SFP: Service Function Chain Provision on Programmable Switches for Cloud Tenants

Hongyi Huang
Tsinghua University
hhy.hongyi@outlook.com

Wenfei Wu
Peking University
wenfeiwu@pku.edu.cn

Yongchao He
Tsinghua University
heyc18@mails.tsinghua.edu.cn

Zehua Guo
Beijing Institute of Technology
guolizihao@hotmail.com

***Abstract*—Recent progress in programmable switches provides opportunities for service function chains (SFCs) provision to cloud tenants, which has the advantage of flexible deployment and high performance. We devise SFP for such SFC provision in the cloud. SFP's data plane installs physical NFs and is virtualized to host logical SFCs from multiple tenants. SFP's control plane uses a relaxed integer programming model to jointly optimize the placement of physical and logical NFs, which can achieve resource efficiency and high tenant traffic processing throughput within efficient execution time. Our prototype and evaluation shows that SFP can significantly offload NFV computation from server to the switch and maximize the switch resource utilization.**

I. Introduction

In modern clouds, network functions (NFs) undertake transparent traffic processing in the network for cloud tenants. They provide security or performance acceleration features, and thus, gradually become primitive services in clouds. A tenant could have multiple requirements in the traffic processing, e.g., load balancing and security, which would be satisfied by chaining multiple NFs in a sequence, called *service function chaining (SFC)*.

Many solutions implement NFs in software[1–6] instead of dedicated hardware boxes and prove their advantages of flexible deployment (copy and boot anywhere), easy asset management (storing software), and flexible traffic engineering (avoiding fixed in-between waypoints). However, the software-based solutions have to place NFs on commodity servers and NFs are suffering from problems of low efficiency (CPU v.s. ASIC), computation resource overhead (CPU and memory), distributed management (one SFC on multiple servers) and communication cost (packet transfer between servers).

A recent trend of deploying programmable switches in data centers [7–9] provides new opportunities for cloud providers to provide SFC. The programmable switch has memory to store states and programmable ASICs to apply programming abstractions with which a user specifies its own packet processing logic. Typical NFs have been proved implementable on programmable switches, such as load balancer, firewall, VPN, gateway, and rate limiters[10–14] in lightweight deployment.

We propose that *SFC can also be implemented on programmable switches*, which could inherit the advantages of flexibility from the software-based solutions, and also overcome the weakness of inefficiency, server resource cost, and distributed management.

We design a solution named SFP for cloud tenant SFC provision on programmable switches. SFP has a data plane and a control plane. The data plane is in charge of executing each tenant's SFC using shared programmable switch chips and the control plane is in charge of coordinating multiple tenants' SFCs efficiently concerned with limited switch resources.

In the data plane, NFs are pre-installed on the switch as *physical NFs* that are statically set up when the switch boots up. Tenant's SFCs are viewed as *logical NFs*, which can arrive and leave dynamically. When a logical SFC is installed to the physical ones, each logical NF's configuration is copied to the physical NF with an extra packet classifier to identify tenant traffic. When the SFC's NFs are in a different order with the ones on the switch, traffic would be recirculated for more than once so as to complement missing NFs in the previous pass. Such virtualization overcomes the logic/performance isolation of multi-tenant SFCs on a shared physical static switch pipeline.

In the control plane, SFP solves the problem of resource-efficient NF placement on switch pipelines. The NF placement is formulated as an Integer Program (IP), where the objective is to place more SFCs to the switch (so as to save more computation resources on servers) with the constraints on various resources (switch memory, stages, etc.) and the chain requirements. Significantly, SFP overcomes the challenge of formulating the joint placement of both physical NFs and logical NFs, which could find the optimal goal compared with placing them separately. We further relax the IP solutions to linear programming with random rounding, which can be solved in polynomial time.

We prototype SFP and conduct experiments on testbed and by simulation. The evaluation shows that SFP's data plane can serve SFC with high throughput and low latency (around 340ns), and the virtualization mechanism makes the physical pipeline successfully host multiple logical SFCs. Moreover, the SFC to pipeline placement algorithm could significantly save computation resources and be run in acceptable time. In this paper, we make the following contributions.

1) a switch data plane that can load static physical NFs and host diverse dynamic SFCs,
2) a control plane that jointly optimizes the physical and logical NF placement, achieving efficient switch re-

Wenfei Wu is the corresponding author.

source usage and significant offloading,

3) a prototype and corresponding experiments to show SFP's good properties of offloading, scalability, and runtime update support.

