Selection Framework (NSF) with three components: Flow Classifier, Service Selection Process, and Multilevel Re-enter Mechanism. Algorithm 1 describes the workflow of NSF.

When the first packet of one flow arrives at the ingress switch, it will be sent to the control plane due to a match table miss in the Flow Classifier. After receiving the packet, the control plane issues the service selection label l_f to the Flow Classifier. Then the NSF performs the following process. First, the Flow Classifier (lines 1-5 in Algorithm 1) assigns a service selection label l_f for incoming packets belonging to flow f based on the flow key key_f (e.g., IP src, IP src/dst or the 5-tuple, implementation depends on the practical demand),

Algorithm 1 NF Selection Framework Logic

```
Input: packet pkt ∈flow f
Output: service selection label l_f, iteration label l'_f
1  // Flow Classifier
2  Get service selection label l_f based on flow key key_f ;
3  Get re-enter times t_f and the amount of network
     functions k;
4  Mask m ← [(1 ≪ k) − 1] ≪ t_f · k;
5  Iteration label l'_f ← l_f&m;
6  // Service Selection Process
7  for i ← 0 to |N| do
8  |   Mask m ← (1 ≪ i) ;
9  |   if l'_f&m = 1 then
10 |   |   Apply the i-th NF n_i ∈ N ;
11 |   end
12 end
13 // Multilevel Re-enter Mechanism
14 Used length l ← (k · (t_f + 1)) ;
15 Remaining label r_f ← l_f ≫ l ;
16 if r_f = 0 then
17 |   Forward ;
18 end
19 else
20 |   if t_f = 0 then
21 |   |   Resubmit ;
22 |   end
23 |   else
24 |   |   Recirculate ;
25 |   end
26 |   t_f ← t_f + 1;
27 end
```

entries of which are installed by the controller at runtime. l_f is composed of k iteration labels l'_f, where k is the number of NF instances deployed on the switch and each l'_f contains the processing information for a single pipeline iteration. The i-th bit of the iteration label l'_f in binary form indicates whether packets of flow f should be processed by the i-th NF in this iteration. For example, for a switch deployed with NAT→LB→FW, an iteration label 0b101 means that the corresponding flow needs to pass through NAT and FW with LB being skipped. Next, the module will truncate the service selection label l_f according to the re-enter times t_f for the iteration label l'_f. The re-enter times t_f should be stored in the available field of the packet header (such as the TOS field of IP protocol and VLAN ID field) to maintain the information across pipeline processing iterations. Then, the iteration label l'_f is truncated from the service selection label l_f after performing the AND bit operation with the constructed mask, whose corresponding bits are set to 1.

The Service Selection Process reduces the granularity of service chain composition from device-level service to NF-level service and ensures the flexibility of service chain construction. Lines 7-12 in Algorithm 1 demonstrate the control logic of the Service Selection Process. N represents the set of network function instances deployed on the PDP, and $|N|$ stands for the number of NFs. For the incoming packets, the Service Selection Process iterates through N and decides whether to apply each NF according to the bit-wise comparison result of iteration label l'_f. If the i-th bit of l'_f is set to 1, the current packets will pass through the i-th NF.

The Service Selection Process brings huge runtime flexibility by eliminating the tight coupling of physical nodes and consolidated service. The extra latency incurred by the Service Selection Process is proved to be negligible in Section VII-B (about 1.6% of the extra forwarding delay).

Conventionally, traffic can only pass through the pipeline once with the predetermined NF order. As a consequence, the number of possible NF combinations is limited by the deployed sequence of switches. In FlexNF, we design a Multilevel Re-enter Mechanism to offer richer service function chain demands with existing switch resources, and bring higher flexibility and scalability. Lines 14-27 in Algorithm 1 describe the control logic of the Multilevel Re-enter Mechanism. First, the mechanism checks whether each flow has completed all processing by calculating the unused part of the service selection label l_f. Based on the re-enter times t_f, we can obtain the remaining label by right shifting the service selection label l'_f by the used length, i.e., l bits, to determine whether the next round of processing is needed.

For packets that have to traverse the pipeline multiple times, we utilize two kinds of pipeline re-enter mechanisms i.e., resubmitting and recirculating provided by the programmable switch chip to balance the latency cost and flexibility. The resubmitting mechanism sends packets from the ingress deparser unit back to the ingress parser unit [27] whereas the recirculating mechanism sends packets back through a dedicated loopback port. However, these mechanisms have their own shortcomings. Resubmitting only supports one iteration of re-entering due to the hardware limitation while the recirculating incurs longer latency than resubmitting (about 5 times by our evaluation in Section VII-B). To this end, the Multilevel Re-enter Mechanism maintains re-enter times t_f and chooses resubmit or recirculation for packets accordingly. When $t_f = 0$, the corresponding packets are passing through the pipeline for the first time and are resubmitted for a shorter delay. Meanwhile, for packets that enter the pipeline more than once, the recirculating is applied. Although the number of recirculating operations is not technically limited, it should be set cautiously, considering the bandwidth limitation of the dedicated loopback port. That is, the controller needs to carefully weigh the number of recirculation on each programmable switch to avoid affecting the throughput performance of flows. Then, for packets that need to be processed for another round, the re-enter times record t_f should be incremented by one. In NSF, the re-enter times record is stored in the idle field of every packet (e.g., the TOS field of IP protocol, or the VLAN ID field), so that consistency is maintained through multiple rounds of pipeline processing. In addition, before forwarding, the re-enter times record will be set to zero to avoid confusing the processing in subsequent switches.

TABLE II
TIME OF CHANGING THE INSTALLED NF ON P4 HARDWARE SWITCH

Method	Edgecore	H3C	Openmesh	PFOD (ours)
Time(s)	31	17	10	0.0028

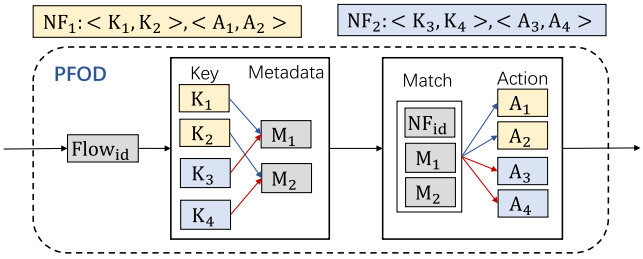


Fig. 5. Per-Flow On-Demand servicing mechanism.

B. Per-Flow On-Demand Servicing Mechanism

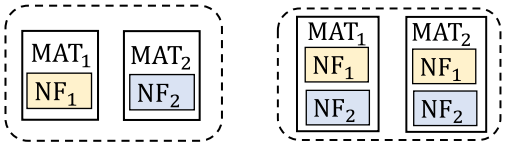
Although the NF Selection Framework (NSF) provides a flexible combination of NFs, services offered by a single switch are still limited by the pre-installed functions. However, the traffic demands are dynamically changing. When the demands of traffic passing through function switches change drastically, the installed NF needs to be adjusted in time to avoid detours and service quality degradation. However, as shown in Table II, reconfiguration of P4 hardware switches induces delays of tens of seconds, significantly damaging the service quality. The reconfiguration process includes shutting down the current P4 program, starting a new ready-compiled program, configuring ports and flow table rules.

HyperVDP [24] is proposed to support online adjustment of installed NF. It uses three pipelines to simulate each native Match-Action Table (MAT), where each MAT in the matching pipeline uses TCAM to match the header, standard metadata and user-defined metadata, respectively. Although it provided flexibility in dynamical adjustment, huge memory overhead (more than 800bits/flow) was introduced.

In FlexNF, we propose the Per-Flow On-Demand (PFOD) servicing mechanism that can dynamically change the NF configuration according to the flow demands at a small cost.

The design of PFOD is shown in Fig. 5. It is constructed by a composite matching-action table. The matching field in PFOD is a collection of matching fields required by all NFs, whose length equals the maximum length of potential NF matching fields. Supposing there are two NF functions:NF_1 with keys K_1 (32 bits) and K_2 (32 bits), NF_2 with keys K_3 (64 bits) and K_4 (64 bits). We can merge them in one MAT with match fields M_1 (64 bits) and M_2 (64 bits). To avoid conflicts in matching fields, we set the five-tuple as $Flow_{id}$ to identify a flow as the basis for allocating NF_{id}. The NF_{id} filed is used to indicate which NF should be run, e.g., $NF_{id} = 0$ stands for running NF_1, $< K_1, K_2 >$ are read into $< M_1, M_2 >$ for executing $< A_1, A_2 >$ (the blue line). Different NFs can be assigned to different flows in PFOD, and they can be switched dynamically by installing corresponding rules.

