A NFV Acceleration Scheme Based on Programmable Data Plane

***Abstract*—To be written**

***Index Terms*—NFV, Programmable Data Plane, SFC**

I. Introduction

Network Functions (NFs) such as load balancers, firewalls, intrusion detection systems, etc. are traditionally deployed on proprietary devices called middleboxes to provide specific services for traffic. A flow typically requires to traverse serveral NFs in order to form Service Function Chain (SFC). These massive proprietary physical devices have brought in the problem of network rigidity and difficulty in network management and infrastructure expansion.

Network Function Virtualization (NFV) was introduced as a new network paradigm to address the limitations of dedicated middleboxes. NFV offers great flexibility and scalability of service provision with lower cost by decoupling network functions from the underlying specialized hardware and realizing them on common Commercial-Off-The-Shelf (COTS) devices. Nevertheless, the benefits of NFV comes with performance compromises. The giant performance gap between general-purpose devices and dedicated middleboxes could result in considerable latency costs. One VNF could add about 600 microseconds of delay to packets in addition to the overhead from transmitting packets from NIC to CPU through PCIe. Meanwhile, more CPU cores are needed to deploy VNFs, resulting in fewer resources for service provision [1].

Recently, some research efforts have been devoted to accelerate NFV, which could be divided into two types. 1) NF Architecture Optimization. For example, OpenBox proposes NF modularization and improves overall performance by sharing common building blocks between NFs. Moreover, NFP innovatively adopts NF parallelism to improve NFV performance. 2) Packet I/O Acceleration. Some works (NetBricks, DPDK, ClickOS, NetVM, Netmap) propose to achieve NF acceleration by optimizing packet delivery from NIC to VM, or between the VMs. However, software-based solutions still cannot bridge the giant gap in performance.

With the advent of Programmable Data Planes (PDP), using data planes to offload NFs is gradually seen as an effective method to improve NFV performance. In Software Defined Networking (SDN) context, the control plane is the centralized control element in charge of making routing and traffic engineering decisions and managing data plane devices. The data plane, in contrast, is responsible solely for executing the packet processing policy set by the control plane. The data plane will perform packet processing policy set by controller through southbound API, and can be rapidly and systematically reconfigured thus providing programmablity [2]. The data plane functionality may be implemented in many platforms including ASIC, FPGA and netwrk processor.

Programmable data plane has processing capabilities that match line rate (e.g., 12.8 Tbps of Barefoot Tofino ASIC), and can flexibly implement complex network functions through programming packet processing logic. For example, by installing and managing IP mapping rules of flows, Network Address Translation (NAT) could be implemented on programmable switches. In addition, offloading NF in the data plane helps reduce the influence of traffic routing loop problem in NFV network, and can effectively reduce the usage of CPU cores to ensure the provision of service.

Due to the features of PDP, there are two critical challenges of NF offloading: 1) Service Chaining. Different flows in network have diverse and dynamic SFC requirements. NFs in PDP should be organized to support multiple service chains and provide flexible reconfigurability for operators to cope with changing needs. To address this issue, a naive method is to implement one service chain in one programmable switch, which clearly lacks scalability and is not able to cope with ever-changing requirements[]. Hyper4 takes another way to fully realize virtualization by configure match filed and instructions at runtime at the cost of massive memory usage[]. 2) Stateful Operations. Operation on states is widely involved in NF behaviors (e.g., a big flow detector would access its counter to decide whether to alert the controller).

to select from two versions

In order to reduce the switch-to-controller's signaling overhead of state access and transition, the stateful function logic should be entirely offloaded to switch. Moreover, state consistency, i.e., packets belonging to the same flow processing the same state, should be strictly ensured.

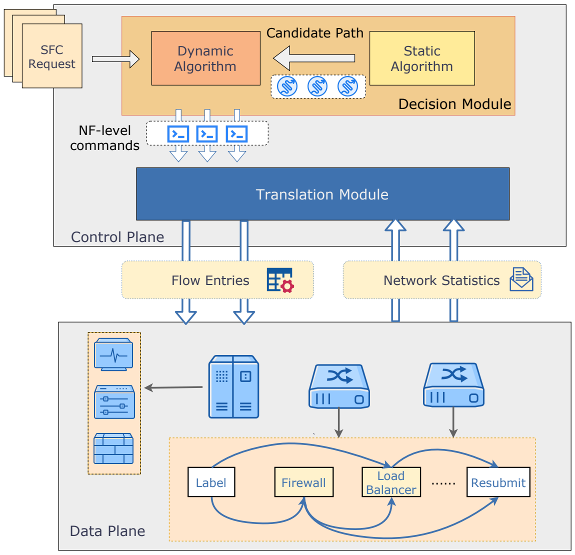


Fig. 1. System architecture overview

However, current PDP platforms lack focus on paradigm of high-level stateful applications. For example, OpenFlow provides little support for maintaining states in switch where flows are not able to react differently when state changes. In contrast, FPGA offers great program flexibility, but is not operator-friendly with hardware description language.