II. Background

A. Motivation and Goal

Opportunities from Programmable Switches. The progress in programmable switch gives new opportunities to SFC provision. The programmable switches have memory to store persistent states (whose life time is longer than each individual packet) and can load user-specified packet processing logic. Thus, several typical NFs can be implemented on programmable switches. Recently, researchers have proposed NFs like load balancer, rate limiter, cache, firewall [10–13, 15].

The packet processing circuit on the switch is a pipeline of several stages (12 physical stages in Tofino). Each stage is composed of necessary resources to implement processing logics, i.e., match-action units (MAUs). The packet processing program is organized similarly, with a match field to classify packets and an action field to process the packet and the switch memory state. Such organization format is analogous to the behavior model presented in NF development[16]. So it shows the capacity to migrate more NFs from servers to switches.

Based on the switch hardware, P4 language[17] comes up with platform independent abstractions to describe core processing logic of these functions and some other work provides opportunities to combine these P4-based implementations to generate SFCs on switches[18, 19].

Advantages of Switch-based SFC Provision. Compared with the hardware-based SFC provision solutions and the software-based ones, switch-based solutions have their unique advantages.

Hardware-based solutions usually have the complexity of deployment, asset management, and load balancing. Because a hardware device can only be placed in the fixed location in the network and all traffic must be routed to that fixed point. Software-based solutions can place an NF on any server, and the operation is to copy the software and boot it up, simpler than hardware (re)wiring. Switch-based solutions can inherit this flexibility because all servers are attached to Top-Of-Rack (TOR) switches, and these TOR switches (as well as other switches) provide equal flexibility in NF deployment than software NFs on servers.

Software-based solutions usually need to consume computation resources (CPU, memory, bus, and network) on servers to support the packet processing. While in clouds, these resources should have been sold to customers to gain revenue. To make things worse, the computation capability of servers is usually two orders of magnitude smaller than hardware ASICs (application specific integrated circuits)[20], causing more significant resource cost.

The recent programmable switches have computation circuit that could process packets at the speed of several tera-bits per second (e.g., 3.2Tbps backplane speed in Tofino switch),

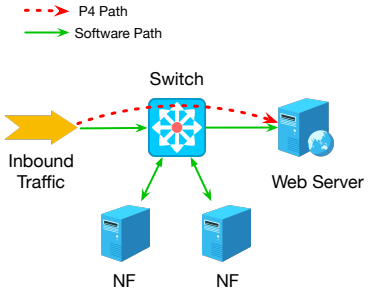


Fig. 1: NF Processing Traffic

and offloading packet processing to the switch is promising to avoid the server resource overhead. As Fig. 1 shows, software-based solutions will direct inbound traffic between switch and NF servers and finally send to the web server. In contrast, P4-based solutions will mainly go through the switch, which reduces latency and ensures high throughput due to high processing capability of the switch.

Goal. In this paper, our goal is to build *an SFC provision solution for cloud tenants*. We name our system SFP. SFP is expected to satisfy the following requirements.

- **R1: Supporting SFC Deployment.** While individual NFs have been invented by the research and industry community, SFP needs to devise the SFC placement mechanism, including placing each NF and direct traffic through them. (Fig. 1)
- **R2: Supporting Multi-tenancy.** The SFCs vary from different tenants. They all need to be placed on the same physical pipeline. More importantly, different tenants should maintain isolation in terms of traffic, performance, resource, and NF behavior.
- **R3: Supporting Dynamic SFC Deployment.** Tenants may join and leave, and their SFC should be correspondingly allocated and deallocated.
- **R4: Performance.** SFP should serve tenant traffic with the same (or better) performance in terms of throughput and latency.
- **R5: Low overhead.** SFP should be able to support reasonable tenants' SFCs within the current switch resources.

B. Preliminaries of P4-based NFs

To deploy software NFs in programmable P4 switches, we first introduce the basic components for composing a basic P4 program.