The PFOD provides great flexibility. As shown in Fig. 6(a), NF_1 and NF_2 are deployed in two tables respectively.



(a) Original deployment. (b) PFOD deployment.

Fig. 6. Comparison of original deployment and PFOD deployment.

The switch can only meet the requirements of NF_1, NF_2 and NF_1-NF_2. If the flow's demand changes to NF_2-NF_1, then the NF of the switch needs to be reconfigured. PFOD satisfies the requirement well by switching MAT_1 to NF_2 and MAT_2 to NF_1 without reconfiguration.

V. CONTROL PLANE DESIGN

To ensure the QoS of flows and provide on-path service for dynamic requests, the forwarding delays on service paths should be reduced as much as possible. The control plane achieves this goal through the joint efforts from an offline NF placement algorithm and a two-stage dynamic service chain construction strategy.

A. SP-Aware NF Placement Algorithms (SNPA)

1) Problem Formulation: We model the data plane topology as a graph $G = (V, N, E, P)$, where V, N and E denote the node set, NF set and link set, respectively. Set P contains all the shortest paths between every two nodes in V. Meanwhile, we model the historical requests as $R = (b_r, S_r, v_s, v_d)$, where S_r denotes the service chain demand of request r, v_s denotes the source node, v_d denotes the destination node and b_r denotes the initial data rate used in Dynamic Path Selection Algorithm (Section V-B2).

When the topology is constructed or changed, a decision on how to place NFs on switches must be made. In FlexNF, we aim at finding an NF placement scheme that can minimize the service path length of all incoming flow requests while satisfying memory constraints along the path. To this end, the problem are formulated as follows:

$$\min_{y} \sum_{r_i \in R} \sum_{p_j \in P_i} |p_j|, \tag{1}$$

subject to

$$\sum_{r_i \in R} \sum_{p_j \in P_{r_i}} \left[\eta_p^v \cdot c_f + \gamma_{p_j}^v \cdot y_v^{S_i^j} \cdot c(S_i^j) \right] \leq c_i, \forall v \in V \tag{2}$$

$$\sum_{j=0}^{|S_i|} y_{s(p_j)}^{S_i^j} \cdot y_{d(p_j)}^{S_i^{j+1}} = |S_i| - 1, \quad \forall r_i \in R \tag{3}$$

$$|P_{r_i}| = |S_i| + 1, \quad \forall r_i \in R \tag{4}$$

$$\sum_{n} y_{v_i}^n \leq \alpha, \quad \forall v_i \in V \tag{5}$$

$$\sum_{v} y_v^{n_i} \geq 1, \quad \forall n_i \in N \tag{6}$$

where η_p^v denotes whether path p passes through node v, and γ_p^v denotes whether path p starts from node v. In the

problem formulation, the decision parameter y_v^n is variable, all other parameters are given. The memory consumption c is directly derived from historical traces.

This problem aims at minimizing the total service path length of all requests, as can be seen from the objective function (1), where p_j denotes the j-th sub-path from NF S_j to NF S_{j+1} on the optimal forwarding path P_i for request r_i ($|p_j|$ is the length of path p_j). For simplicity of notation, we extend S_i with source node v_s and destination node v_d.

On each node $v \in V$, forwarding a flow takes c_f switch memory and NF n takes $c(n_i)$ memory for each request. The constraint (2) ensures that forwarding entries and NF flow entries do not exceed switch memory capacity c_i for every switch. The decision variable y_v^n denotes whether NF n is deployed on node v. $s(p_j)$ and $d(p_j)$ denote the source and destination of sub-path p_j. Constraint (3) ensures each request is correctly fulfilled by its allocated path. Constraint (4) describes the size of sub-path set P_{r_i}, which is equal to $|S_i| + 1$ due to $|S_i| + 2$ stages including NF node, source node and destination node of request r_i.

There are some hardware resource limitations on P4 switches. First, each switch has only a limited number of stages to deploy network functions (e.g., each pipeline on the tofino switch has only 12 stages). Second, the metadata has limited length to store the label (the length is $O(NFs)$) of the NF Selection Mechanism. As such, Constraint (5) limits the number of NFs on each node with an adjustable threshold α. This constraint ensures that the NF deployment scheme does not exceed the hardware limitations of switches. FlexNF operators need to adjust α according to the specific switch hardware resource limitations. In our experiments, after actual deployment testing, we set α to 10. Constraint (6) ensures that each network function is deployed at least once. It is worth noting that the SNPA algorithm is executed in the offline mode, during which the actual utilization overheads of links are not known. As such, bandwidth constraints are not considered by the SNPA algorithm. Instead, they will be considered in the Dynamic Path Selection Algorithm (Section V-B2) in online mode to ensure that the link utilization is not saturated.

2) A Greedy Algorithm for the SP-Aware NF Placement Problem: The SNPA for NF placement takes historical flows as inputs and calculates the total service path length. However, the distribution of traffic can change over time (i.e., the distribution of incoming flows differs from the historical flows). In order to adapt to changes in traffic distribution, we set a time window (i.e., 10s) to periodically collect historical flow information and execute the SNPA to update the deployment of NFs. To this end, we need to design an algorithm to solve the NF placement problem within $O(10s)$ time complexity.

NF Placement Problem has been proven NP-hard [28]. Mathematical programming methods such as Integer Linear Programming (ILP) [29] and mixed ILP (MILP) [30] are applied to solve the problem. However, these methods suffer from high computational complexity when obtaining the optimal solutions. Moreover, since the objective function (1) can not be expressed explicitly by the variable y_v^n, we cannot directly use the ILP/MILP solver to solve the problem. We propose a greedy algorithm (Algorithm 2) to solve the SP-aware NF placement problem with near-optimal solutions but a relatively short execution time.

Algorithm 2 SP-aware NF Placement Algorithm

Input: Topology $G = (N, V, E)$, Requests R
Output: NF Placement Strategy $y = \left\{y_1^1, y_2^1, \ldots, y_{|V|}^1, \ldots, y_1^{|N|}, y_2^{|N|}, \ldots y_{|V|}^{|N|}\right\}$

1 Step 1: Calculate NF popularity: (NF_i, Pop_i, Req_i).
2 $NFList = [(NF_1, 0, \emptyset), \ldots, (NF_{|N|}, 0, \emptyset)]$
3 **for** *request r* $\in R$ **do**
4 **for** *network function nf* $\in r.S_r$ **do**
5 $NFList[nf][1] \leftarrow NFList[nf][1] + 1$
6 $NFList[nf][2] \leftarrow NFList[nf][2] \cup \{r\}$
7 **end**
8 **end**
9 Sort $NFlist$ in descending order by popularity.
10 Step 2: Deploy network functions.
11 **for** *nf* $\in$ *NFlist* **do**
12 $P \leftarrow \emptyset$; $\forall n \in N, M_n \leftarrow \emptyset$;
13 **for** *request r* $\in$ *nf.requests* **do**
14 $path = \text{Dijikstra}(G, r.v_s, r.v_d)$;
15 $P \leftarrow P \cup \{path\}$;
16 $\forall n \in path,\ M_n \leftarrow M_n \cup \{path\}$;
17 **end**
18 flag $\leftarrow$ **True**;
19 **while** *flag && (*$P \neq \emptyset$*)* **do**
20 **for** *node* $n \in N$ **do**
21 $score \leftarrow |M_n \cap P|$;
22 $NodeScore[n] = (n, score)$;
23 **end**
24 Sort $NodeScore$ in descending order by score.
25 flag $\leftarrow$ **False**;
26 **for** *node* $n \in NodeScore$ **do**
27 **if** *meetcondition(n, nf)* **then**
28 $P \leftarrow P \backslash M_j$; $y_j^{nf} \leftarrow 1$; flag $\leftarrow$ **True**;
29 **end**
30 **end**
31 **end**
32 **end**

Our optimization goal is to minimize the route length of all requests. To this end, we endeavor to deploy NF functions on the shortest path as much as possible. The algorithm prioritizes the deployment of popular network functions, preferring nodes that cover the shortest paths.