To conquer the above challenges, we design STH, a flexible and high-performance NF management framework which leverages PDP to offload NFs. STH can allocate Hardware-based NFs (HNF) to dynamic traffic to minimize the impact on data plane. We adopted P4 (Programming Protocol-independent Packet Processors) switch as the data plane to offload NFs and designed a P4-based NF implementation abstraction composed of pipeline tables considering both stateless and stateful NFs, where state access and transition operations are entirely inside the switch. To effectively react to dynamic traffic with diverse SFC requirements, we designed the two-stages NF mapping algorithm including static path generation algorithm and dynamic path selection algorithm in the control plane. Our implementation on (name of hardware switch) switch shows that PDP is feasible to offload NF and achieves high performance with line rate.

II. Background

A. P4 Langauage

P4 [3] is a domain specific language, which describes the packet processing pipeline of data plane device. P4 was proposed as a target-independent abstract model for packet processing, in which the programmer does not need to know the specifications of the underlying forwarding device in terms of the device type or technologies(e.g., ASIC, FPGA, CPU). Therefore, the programs written in P4 language, can be mapped to several kinds of devices.

A P4 program is composed of headers, parsers, metadata, match-action tables, and control flows. A match-action table matches the specified packet header fields and selects an action to execute. We use match-action tables to compose NF implementations or as a flow classifier. Metadata in P4 program is used to exchange processing information between tables, and a control flow specifies the order and logic of table processing. P4 community provides sound ecosystem, with its own compiler, P4C, to load the program to target devices and its southbound API, P4Runtime, to expose the device capabilities for operators to populate rules at runtime. Leveraging these comprehensive building blocks provided by P4, we are able to rapidly build and deploy custom high-level applications.

TABLE I
Comparison of Programmable Data Planes

Platforms	Expressiveness	Simplicity	Performance
CPU	✓	✓	×
OpenFlow	×	✓	✓
FPGA	✓	×	✓
P4	✓	✓	✓

B. Hardware Offloading

Recently, some works proposed to offload certain network functions to programmable data plane to address the performance problem of NFV. OpenState was the first to propose to implement advanced stateful applications by extending match-action paradigm of Open Flow [4]. However, OpenState hasn't mentioned 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. On the other hand, FPGA-based SmartNIC is another available option to offload VNF [5] [6] [1] [7]. Microsoft proposed ClickNP, which deployed FPGA-based SmartNICs in their datacenters in order to save CPU usage and reduce the amount of traffic on a server's PCIe bus, thus improving a network function's packet processing latency by more than an order of magnitude [1]. However, programming a Smart-NIC requires proprietary hardware program expertise, which makes it difficult to support new features in existing devices.

As a target-independent domain-specific language, P4 offers great programmability by allowing network engineers to customize their protocol using descriptive language. As in Table I, 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, offering the potential to offload more advanced network functions. In addition, the advent of barefoot tofino ASIC provides the feasibility of deploying VNF on physical hardware device.

C. Service Chaining Solutions

To cope with different service chain requirements, a critical problem rooted inside programmable data plane is to organize and manage NFs to support multiple service chains.

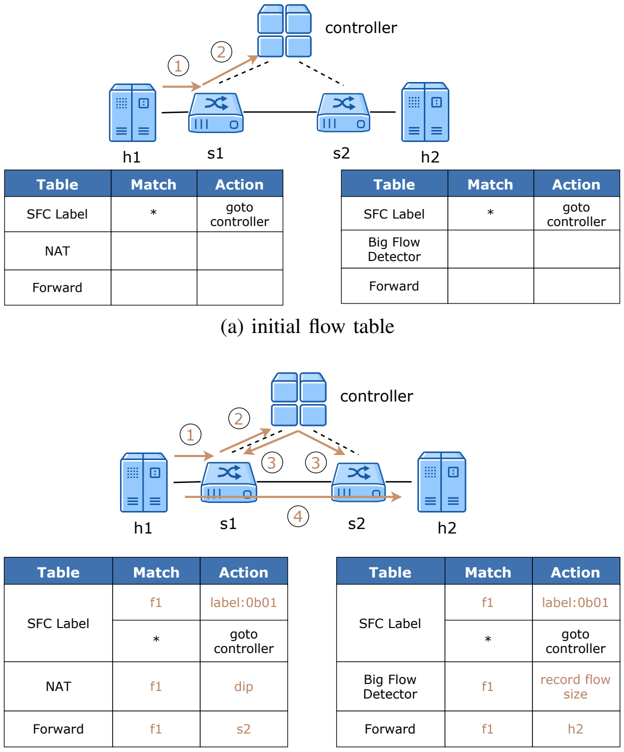


Table	Match	Action
SFC Label	*	goto controller
NAT		
Forward		

Table	Match	Action
SFC Label	*	goto controller
Big Flow Detector		
Forward		

(a) initial flow table

Table	Match	Action
SFC Label	f1	label:0b01
	*	goto controller
NAT	f1	dip
Forward	f1	s2

Table	Match	Action
SFC Label	f1	label:0b01
	*	goto controller
Big Flow Detector	f1	record flow size
Forward	f1	h2

(b) system workflow

Fig. 2. Figure a depicts the flow table of switch 1 and switch 2 when receiving the first packet of incoming flow flow1. The controller receives packet and thus deploy corresponding flow rules in s1 and s2 as in figure b. The green rules means that this rule is matched.