Structures of a P4 Program. As aforementioned, a P4 program is currently written in packet-driven forwarding model, which records how the inbound packets are processed by the pipeline (ingress and egress) in the format of match-actions tables (MATs). Each packet is decomposed to custom-defined header fields and payload with additional fields to store per-packet metadata. Match-action table (MAT) would perform lookups of header fields and apply the user-defined actions

```
control ingress {
    if (valid(tcp) or valid(udp)) {
        apply(tab_firewall);        // firewall
        apply(tab_classifier);      // traffic classifier
        apply(tab_lb) {             // load balancer
            miss {
                apply(tab_lbhash);
                apply(tab_lbselect);
            }
        }
        apply(tab_router);          // router
    }
}
```

Fig. 2: Control Flow of SFC Example

mainly to packet headers. The interconnections between MATs are organized by the control flow primitives.

P4-based NF/SFC Example. Some work[10–13, 21] has justified the feasibility to deploy various types of NFs in P4 switches. To apply an SFC in P4 switches, we should chain individual NF MATs in the program since P4 switches could only run one program at one time. Fortunately, P4 primitives naturally support flexible control flow. Assume we have an SFC containing four sequential NFs – firewall, traffic classifier, load balancer, and router. Fig. 2 shows how we chain them together in one P4 program. As shown, only tcp or udp packets will be processed by the SFC, and the NFs are applied in sequence in the ingress pipeline. Looking at the NFs, the firewall, the traffic classifier and the router are all described using one MAT, whereas the load balancer contains two more MATs. 'tab_lb' reads header fields and looks up the IP address of the server that users specify in rules. Otherwise, if no rule is matched, it will calculate the hash value through table 'tab_lbhash' and determine the IP address from address pools through table 'tab_lbseclect'.

Applying P4 Programs to Switch Pipelines. As aforementioned, a P4 program is logically composed of several match-action tables. To install these tables in switch chips, P4-supported switch vendors enable the programmability atop traditional packet processing pipelines, which best conforms to the structure of a P4 program. In detail, each physical stage of the pipeline is designed as Match-Action Units (MAUs) containing dedicated hardware resources, mainly including SRAM and TCAM to store the match-action tables or states and ALUs to execute the actions. To ensure resource efficiency, the switch is likely to put down multiple orthogonal match-actions tables within one MAU if those tables have no read or write dependency to each other. Otherwise, tables would be applied to continuous physical stages by default. For flexibility, the switch vendors provide APIs for users to customize table-stage affiliations. `if-else` conditions within the program are performed by gateway tables from MAUs so that different conditions can index separate match-action tables to apply.

III. Overview

Assumptions. In the design, we make the following assumptions. First, tenant traffic can be classified by header fields. In typical virtual network construction, tenants' traffic is isolated by protocols such as VLAN, VxLAN, GRE, etc., or even IP subnet, and these protocol headers can be parsed and recognized by programmable switches. In this paper, we uniformly call these header fields `tenant ID`. Second, the types of NFs are fixed, and there is a limited number of them. As a service from the provider to the tenants, the provider predefines a few NFs, and the tenants make selection, chaining, and configurations. While the types of NFs can increase with long-term evolvement, we regard the types of NFs at each deployment cycle (a few days) to be static and fixed.

Architecture. SFP consists of a data plane and a control plane. The data plane is in charge of running SFCs. The data plane would pre-install a few NFs of existing types, called physical NFs, each on one stage. If one NF spans multiple stages, it is viewed as several sub-NFs. Each physical NF is almost the same as the P4-based NF program above, except that the physical NFs have a tenant traffic classifier before its NF logic. When a logical NF from SFC is offloaded to the physical one, the NF configuration (typical switch rules) is copied with tenant ID match at the beginning of each NF rule. **(Challenge 1)** The data plane overcomes the challenge of placing diverse dynamic SFCs onto the static physical pipelines, where a logical SFC may have a different order from the physical NF order. Once such SFC needs to be installed, the tenant traffic would be guided to traverse the physical pipeline multiple times (by the last hop of each pass recirculating the traffic).

The control plane of SFP solves the problem of placing physical NFs on stages and placing logical SFCs onto the physical pipeline. It uses integer programming (IP) to formulate the placement and resource constraints with the objective of offloading more tenant traffic. SFP also relaxes the IP problem to linear programming, which outputs approximate results with higher efficiency. The control plane placement algorithm additionally supports dynamic SFC runtime update, i.e., deleting inactive tenant SFCs and installing new ones to the optimal location without rebooting the device. **(Challenge 2)** The control plane overcomes the challenge of jointly placing the physical NFs and the logical NFs. It solves this challenge by introducing placement variables and carefully justifying the placement affiliation between variables within the IP model.