First, lines 3-8 count the occurrences of different NFs in all service chains as the popularity. Second, for the function nf, lines 13-17 calculate the shortest path set P of requests that need to go through nf and the shortest path set M_n covered by node n. Third, lines 25-29 preferentially select nodes covering the most paths in P to deploy nf. The $meetcondition(n, nf)$ function determines whether the deployment of nf on node n meets the constraints (2) and (5).

Algorithm 3 Static Path Construction Algorithm

Input: Topology $G = (N, V, E)$, Request R
Output: NF-Forwarding Graph g and Path Set P

```
Step 1: Construct Forward Graph
for i ← 1 to |N_r| + 2 do
    if v ∈ N_i.nodes then
        g.add(v);
        for j ∈ N_{i-1}.nodes do
            g[j][v] ← d[j][v] ;
        end
    end
end
Step 2: Get Candidate Path Set
P' ← KSP(g, k * 2)
for path p ∈ P do
    if !p.containsLoop() then
        P.add(p);
        if P.size==k then
            break ;
        end
    end
end
```

Algorithm 4 Dynamic Path Selection Algorithm

Input: Candidate Path Set P of Request R
Output: Optimal Path p^* of Request R

```
minimum metrics min ← MAXIMUM;
for path p ∈ P do
    for edge e ∈ p do
        if b_e < b_r or c_e > c_r then
            if m_p < b_r / b_e then
                m_p ← b_r / b_e;
            end
        else
            m_p ← MAXIMUM;
        end
    end
    if m_p < min then
        min ← m_p; p* ← p;
    end
end
Return p*
```

B. Two-Stage Service Path Construction Algorithm

When a new request arrives, it is necessary to map the requested NFs to the underlying PDP, while satisfying all the service requirements and maintaining low overhead. When designing a service chain mapping algorithm, not only the routing path length but also the dynamic condition of the network should be taken into consideration, since both affect the QoS of flows. Besides, in order to achieve real-time service chain mapping, the algorithm must be time-efficient. To achieve the above goals, we propose the Two-Stage Service Path Construction (TSPC) Algorithm, which consists of a Static Path Generation Algorithm and a Dynamic Path Selection Algorithm.

First, to ensure time-efficiency, the static path generation algorithm is applied to obtain the top-k-shortest candidate path set for each service chain demand in advance, which saves the work of finding the optimal service path of the dynamic algorithm. The static algorithm is only called when the topology is first constructed or changed. Second, to consider dynamic network conditions, the dynamic path selection algorithm evaluates paths from the candidate path set according to the link load ratio in realtime to ensure load balancing.

1) Static Path Generation Algorithm: The static algorithm takes the historic demand file and topology information as inputs. In historic demand file, a request is defined as <Flow key, NF sequence>. FlexNF uses source-destination IP pairs to identify a flow, but it can be easily extended to other flow definition forms. For all the requests defined in historic demand file, the static path construction algorithm calculates the top-k shortest paths, as shown in Algorithm 3.

First, an NF forwarding graph based on the topology graph and SFC request is constructed. The forwarding graph contains $n+2$ stages, where n is the length of service chain. The first and last stages are the source and destination nodes of the flow. The remaining nodes in each stage are the switches or servers with the corresponding NFs installed. The edge of the forwarding graph is assigned with the distance of the shortest path between two nodes, which can be obtained by collecting network topology information. Note that when the nodes of two adjacent stages are the same, the edge weight is set to 0.

Next, we use the top-k shortest path algorithm, i.e., Yen's algorithm [31], to obtain the set of the top-k shortest paths. Since the path set obtained may contain routing loops, the algorithm is configured to find the top-$2 \times k$ shortest paths to provide redundant paths. Paths that do not contain loops can be added to the candidate set, the algorithm stops when the size of the candidate set reaches k.

2) Dynamic Path Selection Algorithm: When the first packet of a new request is forwarded to the controller, dynamic path selection algorithm is activated to select the optimal path from the candidate sets.

As shown in Algorithm 4, the program will first read candidate path set generated by the static algorithm. To avoid network congestion and further improve network performance, we evaluate the candidate paths in terms of link load usage. Thus, for each candidate path $p \in P$, restrictions on switch memory capacity c_e of each link e and link capacity b_e along the path are first checked to guarantee feasibility. To minimize the load differences on each link, we define the evaluation metric as the maximum link bandwidth utilization (i.e., data rate b_r divided by available bandwidth b_e of link e) on the path p. Then, the algorithm selects the path p^* with the smallest metrics after traversing all the paths of the set. The time complexity of dynamic path selection algorithm is $O(n)$ in terms of the number of candidate path n, which guarantees tolerable delay for packets.

TABLE III
NF IMPLEMENTATION EXAMPLES

Function	Match Field	Action
Flow Classifier	hdr.ipv4.src_ip hdr.ipv4.dst_ip	Get meta.label Get meta.bfd_idx
Load Balancer	hdr.ipv4.dst_ip	Get meta.ip_pool_version
	hdr.ipv4.dst_ip meta.ip_pool_version	Get hdr.ipv4.dst_ip
Big Flow Detector	*	Cal. meta.bfd_state= read register(bfd_register, meta.bfd_idx)
	meta.bfd_state	Cal. meta.bfd_cond. = meta.bfd_state>THRESHOLD ? 1 : 0
	meta.bfd_cond.=1	Cal. meta.bfd_state += 1 Send to CPU
	meta.bfd_cond.=0	Cal. meta.bfd_state += 1

TABLE IV
COMPARISON BETWEEN HASH AND CONTROLLER ISSUE

Trace	Scheme	Collision Rate	Memory Used (KB)
univ1	Hash-2^{12}	33.94%	0.5
	Hash-2^{14}	10.49%	2.0
	Hash-2^{16}	2.67%	8.0
	Hash-2^{18}	0.81%	32.0
	controller issue	**0**	**1.67**
univ2	Hash-2^{14}	22.30%	2.0
	Hash-2^{16}	6.27%	8.0
	Hash-2^{18}	1.81%	32.0
	Hash-2^{20}	0.36%	128.0
	controller issue	**0**	**2.28**

VI. P4 IMPLEMENTATION

A network function usually consists of the sophisticated processing logic and dedicated rule set, which is represented as control flow logic with match-action tables and flow table entries on programmable switches. The complicated development process of implementing a complete NF on PDP, especially when involving stateful operations, requires not only a comprehensive understanding of P4 language but also professional knowledge of programmable hardware devices (e.g., specific hardware target primitives and *register* data structure). In order to simplify the implementation process of NFs and provide a unified management interface, we employ two types of models to represent PDP-based NF implementation, i.e., the stateless NF model and the stateful NF model. The stateless NF model leverages match-action tables to execute required actions on specific flows. The stateful NF model is implemented by the "state-condition-action" pipeline, and allows important flow states to be stored and accessed on the data plane.

A. Stateless NF

The model of stateless NF is implemented via a series of Match-Action Table (MAT) units to impose various processing logic on specified flows. The available matching fields of a MAT in a P4 switch include three types of fields: packet header, standard metadata and user-defined metadata. Actions are applied to modify the fields of header and metadata, thus imposing the desired functions on packets, such as modifying the *ttl* field of IP packets or dropping packets.

We use Layer-4 (L4) Load Balancer as an example to elaborate the Stateless NF model. L4 load balancer scales out services hosted in cloud datacenters by evenly mapping flows destined to a service posted at a virtual IP address (VIP) to a pool of servers with multiple direct IP addresses (DIP). To complete this task, two sequentially placed MATs are needed as in Table III. The first MAT allocates the DIP pool version by matching VIP addresses. In the second one, flows eventually obtain randomly selected DIP addresses from a DIP pool using the hash result of the flow key (e.g., *src* and *dst* IP address).

B. Stateful NF

State refers to information generated when processing previous packets of a flow and can guide the processing of subsequent packets [32]. The P4 platform provides stateful units (SRAM) with reading and writing interfaces exposed to P4 programs. In an NF, a series of stateful units can be declared, with widths equal to the number of bits of the state to be stored, and lengths equal to the maximum number of flows that may pass through the NF.

To ensure correctness, state consistency should be completely guaranteed. That is, subsequent packets of the same flow should access stateful units through the same index. Generally, there are two ways to ensure state consistency.

First, we can hash the header fields of packets to get their NF state indexes. However, if a hash collision occurs, different flows sharing the same hash result would operate on the same state, causing damage to correctness-sensitive NFs, such as the stateful Firewall [33]. In addition, hash collision solutions, including open addressing [34] and separate chaining [35], are not hardware-friendly and difficult to implement.