A natural strawman solution for service chaining is to implement one or more service chains in a single programmable switch. P4Visor proposed to improve this solution by merging multiple PDP programs in a single switch through a combination of compiler optimizations and program analysis-based algorithms. Although more flexible than the navie solution, the scheme still lacks reusability and scalability. For each slightly changed request, a service chain entity will be installed and occupy all or half of the switch.

Some researchers [8] [9] offers another way to orchestrate NFs by leveraging match-action tables to simulate control logic, thus virtualizing PDP and realizing real-time reconfiguration. However, these solutions have brought in the problem of resource waste. For a simple service chain example firewall→switch→router, Hyper4 [8] and HyperVDP [9] use 7x and 3.2x table resources respectively, without mentioning other resources such as metadata.

ClickP4 offers the potential of dynamically combining features on programmable switches via tagging flows in front of performing NF logic. But it is limited to program decomposition and feature orchestration without focus on NF implementation model and schedule scheme for multiple switches.

III. System Overview

System architecture consists of HNF instances in data plane and control plane designs for NF management and flow allocation. System workflow is introduced through packet processing procedure.

A. Data Plane

Data plane is responsible for forwarding packets and supporting HNF instances. In STH, we use two kinds of models to represent hardware-based NF implementation. 1) Stateless NF leverages match-action tables to execute required actions on specific flow. 2) As for stateful NF, we adopt the Finite State Machine paradigm simulated by state-condition-action pipeline, thus keeping state access and transition inside data plane.

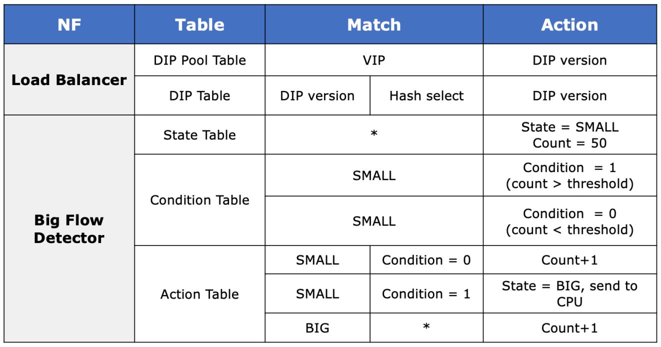
In data plane, multiple HNFs are implemented in a certain order. The specific order and type are determined by the P4 program written by the operator. To fulfill the diverse service chain requirements of flows, HNFs intra- or inter-switches must be dynamically composed. We utilize labels to identify the HNF sequence to go through in the current switch. In this way, flows is able to jump over HNF instances depending on controller decision. As introduced before, maintaining state consistency is a significant task for data plane. To address this issue, indexes for stateful units access of HNFs are installed by controller along with the labels.

Although the primitives provided by the programmable data plane can support a large part of the operations of NF, there are still several NFs that are not suitable for hardware implementation. After research, we found NFs with too complex operations which are not supported by hardware capabilities, such as DPI, Carrier Grade NA(P)T, are not fully supported by the programmable data plane for the operation of packets. In addition, considering that hardware resources are more scarce than software resources, when hardware resources cannot guarantee NF deployment requirements and basic forwarding requirements, it is still necessary to use VNF deployment on general COTS equipment.

B. Control Plane

The control plane is composed of three modules: decision module and translation module.

The decision module is responsible for specifying NF instances for each incoming flow. The rationale of flow allocation is to ensure that switches in path still has sufficient remaining flow space for packets forwarding after installing HNF rules, preserving scalability and flexibility of system. To this end, a two-stage algorithm is contained in decision module that combines static path construction algorithm and dynamic path selection algorithm. The static path construction algorithm takes the flows' demands file as input and calculates the optimal set with k alternative paths for each flow. The candidate set are obtained under a some performance criterion such as having the large capacities or having the shortest number of hops. The static algorithm is only called when the topology changes. The dynamic path selection algorithm selects the currently optimal path from the candidate set through dynamic evaluation criteria (such as link load) combined with the impact of the resubmission operation on the system when



NF	Table	Match		Action
Load Balancer	DIP Pool Table	VIP		DIP version
	DIP Table	DIP version	Hash select	DIP version

Big Flow Detector	State Table	*		State = SMALL
				Count = 50

	Condition Table	SMALL		Condition = 1 (count > threshold)
		SMALL		Condition = 0 (count < threshold)
	Action Table	SMALL	Condition = 0	Count+1
		SMALL	Condition = 1	State = BIG, send to CPU
		BIG	*	Count+1

Fig. 3. Flow Table Implementation of Load Balancer and Big Flow Detector

new flows arrive. The input requirements file defines all possible requests. A request is defined as <source IP, destination IP, NF sequence>. For the simplicity of implementation, our system uses source-destination IP pairs to determine a flow, but it can be easily extended to other flow definition forms. In the request file, operators can customize NF parameters, such as the threshold of Big Flow Detector.