IV. SFC Virtualization in Data Plane

Install Physical NFs. SFP pre-installs NFs on the switch pipeline. These NFs are called physical NFs, and in §V we would elaborate the selection of physical NFs. Each NF is placed in one stage and reserves a piece of switch resource (e.g., register, TCAM) which can host runtime states and rules. Each NF is implemented almost the same as the P4 implementation above, except for two changes. First, each physical NF has a default rule "No-Ops", not processing packets but forwarding them to the next stage. Second, each physical NF's match block is added with two fields, one to

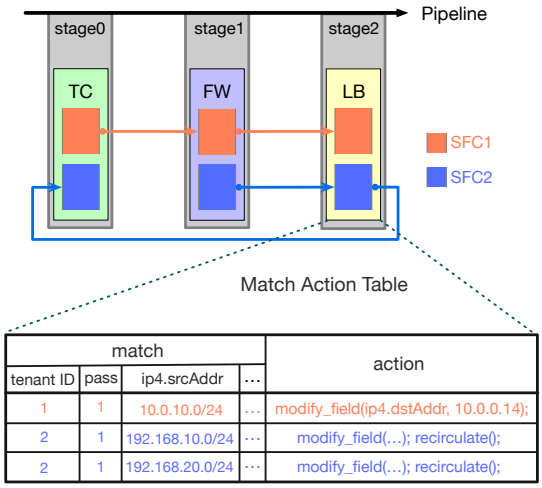


Fig. 3: A Toy Example of SFC Placement in a 3-stage Switch

match `tenant ID` and one to match recirculation `pass`. The `pass` is in the packet metadata.

NFs in the last stage is specially crafted. All their actions are added with an extra argument — `REC`, and the action is appended with two pieces of logic: (1) recirculate (or not) the packet according to the argument, and (2) increase the `pass` by one.

(De)allocate Logical NFs. NFs in an SFC are viewed as logical NFs. When allocating an SFC to the physical pipeline, its NFs are sequentially assigned to physical NFs; if one pass of the pipeline cannot accommodate all NFs (due to SFC length or NF orders), the SFC is "folded" and get into the pipeline in the next pass.

In details of logical-to-physical assignment, SFP maintains a variable to record the pass traversing the pipeline, named `currPass`. Starting from the first NFs in the SFC and the first stage in the pipeline, SFP find a matching pair in SFC and the pipeline with the same NF type, and copied the logical NFs' configuration to the physical NF with two extra changes in the match block: matching the `tenant ID` with the SFC's owner and `pass` with the `currPass`. If the SFC cannot be completely allocated to the pipeline, the last allocated NF in the current pass are configured with the `REC` argument in its rules to be set; and `currPass` is increased by one, and SFP proceeds to the next pass to assign the remaining SFC NFs from the beginning of the pipeline.

When a tenant leaves the system, all its SFC NF rules are deleted from the switch, and the switch resources (rule entries, memory, and backplane processing rate) are released, and new SFCs can be allocated. (More details are stated in §V-E.)

Example. As an example, Fig. 3 shows that the current physical pipeline (with 3 stages) has been set up with several physical NFs, each in a stage and of a specific type (e.g., firewall (FW), traffic classifier (TC), load balancer (LB)). SFC 1, with NFs following the order of *TC, FW, LB*, can be sequentially applied to the physical pipeline. But SFC 2, with NFs of *FW, LB, TC* needs to be assigned in two passes, in the first pass, *FW, LB* can be assigned, and in the second pass, *TC* is further assigned; *LB* is the last hop of the first pass and is configured to recirculate the packet. In the match-action table of the LB in SFC 2, the "tenant ID" shows the virtualization of one physical NF hosting multiple logical ones, and the "recirculation" in the action is used by the last hop to redirect the packet to the beginning of the pipeline.