The second way requires the controller to issue the state index through Flow Classifier function as in Table III. The controller ought to know the occupations of all the declared register arrays on each switch. Incoming flows of a switch will first go through the Flow Classifier and obtain the state index for all the NFs needed on this switch. The implementation details of Flow Classifier are elaborated in IV-A. Though a little extra memory is required for storing the state index, the possibility of state collision can be effectively eliminated.

We use two traces to compare collision rate and memory usage for the two schemes. Table IV shows the result. The first four entries of each trace use a hash function on source and destination IPs and the fifth one obtains the state index through flow table entries issued by the controller. According to Table IV, the hash function method shows poor performance under both circumstances. Only when a larger space is given to stateful units (19.1x and 56.1x larger than the memory used by the controller issue method) can the hash collision rate reduce to a relatively small range (i.e., $< 1\%$), which is still unbearable for correctness-sensitive NFs. Meanwhile, for the controller-issue solution, we allocate stateful units with only 10,000 register cells, which is more than enough for these two traces. The extra memory space used for storing the state index

is positively related to the number of the flows. Thus, we use flow entries to issue the state index.

To better illustrate the implementation model of stateful NF, we use the Big Flow Detector (BFD) as an example. As shown in Table III, the BFD checks whether the recorded flow size exceeds a specific threshold. If so, it reports the corresponding flow to the controller. To simulate the logic of BFD in P4 switches, we use three types of tables, including state table, condition table and action table.

State table is responsible for obtaining the stored state of the flow. Each NF instance has its own stateful units as registers in the programmable switch. SRAM resource for registers is allocated with pre-declared width and length when deployed. The state in BFD is the set of flow sizes for all flows. After fetching the index of state in the Flow Classifier, the packet reads and preserves state in metadata for further operations in State Table. The state fetching is implemented in the default action of State Table, requiring no extra entries.

Condition table directly operates on the state to obtain the transition condition information. By conducting predefined calculations with the state in metadata and the given parameters, we can obtain the transition condition result which is used to determine the next operation. In the BFD, if the state is larger than the $THRESHOLD$, the condition is set to 1, otherwise 0. The number of entries in the condition table is equal to the number of possible state transitions of each NF.

Action table is used to apply actions on both packet and state based on the condition variable. State action allows modifying and rewriting the state back to the register array, so that it can be reused by the next packet. Packet actions include modifying packet header and redirecting packet path, such as dropping and sending to CPU. In the BFD example, there are two different actions. When condition is 0 (which indicates that the flow size is smaller than threshold), it will increment the counter. If condition is set to 1, it represents that a transition of flow state happens, and the packet should be sent to CPU for notification.

C. Hardware-Applicable Modification

Although the NF implementation model conforms to the P4 language grammar and P4 abstract forwarding model, the code implemented based on this model cannot be directly applied to the Tofino switch [16] due to the special design of the hardware switch chip. Therefore, several modifications are made for the real deployment in hardware switches.

1) Limited state accesses. Multiple accesses of the same state in one round of pipeline processing in the hardware switch are not permitted. However, the abstraction of the stateful NF as a finite state machine includes the data path with a loop of state reading → writing, unable to be implemented in hardware. Therefore, we aggregate the actions of each state into a table, and select the corresponding state actions according to the condition and state information of the flow. Fig.7 shows the state access loop through the finite state machine of Dynamic NAT, which dynamically assigns a new source port number and a common global IP address to a new connection and performs the corresponding packet header rewriting. In Dynamic NAT, the current number of ports to be

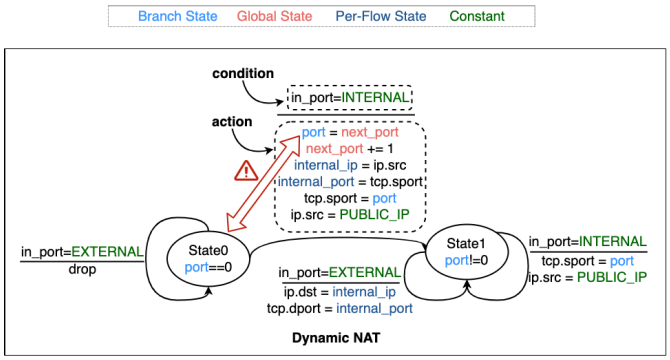


Fig. 7. Workflow of dynamic NAT.

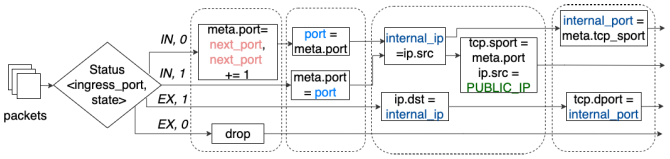


Fig. 8. Workflow of Hardware-applicable dynamic NAT.

allocated is firstly read for each packet and is used along with the *in_port* information to select corresponding actions to take. This process leads to a state access loop, as marked by the red alert arrow in Fig.7. Therefore, we aggregate the actions of each state into a table, and select the corresponding state actions according to the condition and state information of the flow. As shown in Fig. 8, the modified version of Dynamic NAT fits the hardware switches well while maintaining the original function.

2) Stage allocation. When different MATs with data dependencies are allocated to the same stage, the execution sequence will be interfered. Thus, hardware switch programming needs to assign a pipeline stage to each MAT.

3) Limited capacity of registers. The length of a register can only be 8, 16, or 32 bits. Thus, a 48-bit MAC address can only be stored in two registers. Similarly, in the Per Flow Policer and SYN Flood Detection, we also adopt two registers to store 48-bit timestamps, thereby prolonging the overflow time of the function. When the 48-bit state encounters subtraction calculation, the result is obtained by combing the results of calculating the registers with higher and lower bits separately.

VII. Evaluation

We implement FlexNF on both software (BMv2) and hardware (Tofino chip) programmable data plane. In this section, we evaluate the performance of FlexNF.

A. Numerical Analysis of FlexNF on BMv2

1) Experiment Setup: To evaluate the flexibility and scalability of the NF Selection Framework in FlexNF, we build a software experiment environment using BMv2 switches with an ONOS controller running the SP-aware NF Placement Algorithm and the Two-Stage Service Path Construction Algorithm. In the software experiment environments, we deploy 15 types of network functions on each service BMv2 switch, details are shown in Table V. We choose six

TABLE V
NF IMPLEMENTATION PARADIGMS

NF	Description
Firewall (FW)	Blocks specific traffic based on security rules.
SNAT	Translates the private IP to the public IP.
DNAT	Translates the public IP to the private IP.
Load Balancer	Evenly distributes traffic across some servers by mapping the virtual IP address to direct IP addresses.
ARP Proxy	Answers the Address Resolution Protocol (ARP) queries intended for other network devices.
UDP Stateful Firewall	Tracks the states of UDP connections and only allows flows in one direction if a corresponding flow in the opposite direction was first seen.
Port Knocking Firewall	Allows communication only if a sequence of packets to a sequence of predefined ordered ports is received.
TCP Stateful Firewall	Keeps track of the states of active TCP connections and only allows the legal ones.
Per-flow policer	Traffic policer based on per-flow sliding windows.
Big Flow Detector	Keeps track of the number of packets of flows and alerts the controller if a given threshold is exceeded.
SYN flood Detection	Identifies hosts generating more traffic than expected.
Dynamic NAT	Assigns a port number to new connections and rewrites the corresponding packet header.
Super Spreader Detection-TCP	Monitors the number of TCP connections initiated by each host.
ARP Spoofing Detection	Keeps the MAC-IP mapping state and detects if one MAC is mapped to more than one IPs.
DNS Reflection Defense	Tracks if a host has sent a DNS request and filters the replies without a recorded request.

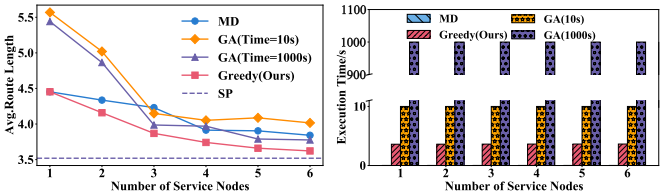
TABLE VI
TOPOLOGIES INFORMATION

Topology	abilene	geant	france	norway	brain	germany
No. of nodes	12	22	25	27	161	50
No. of links	22	36	45	51	166	88

topologies from SDNlib [22] for experiment, whose information is shown in Table VI. The α in SNPA is set to 10 based on the actual deployment tests. We randomly construct some real service chains based on implemented network functions, and randomly assign them to different pairs of source and destination nodes in the experimental topologies, while ensuring the number of flows to request each service chain to be the same. Flow completion time (FCT) and link load information are collected during evaluation through ONOS APIs. They are used as performance metrics to evaluate the scalability and efficiency of our scheme.