Translation module simplifies operator workflow by providing primitives for decision module to translate NF-level commands into flow table rules of data plane.

C. A Packet's Life

We then describe the packet processing procedure of STH. As shown in 2, the control flow ofSFC table, forward table and corresponding NF table will be proactively installed in data plane as the processing logic of HNF framework. Assume a request flow $f1$ with SFC requirements NAT→Big Flow Detector arrives at switch $s1$, the system will process it through 4 steps. (1) After the first packet of flow $f1$ enters the edge switch, according to Fig. II-C, it will first be sent to the controller via match miss of the SFC label table since label allocation entries of this flow havnt't been installed yet. (2) After receiving packet-in, the controller will run a dynamic selection algorithm to choose an optimal path with requested NFs for the flow. The specific detail of the algorithm is described in Section-5. (3) The controller will install rules of SFC table, forwarding table and NF table for $f1$ on switches along with the selected path. (4) Subsequent packets of $f1$ will match flow table rules at $s1$ and $s2$, and be correctly forwarded to destination with service chain requirements accomplished.

IV. Data Plane Design

Data plane supports the dynamic HNF framework, which could be divided into three parts: NF implementation paradigm, state management and dynamic chaining.

A. NF Implementation Paradigm

Hardware-based NFs in PDP could be classified into two types: stateful NF and stateless NF, and we propose two models to better illustrate their features.

(i) Stateless NF is implemented via a series of Match-Action Table (MAT) units. We use Layer-4 Load Balancer as an example. Stateful layer-4 (L4) load balancers scale out services hosted in cloud datacenters by mapping packets destined to a service with a virtual IP address (VIP) to a pool of servers with multiple direct IP addresses (DIPs or DIP pool). To complete this task, two sequentially placed tables are needed as in Fig.**??**, where the first one allocates DIP pool labels by matching VIPs, and in the second one, flows finally get DIP with DIP pool label and hash result (to randomly select DIP from DIP pool).

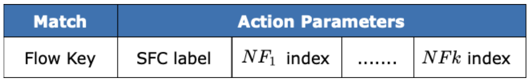
(ii) Stateful NF requires information generated during processing the former packets of the same flow [10]. As Fig.**??** shows, stateful NF can be represented by the Finite State Machine. Each node represents a different state, and each arrow represents the transition between states. When condition on the shoulder are met, state transition will be performed. As shown in the example of Big Flow Detector, when number of recorded packets exceeds the threshold, state will move to BIG and the corresponding flow should be reported to the controller.

To simulate the logic of FSM in P4 switch, we use three types of tables to process packets, called state table, condition table and action table.

- **State table** is responsible for obtaining corresponding state of the flow. Each HNF has its own stateful units as register array in programmable switch. In this example, SMALL and BIG states are respectively represented as 0 and 1. In addition, flow will also need to count packets to compare with threshold. Thus, we concatenate state and packet count information together, e.g. 1100100 means state SMALL with packet count 50 as in Fig.**??**. After fetching state index of this HNF in SFC table, the packet will read and preserve state in its metadata.
- **Condition table** implements state transition logic in FSM. By conducting predefined calculations with metadata and threshold, we can obtain condition result and determine which state to go next. In the BFD example, if state is greater than threshold, the condition value will be set at 1, otherwise 0. The number of condition table entries is equal to the number of state transitions (arrows) in FSM model.
- **Action table** performs actions on both packet and state. State action allows modifying and rewriting state back to the register array, which will be obtained by the next packet. Packet action includes packet header modification or packet dropping. In this example, there are three different possible action. When flow state is small and condition is 0, it is still a small flow, we will just increment the counter; when the flow state is small and condition is 1, it represents that flow state transition

TABLE II
COMPARISON BETWEEN HASH AND CONTROLLER ISSUE

Trace	Hash Space	Collision Rate	Memory Used (KB)
univ1	2^{12}	33.94%	0.5
	2^{14}	10.49%	2.0
	2^{16}	2.67%	8.0
	2^{18}	0.81%	32.0
	controller issue	0	5.8
univ2	2^{14}	22.30%	2.0
	2^{16}	6.27%	8.0
	2^{18}	1.81%	32.0
	2^{20}	0.36%	128.0
	controller issue	0	15.94



Match	Action Parameters			
Flow Key	SFC label	NF_1 index	....	NFk index

Fig. 4. SFC table implementation

happens, and packet will be sent to CPU for notification; When state is big, HNF will keep counting and waiting for controller instruction.