TABLE I: Symbols Used in Model

Symbol	Type	Explanation
i/I	index/const	the i-th/total number of NF type(s)
j/J_l	index/const	the j-th/total number of NFs in an SFC (l)
k/K	index/const	the k-th/total number of stages in the switch pipeline
l/L	index/const	the l-th/total number of SFCs
x_{ik}	var	indicates whether network function of type i is implemented in the k-th stage
z_{ijkl}	var	indicates whether the j-th function box of service chain l is of type i and implemented on stage k
d_{jl}	der. var	indicates whether j-th box of chain l is deployed
g_{jl}	derived var	the stage index where the j-th box in chain l is deployed
s_l	derived var	indicates the maximal stage where the chain l is implemented, which is also the stage of the last box of chain l.
R_l	der. var	recirculation times for chain l
b	const	bit width of each rule
B	const	number of memory blocks available in each stage
C	const	bandwidth capacity of the switch chip
E	const	memory in a switch is assigned on a block basis with uniform size of E
S	const	number of physical stages in the switch
f_{jl}	const	the type of j-th function box in service chain l
F_{jl}	const	the number of rules of box j in chain l
T_l	const	the bandwidth requirement for service chain l

V. SFC Placement in Control Plane

The control plane is in charge of placing multiple SFCs into the physical pipeline. A proper placement algorithm would improve the resource utilization of the switch.

A. Problem Formulation and Solution

Mapping a logical graph/chain to a physical topology has been widely studied in networking [5, 22], and the methodology has been used in the management of public cloud, big data, and machine learning.

SFP solves a more complicated scenario: the allocation is two-level. The physical NFs need to be pre-allocated to the pipeline, and the logical NFs need to be allocated to the physical NFs. If the two-level allocation is considered separately, it is challenging to guarantee global optimality. SFP formulates the NF placement problem as an Integer Programming problem, with special constraints to consistently allocate both physical NFs and logical NFs. The symbols and notations in the formulation are in Table I.

Physical Pipeline. There are S stages in physical pipeline. Each stage has the memory of size B. Physical NFs need to be pre-allocated to the pipeline before logical SFCs. We assume each physical NF would reserve a piece of memory, which

can install rules of size b in the MAU; and thus, there are B/b entries available in each NF.

Logical SFC. Each NF has a type (e.g, load balancer, firewall), and types are indexed by i ranging in $[1, I]$. Each SFC is modeled as a tuple of traffic and a list of NFs. For the l-th SFC, the traffic is denoted as T_l, and the j-th NF on the chain l is of type f_{jl} and is configured with F_{jl} entries.

Formulating Recirculation. Assume the traffic can be recirculated several times, we would alternatively view the pipeline as a "virtual" pipeline of K stages (K is a multiple of S), with the constraints of the j-th stage and the $(j+S)$-th stage shared the resource and NF placement constraints $(1 \leq j, (j+S) \leq K)$. The indices of stages in the following description are in terms of the location in the virtual pipeline. In §VI, we would describe how to decide the recirculation times.

Variables for Placement. We use x_{ik} to indicate a type-i NF is implemented on stage k, and z_{ijkl} to indicate whether the j-th NF on the l-th SFC is of type i "and" placed in stage k.

$$x_{ik} = \begin{cases} 1, & \text{if NF of type } i \text{ is implemented in stage } k, \\ 0, & \text{otherwise.} \end{cases}$$

$$z_{ijkl} = \begin{cases} 1, & \text{if SFC } l\text{'s } j\text{-th box is of type } i \\ & \text{and allocated on stage } k, \\ 0, & \text{otherwise.} \end{cases}$$

We derive a few variables to simplify the description. d_{jl} describes whether the j-th NF of chain l is deployed:

$$d_{jl} = \sum_{i,k} z_{ijkl}, \forall j, l.$$

g_{jl} indicates the stage index where the j-th NF of the l-th SFC is placed:

$$g_{jl} = \sum_{i,k} z_{ijkl} \times k.$$

The largest stage index s_l of an SFC l is the stage index of last box on the chain.

$$s_l = g_{j=J_l,l}.$$

The traffic on the l-th SFC would be recirculated for R_l times:

$$R_l = \lceil \frac{s_l}{S_{max}} \rceil - 1, \forall l.$$

Note that if an SFC is not placed on the pipeline, s_l would be 0 and R_l would be -1.

Problem Formalization. Using the aforementioned denotations, we formalize SFP placement as an optimization problem whose objective is to offload as much traffic processing as possible, which is proportional to the traffic $d_{j=*,l}$ and the length of the chain J_l.