We implement the control plane algorithms based on the ONOS controller with about 1500 LoC in Java. The control plane generates NF deployment policies through the SP-aware NF Placement Algorithm. At runtime, the control plane will intercept the *PacketIn* message from a specific interface, and then construct an optimal service path for the incoming flow based on the Two-Stage Service Path Construction Algorithm. After that, the controller installs routing and NF entries, and sends back the first packet that triggers the *PacketIn* message to the data plane.

2) Evaluation on SP-Aware NF Placement Algorithm: **Benchmark**. To verify the effectiveness of the greedy algorithm, we set up three schemes for comparison: 1) Shortest Path (SP), i.e., all requests are completed by the shortest routes, which stands for the theoretical optimal result; 2) Maximum Degree (MD), i.e., service nodes are selected in descending order of node degrees; and 3) Genetic Algorithm (GA) [36], i.e., GA is applied to solve SP-aware NF Placement Problem [2] We randomly generate 10,000 requests on Abilene, and then calculate the average route length of all requests and algorithm execution time as evaluation metrics.



(a) The route length. (b) The execution time.

Fig. 9. Performance of different NF placement schemes.

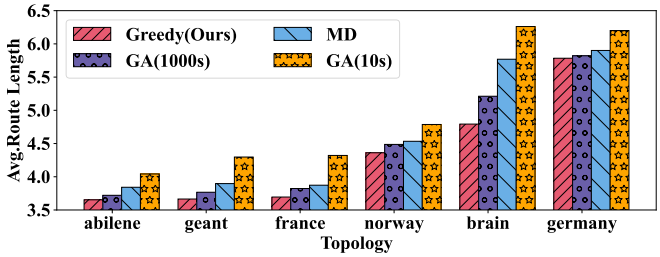


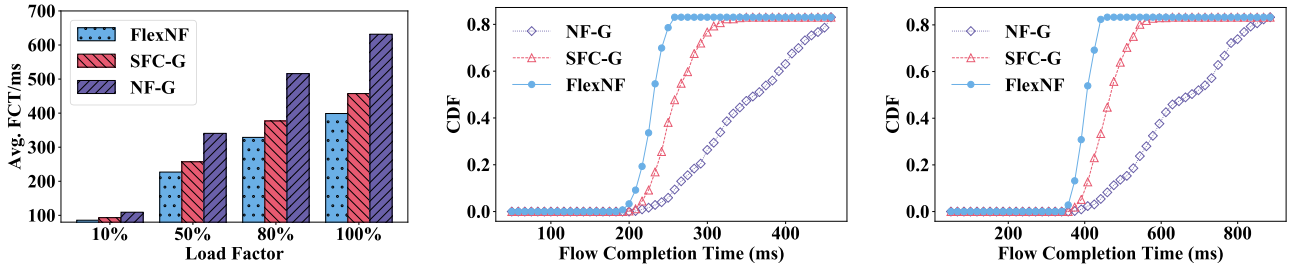
Fig. 10. Average route lengths of different schemes on different topologies.

The route length results are shown in Fig. 9(a), we set a time limit to control the search time of the GA algorithm. GA(Time=10s) and GA(Time=1000s) represent the GA algorithm running for 10s and 1000s, respectively. Greedy (Ours) stands for Algorithm 2. As the number of service nodes increases, the average requests route length of Greedy is more closer to the average route length of SP than MD and GA. When the number of service nodes is 6, Greedy can complete the request within an average of 3.6 hops, while the results of MD, GA(Time=10s) and GA(Time=1000s) are 3.8 hops, 4 hops and 3.7 hops, respectively. Fig. 9 shows the execution time results, the execution time of MD and Greedy are around 0.0175s and 3.6s, respectively. However, GA needs to consume 1000s to achieve good performance, which is still inferior to the performance of Greedy(ours). SP represents the theoretical optimal result, we can find that the result of Greedy(ours) is closer to the optimal result than other schemes. In summary, Greedy(ours) can achieve the shortest route length while ensuring a relatively short execution time.

To evaluate the scalability of FlexNF, we also conduct experiments on different topologies from SDNlib [22]. We compare the average route lengths on different topologies when deploying six service nodes. The results are shown in Fig. 10. Our proposed Greedy algorithm beats the GA algorithm and the max degree algorithm on all six topologies.

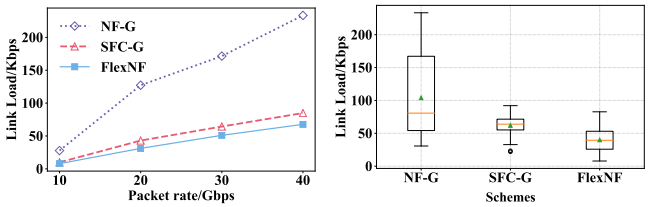
3) FCT and Throughput: **Benchmark**. We compare NSF against two other benchmarks with different deployment granularity for evaluation. The first benchmark deploys at NF granularity, that is, only a single NF is deployed on each

[2]The objective function (1) can not be expressed explicitly by the variable y_v^n, we cannot directly use the ILP/MILP solver to solve the problem, so we use GA for comparison.



(a) Avg. FCT vs. Different loads on Abilene. (b) CDF vs. FCT with 50% load. (c) CDF vs. FCT with 100% load.

Fig. 11. FCTs of different deployment schemes on Abilene. NF-G stands for NF-Granularity deployment while SFC-G stands for SFC-Granularity deployment. FlexNF uses the NF selection framework on PDP for flexbile service chaining.



(a) Loads of heavy link. (b) Link loads vs. Schemes.

Fig. 12. Link load evaluation of different deployment schemes on abilene.

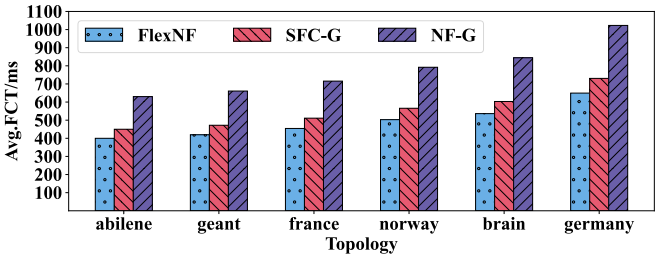


Fig. 13. FCTs of different deployment schemes on different topologies.

programmable switch. In this solution, the service chain path is constructed by traversing all function nodes where the required NFs are located. To solve the potential routing loop problem, we implement the multi-level default path routing solution that combines SFC Table, NF Table, and Flow Table on the P4 switch, as proposed by SAFE-ME [4]. The second benchmark deploys at SFC granularity, that is, an entire service chain is installed on each switch. For each newly-arrived flow, the controller calculates a loop-free routing path passing through the service chain nodes.

We use six nodes in Abilene as the cornerstone to deploy NFs on the switches with NF-Granularity Scheme (a single NF in a switch), SFC-Granularity Scheme (a complete service chain in a switch) and NSF Scheme (the NF Selection Framework in FlexNF, NF placement strategy is generated by SP-aware NF Placement Algorithm). To simulate different intensities of link load, we randomly assign the 132 possible service chain requests with different flow sizes (at a fixed rate), and repeat the experiment for five times.

In Fig. 11(a), for the average FCT, we observe that at 100% load level, deployment with NF Selection Framework significantly outperforms other solutions by 12.9% and 36.9%. The performance advantage is also obvious in Fig. 11(b) and Fig. 11(c). Although the SFC-granularity deployment yields a slightly smaller latency gap, it has obvious scalability shortcomings. That is, the number of service chains supported with SFC-granularity deployment is restricted by the number of functional nodes. On the contrary, the NSF scheme theoretically supports all possible service chain requirements.