B. Service Chaining Design

To identify diverse SFC path, we utilize SFC labels coding in one-hot form for each flow. A SFC label table is implemented ahead of all the HNF tables in every switch, entries of which are deployed by the controller. To enable reverse order of HNFs on a switch, resubmit operation provided by P4 platform that sends the packet back to the start of pipeline.

The length of SFC label is equal to $|SFC| + 1$ ($|SFC|$ means length of HNF sequence in a switch). The kth bit of SFC label in binary form indicates whether packets should pass the kth NF ($k < |SFC|$). For example, on a switch with SFC Big Flow Detector → Firewall → Load Balancer installed, $0b0101$ indicates that corresponding flow needs to pass through Big Flow Detector and Load Balancer in sequence.

In order to identify resubmission requirements, the highest bit of SFC label indicates whether resubmission is required. We add a resubmission table at the terminal of switch pipeline, with matching field set as flow key, and action parameter set as new flow label. The system now supports flow to go through pipeline for two times. But for situations that require multiple resubmissions, we can simply add an additional matching field of resubmit table to identify the times of resubmissions. However, since resubmission will impact system performance, it's not recommended to overuse resubmit operations.

C. State Management

Operations of network functions usually involve stateful operations. State refers to information generated from operation of previous packet, which could be used to guide subsequent packets of the same flow.

P4 platform provides stateful units (embodied as register array) with reading and writing API exposed to P4 program. In a HNF instance, a series of stateful unit memories will be declared, with width equal to bits of state that needs to be stored, and length equal to the maximum number of flows that may pass through the function.

In order to enable the correctness of function, the data plane needs to ensure state consistency, that is, subsequent packets of the same flow access stateful units through the same index. There are two ways to ensure state consistency. The first one is to hash header fields of packets to get theirs HNF state indexes, thus packets of the same flow is able to get the same result. However, hash collision is an unavoidable problem, leading to different flows operating on the same state, which interferes result correctness. Moreover, hash collision solutions including open addressing and separate chaining are hardware-friendly. The second way is for controller to issue state index through SFC table as in 4. Incoming flows will first have to go through SFC table and obtain the state index for all HNFs in need on this switch. Extra memory will be introduced through storing state index.

To evaluate these methods, we use two traces to compare collision rate and used memory under two schemes. The result is shown in II, the first four entries of each trace use hash function and the fifth one obtains state index issued by controller. Hash function shows a bad performance under two circumstances. Only when a larger space is given to stateful units (6.4x and 8.6x larger than memory used by table rules method) can hash collision rate be reduced to a relatively small range ($< 1\%$).

V. CONTROL PLANE DESIGN

The decision module uses a two-stage algorithm consisting of static and dynamic algorithm to construct SFC routing path for flows.

A. Problem Definition

When a new request arrives, a forwarding path should be constructed meeting following rules: (1) NF sequences in order as requested is provided on the path. (2) nodes along the path have available remaining resources (flow table resources of switches and processing resources of servers) (3) links along the path have available remaining bandwidth. Meanwhile, in order to ensure network performance and avoid potential congestion, the shortest possible path should be selected on the basis of restrictions.

The data plane topology is modeled as an undirected graph $G = (V, N, E)$, where V, N and E denote the node set, NF set and link set respectively. Meanwhile, we model the incoming request to be deployed as $R = (N_r, v_s, v_d, b_r)$. N_r denotes the required service chain of request r. Other attributes of request r include source and destination node v_s, v_d, as well as initial data rate b_r.

$$\min \quad w_l \cdot \frac{m_l}{l} + w_r \cdot \frac{m_r}{|r|} \tag{1}$$

In order to improve performance of request while reducing the possibility of network congestion, we set the target of

TABLE III
NETWORK MODEL NOTATION

Notations	Description
N	The set of NFs within the network
V	The set of nodes including switches V_s and hosts V_h
E	The set of links within the network
N_r	Required SFC of request r
b_r	Estimated data rate of request R
c_v	Residual processing capacity of node v
$b(v, v')$	Residual bandwidth resources of link (v, v')
$x^v_{N_i}$	=1 if NF N_i is implemented on node v
$d^v_{N_i}$	=1 if NF N_{i+1} is implemented before N_i on node v
$y^v_{N_i}$	Binary variable to designate NF N_i is assigned to node v
$t_{v,v'}$	Binary variable to designate result path p contains link (v, v')

constructing forwarding path as Eq. (1), parameters of which are defined as:

$$m_l = \sum_i \sum_{v,v' \in V, v \neq v'} t_{v,v'} \cdot y^v_{N_i} \cdot y^{v'}_{N_{i+1}} \cdot \frac{b_r}{b_{v,v'}} \tag{2}$$

$$l = \sum_i \sum_{v,v' \in V, v \neq v'} t_{v,v'} \cdot y^v_{N_i} \cdot y^{v'}_{N_{i+1}} \tag{3}$$

$$m_r = \sum_{i=1}^{|r|-1} \sum_{v \in V_s} y^v_{N_i} \cdot y^v_{N_{i+1}} \cdot d^v_{N_i, N_{i+1}} \tag{4}$$