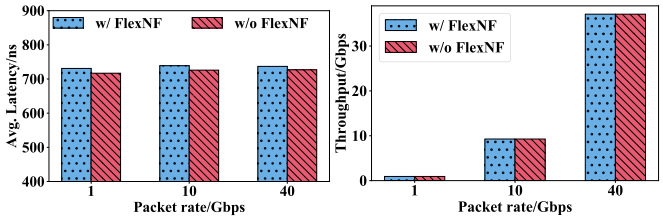
We also evaluate these schemes by link load information collected every 100ms. We compare the load of the heaviest link in each scheme in Fig. 12(a) and the distribution of the average load of all links in Fig. 12(b). Benefiting from jointly considering the shortest routing paths and the load balancing when constructing service chains, FlexNF performs the best among the three and enjoys the lightest link loads. NF-Granularity Scheme shows the heaviest loads due to the longest routing paths. By placing the whole service chain on one switch, SFC-Granularity achieves lighter link loads than NF-Granularity due to the shorter route lengths. However, its link load is still higher than that of FlexNF because of its poor NF deployment flexibility.

To evaluate the scalability of FlexNF, we also conduct experiments on different topologies from SDNlib [22]. We compared the average flow completion time on different topologies under a link load of 100%. The results are shown in Fig. 13. The average flow completion time of FlexNF on six topologies is consistently shorter than that of SFC-G and NF-G. For example, FlexNF outperforms other solutions by 11.1% and 36.5% on Germany topology.

B. Performance of FlexNF on PDP

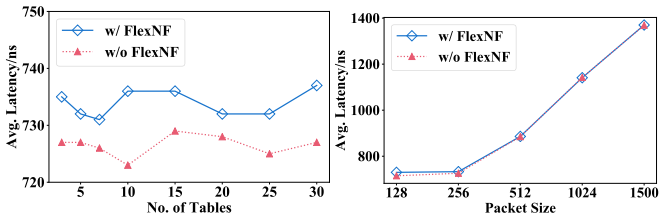
1) Experiment Setup: On the Tofino ASIC target, we implement the same NFs as the software experiment in Section VII-A. Due to the limitation of ASIC on stateful actions, we slightly modify the stateful NFs by merging read, comparison and state modification into a single register action.

2) NF Selection Framework Evaluation: To evaluate the overhead of the NSF in different scenarios, we conduct experiments on different packet rates, different numbers of tables, and different packet sizes with/without the NSF. As shown in Fig. 14, under different packet rates, the delay gap (with or without NSF) is 12ns on average, which only adds 1.6% extra overhead compared to directly installed NFs without NSF, while achieving similar throughput. Fig. 15(a) shows that under 40Gbps traffic, the two schemes tend to have small jitters



(a) Latency vs. Packet rate. (b) Throughput vs. Packet rate.

Fig. 14. Overheads of NSF with different packet rate.



(a) Latency vs. No. of tables. (b) Latency vs. Packet size.

Fig. 15. Overheads of NSF vs. No. of tables and packet size.

when using different numbers of tables in the pipeline, and the maximum difference in delay is 13ns. Fig. 15(b) shows that with a fixed sending rate and number of tables, the latency of the two schemes increases as the packet size increases, but the gap between them reduces to only 1 ns with 512B packet size. In summary, we observe that the cost of the NSF is negligible.

3) Resubmit Evaluation: Fig. 16 shows the impact of the two pipeline re-enter methods on traffic performance (e.g., forwarding delay and throughput). Compared with recirculation, resubmit function has a much smaller impact on the forwarding delay (i.e., an average delay of 100ns) at different data packet sending rates, as shown in Fig. 16(a). Moreover, we can observe from Fig. 16(b) that, although resubmitting traffic would cause the weighted-average forwarding delay to increase, the maximum increment is no more than 40ns. As for the recirculation test in Fig. 16(c), when the recirculate time does not exceed 3, each additional iteration of recirculate adds an additional delay of about 530ns. Otherwise, forwarding delay increases exponentially with hugely-compromised performance and massive packet loss. Nevertheless, the influence of one recirculation iteration on latency is still about 100ns smaller than that of forwarding one more network hop. As such, we expect to utilize recirculate mechanism for more flexibility when more than one iteration is needed.

4) Network Functions Switching Evaluation: We test the performance of the PFOD scheme for switching NFs on the hardware P4 switch. The benchmark is the traditional program switching on the P4 switch. The process is automatically implemented through scripts. We switch the installed NF at different intervals, perform this operation 20 times, and obtain the results as shown in Fig. 17.

As shown in Fig. 17(a) and Fig. 17(b), with the interval time of NF adjusting increasing, the throughput of the PFOD method gradually increases to $98.684Gbps$ when adjusted once in 60s, which almost reaches the normal throughput, and the packet loss rate gradually decreases to 0.007%. However, the program switching method results in a reduction in throughput to $96.644Mbps$ and a packet loss rate of 8.276%. Moreover, as shown in Fig. 18, the stop of the device due to program switching causes a few seconds of zero throughput and escalated packet loss rate, while the PFOD method is able to maintain stable performance.

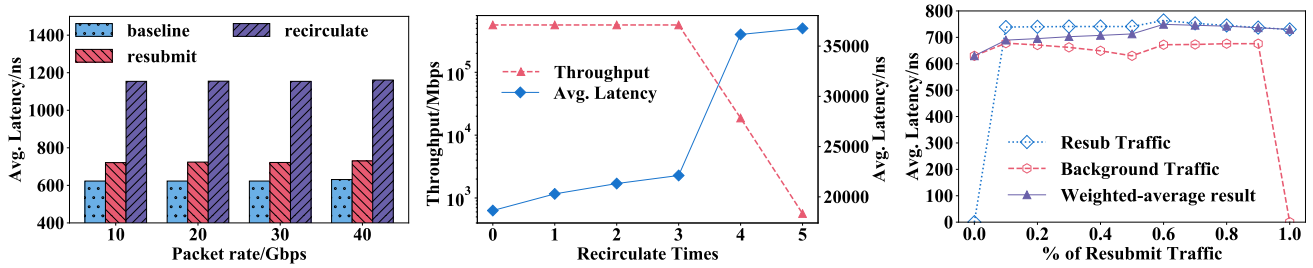
5) Overhead of the Deployment of FlexNF: In this section, we test the hardware resource overheads of FlexNF deployment on the hardware P4 switch.

First, we evaluate the total hardware resource consumption caused by different components (i.e., Labeling and Multilevel Re-enter Mechanism) of FlexNF compared to the baseline Switch.p4 [37]. Switch.p4 is a foundational P4 program that incorporates a range of essential networking functionalities suitable for a standard data center switch [38]. As shown in Table VII, FlexNF only introduces 7.5% SRAM overhead, 8.68% VLIW Actions, 9.62% hash bits and 16.46% Exact Xbar. FlexNF introduces little hardware resource overhead and can be deployed lightweightly on hardware switches.

Second, we implement several network functions on hardware switches to compare overheads of FlexNF with Hyper4 and HyperVDP. Table VIII shows the number of tables used in Hyper4, HyperVDP and FlexNF. Compared with native P4 switch, FlexNF only needs to introduce two additional tables, one for Flow Classifier (i.e., get the service label) and another for Multilevel Re-enter Mechanism (i.e., resubmit/recirculate). Generally, FlexNF reduces 2x to 9x of table usage comparing with Hyper4 and reduces 1x to 2x of table usage comparing with HyperVDP. We also compare TCAM and SRAM usage of FlexNF with Hyper4 and HyperVDP. Fig. 19(a) shows that HyperVDP and Hyper4 introduce an additional TCAM overhead of 5%-15% and 2%-5%, respectively, while FlexNF hardly introduces additional TCAM overhead. Fig. 19(b) shows that SRAM overhead introduced by FlexNF is less than that of HyperVDP and Hyper4. In summary, FlexNF has lower memory overhead than other network virtualization solutions.

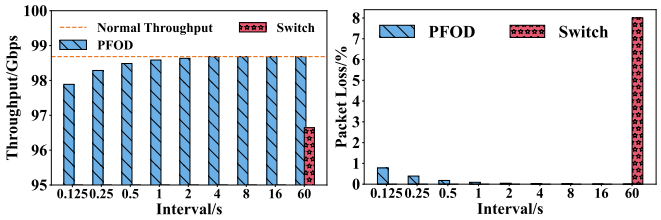
VIII. Related Work

Existing works on hardware offloading are devoted to the following two aspects: 1) implementing specific NFs on PDP with complete function and high-performance service; and 2) proposing new paradigms for NFs on various data plane platforms (including OpenFlow [39], FPGA [33] and the combination of P4 switch and x86 server [10]), focusing on state storage in limited memory space and the problem of state consistency. OpenState [39] was the first to implement advanced stateful applications by extending match-action paradigm of Open Flow [39]. However, OpenState does not mention the rationale to support and organize multiple applications. Besides, without the extension of OpenState, OpenFlow itself provides limited support for stateful operation and is not able to keep states inside the data plane. FPGA-based SmartNIC is another available option to offload VNFs [33], [40], [41], [42]. Microsoft proposes ClickNP [41], which deploys FPGA-based SmartNICs in their datacenters to save CPU usage and reduce traffic transmitted over server's PCIe bus, thus improving NFs' packet processing latency by more than an order of magnitude [33]. However, implementing a complete NF on top of a SmartNIC requires a professional



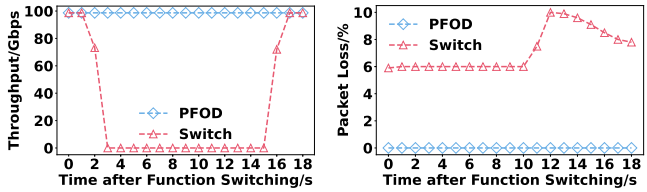
(a) Latency overhead of re-enter operations. (b) Throughput and Latency vs. Times of recirculation. (c) Latency vs. % of resubmit traffic.

Fig. 16. Performance evaluation on resubmit and recirculate operations.



(a) Throughput vs. Switch interval. (b) Packet loss vs. Switch interval.

Fig. 17. Throughput and packet loss at different program switching intervals.



(a) Throughput with interval of 60s. (b) Packet loss with interval of 60s.

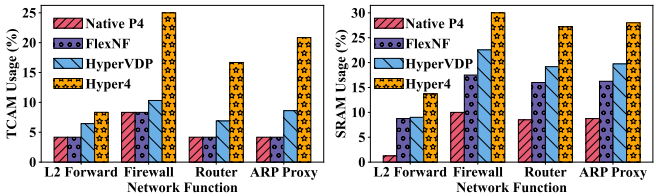
Fig. 18. Throughput and packet loss with program switching interval of 60s.

TABLE VII
HARDWARE RESOURCE CONSUMPTION OF FLEXNF COMPARED TO THE BASELINE SWITCH.P4

Resource	Switch.p4	Labeling	Multilevel Re-enter Mechanism	Combined
SRAM	29.58%	5%	2.5%	32.08%
Stateful ALU	14.58%	0%	0%	14.58%
VLIW Actions	36.72%	3.47%	5.21%	45.40%
TCAM	32.29%	0%	0%	32.29%
Hash Bits	34.74%	4.81%	4.81%	44.36%
Ternary Xbar	43.18%	0%	0%	43.18%
Exact Xbar	29.36%	12.85%	3.61%	45.82%

TABLE VIII
TABLE USAGE FOR DIFFERENT PROGRAMS

NFs	Native P4	Hyper4	HyperVDP	FlexNF
L2 Forward	2	13	5	4
Firewall	3	22	8	5
Router	4	28	16	6
ARP Proxy	4	48	10	6



(a) TCAM usage. (b) SRAM usage.

Fig. 19. TCAM and SRAM usage of different network functions.

understanding of the hardware, making it difficult to introduce new features on existing devices. As a target-independent domain-specific language, P4 offers great programmability by allowing network engineers to customize their protocols using a descriptive programming language. P4 maintains a good balance between expressiveness and simplicity. Meanwhile, P4 allows to maintain information in the data plane during runtime based on its register data structure, and thus has the potential to offload advanced network functions.

To solve the service chaining problem on the PDP, P4Visor [43] merges multiple service chains in a single switch to enhance the service chain scalability. However, PDP still cannot accommodate the service chains that have not been installed in advance. Hyper4 [23] and HyperVDP [24] propose to virtualize the PDP and enable runtime reconfiguration. However, these solutions inevitably cause switch resource waste. ClickP4 [25] offers the potential of dynamically combining features on the PDP, though it still lacks a framework to solve the actual service chaining problem. Some works [44] solve the service chaining problem on the algorithmic aspect by establishing an Integer Linear Programming model regarding different targets. Hyper [42] first proposes a service chaining algorithm on the PDP considering the QoS requirements. However, it only considered service performance, but not the stability of network performance and unfortunately introduced high execution time.

IX. CONCLUSION AND FUTURE WORK

In this paper, we propose FlexNF, a flexible service chain composition framework based on the PDP. First, we design the NF Selection Framework to support the service selection of flows at runtime and enable fine-grained NF orchestration. Second, we propose the Per-Flow On-Demand (PFOD) servicing mechanism to achieve NF runtime switching. In PFOD, one MAT with multiple mixed NFs installed works as different NFs for different flows. Third, to provide on-path service, we propose the SP-aware NF Placement Algorithm and the

Two-Stage Service Path Construction Algorithm to ensure the traffic QoS. Evaluation results show that our system not only outperforms solutions with fixed-node deployment schemes in both traffic QoS and request acceptance ratio, but also shows higher throughput and low packet loss rate. In future work, we will further enhance the SP-aware NF Placement Algorithm and the Two-Stage Service Path Construction Algorithm to achieve differentiated path selection by considering the QoS requirements of flows. For example, we can assign longer paths for flows with lower QoS requirements to spare space for more latency-sensitive flows.

REFERENCES

[1] S. Homma, S. Kumar, C. Captari, M. Tufail, and S. Majee, "Service function chaining use cases in data centers," Internet Engineering Task Force, Internet-Draft, Fremont, CA, USA, Tech. Rep., 2017.

[2] C. Miao et al., "Detecting ephemeral optical events with $\$OpTel\$$," in *Proc. 19th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2022, pp. 339–353.

[3] J. Uttaro, M. Stiemerling, J. Napper, W. Haeffner, and D. López, "Service function chaining use cases in mobile networks," Internet Eng. Task Force, Internet-Draft, Fremont, CA, USA, 2019.

[4] G. Zhao, H. Xu, J. Liu, C. Qian, J. Ge, and L. Huang, "SAFE-ME: Scalable and flexible middlebox policy enforcement with software defined networking," in *Proc. IEEE 27th Int. Conf. Netw. Protocols (ICNP)*, Oct. 2019, pp. 1–11.

[5] B. Yi, X. Wang, K. Li, S. K. Das, and M. Huang, "A comprehensive survey of network function virtualization," *Comput. Netw.*, vol. 133, pp. 212–262, Mar. 2018.

[6] X. Pei et al., "Network functions virtualisation (NFV)," *Manage. Orchestration*, vol. 1, p. V1, 2014.

[7] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu, "NFP: Enabling network function parallelism in NFV," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 43–56.

[8] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "SilkRoad: Making stateful Layer-4 load balancing fast and cheap using switching ASICs," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 15–28.

[9] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proc. Symp. SDN Res.*, Apr. 2017, pp. 164–176.

[10] K. Zhang, D. Zhuo, and A. Krishnamurthy, "Gallium: Automated software middlebox offloading to programmable switches," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun.*, Jul. 2020, pp. 283–295.

[11] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes," in *Proc. ACM SIGCOMM*, 2015, pp. 1–2.

[12] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "HULA: Scalable load balancing using programmable data planes," in *Proc. Symp. SDN Res.*, Mar. 2016, pp. 1–12.

[13] R. Bifulco and G. Rétvári, "A survey on the programmable data plane: Abstractions, architectures, and open problems," in *Proc. IEEE 19th Int. Conf. High Perform. Switching Routing (HPSR)*, Jun. 2018, pp. 1–7.

[14] B. Vass, E. Bérczi-Kovács, C. Raiciu, and G. Rétvári, "Compiling packet programs to reconfigurable switches," in *Proc. 3rd P4 Workshop Eur.*, Dec. 2020, pp. 103–115.

[15] T. Dargahi, A. Caponi, M. Ambrosin, G. Bianchi, and M. Conti, "A survey on the security of stateful SDN data planes," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 3, pp. 1701–1725, 3rd Quart., 2017.

[16] J. Jones. *Intel Tofino2: Second-Generation P4-Programmable Ethernet Switch ASIC That Continues to Deliver Programmability Without Compromise*. [Online]. Available: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html

[17] L. Ye et al., "PUFF: A passive and universal learning-based framework for intra-domain failure detection," in *Proc. IEEE Int. Perform., Comput., Commun. Conf. (IPCCC)*, Oct. 2021, pp. 1–8.