The optimization goal is divided into two parts: 1) In order to guarantee the quality of service for request, the link bandwidth utilization should be minimized, denoted as m_l in Eq.(2), where $t_{v,v'}$ indicates whether (v, v') is included in the optimal path p, $y^v_{N_i}$ indicates whether NF N_i is allocated to node v, and the bandwidth utilization ratio is obtained by dividing the data rate b_r by the remaining bandwidth $b(v, v')$ of link (v, v'). 2) Since resubmission operation(i.e., packets return to the beginning of current pipeline) is allowed in the system , which brings flexibility while affecting the throughput of both request and corresponding switch. Therefore, we added a second part to the optimization goal to weigh the impact of path stretching and resubmission. m_r, times of submissions on the entire service chain of request r, is calculated by Eq.(4), where $d^v_{N_i, N_{i+1}}$ indicates whether N_{i+1} is implemented before N_i on switch v.

In the optimization goal, metrics are respectively normalized (m_l divided by the total length of path in terms of links l in Eq.(3), m_r divided by the length of service chain) to evaluate them from the same dimension. Since m_l and m_r have different effects on performance under different network circumstances, we use two weights w_l and w_r to change the importance of these metrics in path construction.

$$t_{v,v'} \cdot b_r < b_{v,v'}, \quad \forall v, v' \in V, (v, v') \in E \tag{5}$$

$$\sum_i y^v_{N_i} \cdot m_i < c_v \quad \forall v \in V_s \tag{6}$$

$$\sum_i y^v_{N_i} \cdot p_i < c_v \quad \forall v \in V_h \tag{7}$$

Eq. (5) describes the traffic load constraints on each link v, v'. Eq. (6) and Eq. (7) state resource capacity on each switch v_s and each server v_h, where m_i and p_i denote memory and processing resource consumption of NF i respectively.

$$\sum_{v', \tilde{v} \in V_s, v \neq \tilde{v}} (t_{v,v'} \cdot t_{v,\tilde{v}} + t_{v',v} \cdot t_{\tilde{v},v}) = 0 \quad \forall v \in V \tag{8}$$

Eq. (8) defines an important problem of loops in path, i.e., traffic passing through the same switch twice, resulting in traffic routing conflicts. For example, as in Fig. **??**a, for a flow with path: s1 → s2 → s1 → s3, routing entries of traffic on s2 would conflict, leading to wrong destination.

It should be noted that the loop on switches directly connected with software-NF service installed server is not within the limit. Because it is easy to install a control logic on directly connected switch that identifies and tags the traffic from the server to distinguish and forwards traffic depending on the tag.

$$\sum_{v \in V} y^v_{N_i} = 1 \quad \forall i \in [1, |r|] \tag{9}$$

$$\sum_{v \in V} \mathrm{t}_{v_s,v} = 1 \quad \sum_{v \in V} t_{v,v_d} = 1 \tag{10}$$

$$y^v_{N_i} = 0, \quad \text{if} \quad x^v_{N_i} = 0 \tag{11}$$

$$\sum_{v' \in V_s} (t_{v,v'} + t_{v',v} = 2), \quad \text{if} \quad y^v_{N_i} = 1 \quad \forall v \in V \tag{12}$$

Eq. (9) requires that each NF is only installed once, while Eq. (10) defines the start and destination of path. Eq. (11) and Eq. (12) ensure the correctness of variable $y^v_{N_i}$ and $t_{v,v'}$.

In order to quickly respond to dynamic traffic, we need a heuristic algorithm for path construction. However, as a naive method, it is necessary to construct a multi-stage forwarding graph, and then use the shortest path algorithm to select the optimal path. Its time complexity is $O(N^2)$, growing with the total number of NF nodes in every stage, which is not conducive to rapid response to traffic. Therefore, we propose a two-stage algorithm to solve this problem.

B. Static Path Construction Algorithm

In order to reduce the workload of the dynamic algorithm, the candidate path set will be calculated from static algorithm stage as the input of dynamic algorithm. The static algorithm is only called when the topology is constructed or changed. Since realtime link load statistics cannot be obtained in static algorithm, we set the optimization goal in this stage as the weighted sum of path distance and resubmission times. The dynamic algorithm will evaluate paths according to link load ratio in realtime.