[18] X. Zuo, Q. Li, J. Xiao, D. Zhao, and J. Yong, "Drift-bottle: A lightweight and distributed approach to failure localization in general networks," in *Proc. 18th Int. Conf. Emerg. Netw. Exp. Technol.*, Nov. 2022, pp. 337–348.

[19] Q. Li, J. Xiao, D. Zhao, X. Zuo, W. Tang, and Y. Jiang, "Themis: A passive-active hybrid framework with in-network intelligence for lightweight failure localization," *SSRN 4604412*, Oct. 2023.

[20] Z. Zhang et al., "Pontus: Finding waves in data streams," *Proc. ACM Manage. Data*, vol. 1, no. 1, pp. 1–26, May 2023.

[21] H. Zhao, Q. Li, J. Duan, Y. Jiang, and K. Liu, "FlexNF: Flexible network function orchestration on the programmable data plane," in *Proc. IEEE/ACM 29th Int. Symp. Quality Service (IWQOS)*, Jun. 2021, pp. 1–6.

[22] S. Orlowski, M. Pióro, A. Tomaszewski, and R. Wessäly, "SNDlib 1.0–Survivable network design library," in *Proc. INOC*, 2007, pp. 1–16.

[23] D. Hancock and J. van der Merwe, "HyPer4: Using p4 to virtualize the programmable data plane," in *Proc. 12th Int. Conf. Emerg. Netw. Exp. Technol.*, Dec. 2016, pp. 35–49.

[24] C. Zhang, J. Bi, Y. Zhou, and J. Wu, "HyperVDP: High-performance virtualization of the programmable data plane," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 3, pp. 556–569, Mar. 2019.

[25] Z. Yu and J. Bi, "ClickP4: Towards modular programming of P4," in *Proc. ACM SIGCOMM (Posters Demos)*, 2017, pp. 100–102.

[26] D. Wu, A. Chen, T. S. E. Ng, G. Wang, and H. Wang, "Accelerated service chaining on a single switch ASIC," in *Proc. 18th ACM Workshop Hot Topics Netw.*, Nov. 2019, pp. 141–149.

[27] P. Bosshart et al., "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," in *Proc. ACM SIGCOMM Conf. SIGCOMM*, Aug. 2013, pp. 1–12.

[28] Q. Zhang, Y. Xiao, F. Liu, J. C. S. Lui, J. Guo, and T. Wang, "Joint optimization of chain placement and request scheduling for network function virtualization," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2017, pp. 731–741.

[29] Q. Sun, P. Lu, W. Lu, and Z. Zhu, "Forecast-assisted NFV service chain deployment based on affiliation-aware vNF placement," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2016, pp. 1–6.

[30] T. Lin, Z. Zhou, M. Tornatore, and B. Mukherjee, "Demand-aware network function placement," *J. Lightw. Technol.*, vol. 34, no. 11, pp. 2590–2600, Jun. 2016.

[31] J. Y. Yen, "Finding the k shortest loopless paths in a network," *Manage. Sci.*, vol. 17, no. 11, pp. 712–716, Jul. 1971.

[32] A. Gember-Jacobson et. al., "OpenNF: Enabling innovation in network function control," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 163–174, 2014.

[33] S. Pontarelli et al., "FlowBlaze: Stateful packet processing in hardware," in *Proc. USENIX NSDI*, 2019, pp. 1–17.

[34] H. Gao, J. F. Groote, and W. H. Hesselink, "Lock-free dynamic hash tables with open addressing," *Distrib. Comput.*, vol. 18, no. 1, pp. 21–42, Jul. 2005.

[35] M. V. Ramakrishna, E. Fu, and E. Bahcekapili, "Efficient hardware hashing functions for high performance computers," *IEEE Trans. Comput.*, vol. 46, no. 12, pp. 1378–1381, Dec. 1997.

[36] D. Whitley, "A genetic algorithm tutorial," *Statist. Comput.*, vol. 4, no. 2, pp. 65–85, Jun. 1994.

[37] P. L. Consortium. *Baseline Switch.p4*. Accessed: Jan. 15, 2023. [Online]. Available: https://github.com/p4lang/switch/blob//master/p4src/switch.p4

[38] P. G. Kannan, N. Budhdev, R. Joshi, and M. C. Chan, "Debugging transient faults in data centers using synchronized network-wide packet histories," in *Proc. 18th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2021, pp. 253–268.

[39] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "OpenState: Programming platform-independent stateful openflow applications inside the switch," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 44–51, Apr. 2014.

[40] C. Sun et al., "SDPA: Toward a stateful data plane in software-defined networking," *IEEE/ACM Trans. Netw.*, vol. 25, no. 6, pp. 3294–3308, Dec. 2017.

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

[42] C. Sun, J. Bi, Z. Zheng, and H. Hu, "HYPER: A hybrid high-performance framework for network function virtualization," *IEEE J. Sel. Areas Commun.*, vol. 35, no. 11, pp. 2490–2500, Nov. 2017.

[43] P. Zheng, T. Benson, and C. Hu, "P4 Visor: Lightweight virtualization and composition primitives for building and testing modular programs," in *Proc. 14th Int. Conf. Emerg. Netw. Exp. Technol.*, Dec. 2018, pp. 98–111.

[44] G. Lee, M. Kim, S. Choo, S. Pack, and Y. Kim, "Optimal flow distribution in service function chaining," in *Proc. 10th Int. Conf. Future Internet*, Jun. 2015, pp. 17–20.

Jingyu Xiao received the B.S. degree from Wuhan University, Wuhan, China, in 2021. He is currently pursuing the M.S. degree with the Tsinghua Shenzhen International Graduate School. His research interests include network failure localization, programmable data planes, and AI for networks.

Xudong Zuo (Member, IEEE) received the B.S. degree in mathematics from Nanjing University in 2018. He is currently pursuing the M.S. degree with the Tsinghua Shenzhen International Graduate School. His research interests include programmable switches and failure detection and localization in networks.

Qing Li (Senior Member, IEEE) received the B.S. degree in computer science and technology from the Dalian University of Technology, Dalian, China, in 2008, and the Ph.D. degree in computer science and technology from Tsinghua University, Beijing, China, in 2013. He is currently an Associate Researcher with the Peng Cheng Laboratory, China. His research interests include reliable and scalable routing of the internet, software defined networks, network function virtualization, in-network caching/computing, and intelligent selfrunning networks.

Dan Zhao received the bachelor's degree in telecommunications from the Beijing University of Posts and Telecommunications in 2011 and the Ph.D. degree in electronic engineering from the Queen Mary University of London in 2015. She was a Post-Doctoral Researcher with the School of Electronic Engineering, Dublin City University, and the School of Computing, National College of Ireland. She is currently an Assistant Researcher with the Peng Cheng Laboratory, Shenzhen, China.

Hanyu Zhao received the B.E. degree from the Huazhong University of Science and Technology and the master's degree from Tsinghua University in 2021. Her research interests include programmable data planes and network function virtualization.

Yong Jiang (Member, IEEE) received the B.S. and Ph.D. degrees in computer science and technology from Tsinghua University, Beijing, China, in 1998 and 2002, respectively. He is currently a Full Professor with the Tsinghua Shenzhen International Graduate School. His research interests include the future network architecture, the internet QoS, software defined networks, and network function virtualization.

Jiyong Sun received the bachelor's degree in computer science and technology from the Hunan University of Arts and Science in 2005. He is currently with China Mobile Communications Group Guangdong Company Ltd. His research interests include 5G network architecture, car networking, and industrial and internet solutions.

Bin Chen received the bachelor's degree from the South China University of Technology in 2002 and the master's degree in 2005. He is currently with China Mobile Communications Group Guangdong Company Ltd. His research interests include intelligent self running networks, big data analysis and mining, and network quality analysis and optimization.

Yong Liang received the bachelor's degree in microelectronics technology and the master's degree in electronics and information engineering from the South China University of Technology in 2002 and 2006, respectively. He is currently the Project Manager of China Mobile Communications Group Guangdong Company Ltd. His research interests include planning for IT support systems, data centers, and computing force networks.

Jie Li (Member, IEEE) received the bachelor's degree in computer science and technology from the Huazhong University of Science and Technology in 2002 and the master's degree in engineering project management from Sun Yat-sen University in 2010. He is currently an Engineer with China Mobile Communications Group Guangdong Company Ltd. His research interests include planning for IT support systems, data centers, network security, and data security.