Algorithm 1: Static Path Construction Algorithm

```
Input: Network Topology Graph $G = (N, V, E)$,
       Request $R$
Output: NF-Forwarding Graph $g$ and Path Set $P$
Step 1: Construct Forward Graph $g.add(v_s)$
for $i \leftarrow 1$ to $|N_r| + 2$ do
    if $v \in N_i.nodes$ then
        $g.add(v)$;
        for $j \in N_{i-1}.nodes$ do
            $g[j][v] = w_l \cdot d[j][v] + m_r \cdot w_r$
        end
    end
end
Step 2: Get Candidate Path Set
$P' = KSP(graph, k * 2)$
for path $p \in P$ do
    if !p.containsLoop() then
        $P.add(p)$;
        if(P.size==k)    break;
    end
end
```

For all the requests defined in requirements file (introduced in section 3), we use the static path construction algorithm to select the top-k-shortest path as Algo. 1.

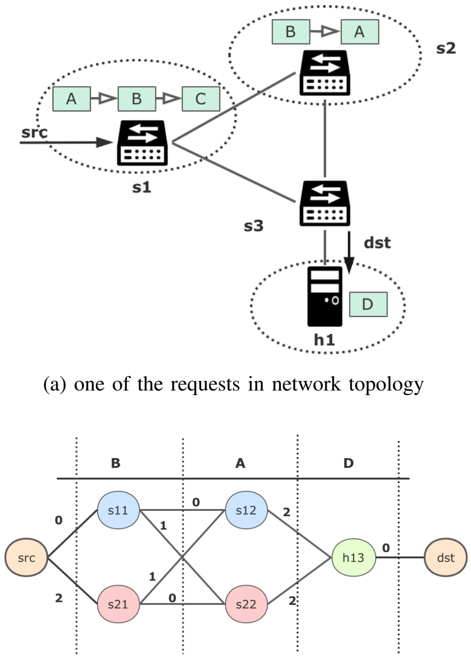
At first, an NF forwarding graph based on the topology graph and SFC request is constructed. The forwarding graph contains n + 2 stages (n is the length of SFC requirement). The first stage and the last stage are the source and destination nodes of the flow. The remaining nodes in each stage are the switches or hosts with corresponding NFs installed. The edge of the forwarding graph is assigned with the weighted sum of the distance of the shortest path between two nodes and resubmission times, which can be obtained by collecting network topology and request information. The procedure is presented in Fig. 5a, one of the request is depicted in the figure with the SFC request: B→A→D.

Next, we use the top-k-shortest path algorithm (i.e., Yen's algorithm) to obtain the set of the first k shortest paths. Since the path set obtained may contain routing loops, more paths than required are calculated from the KSP algorithm (i.e., set the parameter to $2k$), and then filter the paths in order. Paths that do not contain loops can be added candidate set, the algorithm stops when the size of candidate set reaches k.

C. Dynamic Path Selection Algorithm

When the first packet of each request is forwarded to the controller, dynamic path selection algorithm will be activated to select the optimal path from candidate sets considering cumulative link load.

As shown in Algorithm 2, the program will first read candidate path set from static algorithm. In order to avoid network congestion and further improve network performance, we set the metrics as cumulative sum of link load ratio (i.e., data rate divided by available bandwidth). Thus, for each candidate path, restrictions of node capacity and link capacity along the path is first check to guarantee feasibility. Then, an optimal path in terms of link load usage is selected after traversing candidate set. The time complexity of dynamic path selection algorithm is $O(n)$ in terms of number of candidate path, which is a tolerable delay for packets.



(a) one of the requests in network topology

(b) constructed graph by algorithm

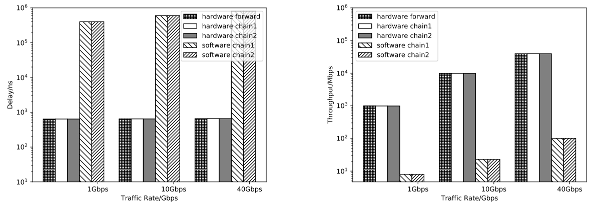
Fig. 5. Figure 5a is the network topology with hardware and software NF instances installed and figure 5b is the constructed forward graph for one of the request.

Algorithm 2: Dynamic Path Selection Algorithm

```
Input: Candidate Path Set P of Request R
Output: Optimal Path $p^\star$ of Request R
minimum metrics $min = MAXIMUM$;
for path $p \in P$ do
    for edge $e \in p$ do
        if $l[e] <= b_r$ or $c_{e.src} <= m_r$ then
            break;
        end
        if $m_l < min$ then
            $min = m_l$;
            $p^\star = p$
        end
    end
end
```

VI. Evaluation

We have evaluated the system on both a software (Bmv2) and a hardware (Barefoot Tofino) programmable data plane about performance and overheads of hardware offloading, as well as performance of two-stage NF mapping algorithm.



(a)

(b)

Fig. 6. Latency and throughput result of hardware performance test

A. Performance Benefits and Overheads of PDP

Experiment Setup. On the (Barefoot Tofino) target, we implemented the following NFs: NAT, Firewall, Load Balancer, Stateful Firewall, Port Knocking Firewall and Big Flow Detector. Due the limitation of hardware devices on stateful actions, we made a little modifications on stateful HNF models by combining reading, comparing and writing HNF state into a single register action, thus avoiding conflicts. To evaluate the performance of hardware HNF from common NFV solutions, we build two chains: stateful chain of Port Knocking Firewall → Stateful Firewall → Big Flow Detector and stateless chain NAT → Firewall → Load Balancer, and use Ixia Tester to generate packets and analyze collected information. Throughput and hardware overhead(caused by HNF) are considered to evaluate hardware offloading scheme. We also evaluated the impact of multiple flows (1 - 1000) and the extra costs caused by dynamic chaining framework. As for software platforms, we utilized OpenNetVM to collect needed information.

Performance.

Dealing with massive flow.

Hardware Overheads.

B. Analytical Evaluation of Control Plane Algorithm

Experiment Setup. We built the network topology with BMv2 switches on mininet with ONOS controller. We implemented the same HNFs as hardware experiment. The topology used is abilene in SDNlib with 12 nodes and 22 links, we also used the trace data set provided by SDNlib in the same topology, where traffic throughput of every node pairs are listed. We wrote the script to generate and receiver packets on different nodes, every flow in network is selected randomly from data set and is allocated with randomly selected service chain requirements. Flow completion time, throughput, link load and memory usage are informations collected to be evaluated. Implement the control plane(NF manage platform) based on ONOS controller with about 1000 LOC in JAVA. The control plane consists of modules: basic forward module, static algorithm module, dynamic algorithm module, NF rule translation module. We use Yen'algorithm[] to select k-shortest-path, and use the API provided by ONOS to collect metrics. In addition, we expands the ONOS framework to support the deployment of different P4 programs on different switches.

Bechmark.

FCT and Throughput Performance.

Bandwidth Resources.

Flow Entry Resources.

VII. Conclusion

Acknowledgment

References

[1] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Bianchi. Flowblaze: Stateful packet processing in hardware. *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019*, pages 531–547, 2019.

[2] Roberto Bifulco and Gábor Rétvári. A survey on the programmable data plane: Abstractions, architectures, and open problems. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–7. IEEE, 2018.

[3] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[4] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. Openstate: Programming platform-independent stateful openflow applications inside the switch. *Computer Communication Review*, 44(2):44–51, 2014.

[5] Chen Sun, Jun Bi, Haoxian Chen, Hongxin Hu, Zhilong Zheng, Shuyong Zhu, and Chenghui Wu. SDPA: Toward a Stateful Data Plane in Software-Defined Networking. *IEEE/ACM Transactions on Networking*, 25(6):3294–3308, 2017.

[6] Bojie Li, Kun Tan, Layong Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. ClickNP: Highly flexible and high performance network processing with reconfigurable hardware. *SIGCOMM 2016 - Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication*, pages 1–14, 2016.

[7] Chen Sun, Jun Bi, Zhilong Zheng, and Hongxin Hu. Hyper: A hybrid high-performance framework for network function virtualization. *IEEE Journal on Selected Areas in Communications*, 35(11):2490–2500, 2017.

[8] David Hancock and Jacobus Van der Merwe. Hyper4: Using p4 to virtualize the programmable data plane. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 35–49, 2016.

[9] Cheng Zhang, Jun Bi, Yu Zhou, and Jianping Wu. Hypervdp: High-performance virtualization of the programmable data plane. *IEEE Journal on Selected Areas in Communications*, 37(3):556–569, 2019.

[10] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. OpenNF: Enabling innovation in network function control. *Computer Communication Review*, 44(4):163–174, 2015.

[11] Tooska Dargahi, Alberto Caponi, Moreno Ambrosin, Giuseppe Bianchi, and Mauro Conti. A Survey on the Security of Stateful SDN Data Planes. *IEEE Communications Surveys and Tutorials*, 19(3):1701–1725, 2017.

[12] Katsuki Fujisawa, Masakazu Kojima, Kazuhide Nakata, and Makoto Yamashita. Sdpa (semidefinite programming algorithm) user's manual—version 6.2. 0. *Department of Mathematical and Com-puting Sciences, Tokyo Institute of Technology. Research Reports on Mathematical and Computing Sciences Series B: Operations Research*, 2002.

[13] Zhifeng Xu, Fangming Liu, Tao Wang, and Hong Xu. Demystifying the energy efficiency of network function virtualization. In *2016 IEEE/ACM 24th International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2016.

[14] Peng Zheng, Theophilus Benson, and Chengchen Hu. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, pages 98–111, 2018.

[4] [5] [6] [1] [11] [10] [5] [7] [3] [12] [7] [13] [2] [14] [9] [8]