Objective:

$$\max \sum_{l} d_{j=*,l} \times T_l \times J_l. \tag{1}$$

Subject to:

$$x_{ik} \in \{0,1\}, \forall i, k \tag{2}$$

$$z_{ijkl} \in \{0,1\}, \forall l, k, i, j \tag{3}$$

$$\sum_{k} x_{ik} \geq 1, \forall i \tag{4}$$

$$d_{jl} \leq 1, \forall j, l \tag{5}$$

$$\sum_{i,k} z_{ijkl} \times i = f_{jl} \times d_{jl}, \forall j, l \tag{6}$$

$$d_{j=1,l} = d_{j=2,l} = \cdots = d_{j=J_l,l}, \forall l \tag{7}$$

$$g_{j=1,l} \leq g_{j=2,l} - d_{jl} \leq \cdots \leq g_{j=J_l,l} - d_{jl} * (J_l - 1) \tag{8}$$

$$z_{ijkl} \leq x_{ik}, \forall i, j, k, l \tag{9}$$

$$x_{ik} = x_{i,k+S}, \forall i, 1 \leq k, k+S \leq K \tag{10}$$

$$\sum_{i} \left\lceil \sum_{k=s+nS}^{K} \sum_{j,l} (z_{ijkl} \times F_{jl} \times b)/E \right\rceil \leq B \tag{11}$$

$$\sum_{l} (R_l + 1) \times T_l \leq C \tag{12}$$

We summarize the constraints as follows and elaborate the details in the Appendix:

- placement constraints that describe correct and proper multiple-SFC placement. (2), (3), (4) and (5) together determine the physical/logical NF placement. (6), (7) and (8) ensure the placement compliance with NF type and order.
- consistency constraints including (9) to ensure consistency between logical and physical placement and (10) to ensure consistency between virtual and physical stages.
- memory constraint (11) that describes the coloration between memory utilization and hardware limitations.
- processing capacity constraint (12) that describes both inter-SFC and intra-SFC contention in switch backplane bandwidth.

Linearization. The Equation (11) above contains rounding functions. However, rounding functions are not well dealt with in IP solver. Therefore, we transform them into linear constraints. For an expression $\lceil exp \rceil$, we define a new integer variable $Y = \lceil exp \rceil$ and add constraints $Y \in Z$ and $Y - 1 < exp \leq Y$. In SFP, this transformation adds $(i \times S_{max} + j)$ variables. Since IP cannot allow strict inequalities, we further derive the constraints to $Y \in Z$ and $Y - 1 + \epsilon \leq exp \leq Y$ for $\epsilon \to 0^+$ by adding a very small positive number.

B. Approximation

Integer programming is an NP-hard problem, and current solvers cannot find the solution quickly once the problem scale is too large. We use randomized rounding method to find approximate solutions efficiently. [1]

[1]While the SFC placement problem is a subset of the IP problem, it is still NP hard. In one special scenario, all SFCs are of length one, and the physical pipeline has two stages with the same memory space; placing the SFC is equivalent to a bin-packing problem, which is NP hard. Thus, the SFC placement problem is NP hard.

Algorithm 1 SFC Placement with Approximation

Input: constants in TABLE I, recirculation times R
Output: variables x, z

$r = 0$;
for $r \leq R$ **do**
  $Relax\_vars()$;
  $LP()$;
  while not verified **do**
    $Round\_vars()$;
    $x', z' = Verify\_vars()$;
  end while
  if result is optimal **then**
    $x = x'; z = z'$;
  end if
  $r = r + 1$;
end for

First, the integer programming problem (IP) can be relaxed to linear programming (LP), where the integer variables are relaxed to real numbers in the same range. And the LP problem is solved with the variable in real number domain.

Second, the integer variables in IP (which are real numbers in LP) are rounded to nearby integers with probability. For example, a value of $X.Y$ (X is integer and Y is decimal) is rounded to X with probability $1 - Y$ and to $X + 1$ with probability Y.

Finally, the rounded variables are put to the original IP problems to check with the constraints. If the rounded variables cannot satisfy the constraints, the algorithm will strip one SFC that requires most resource but least bandwidth (Equation 13) and go back to the second step for another trial; otherwise, rounded results are output. Randomized rounding can guarantee the expected result of each rounding step to be the optimal (i.e., E(Objective with rounded variable) = Objective of LP).

C. Overall Algorithm

Algorithm 1 shows the overall algorithm of SFC placement. The algorithm tries different recirculation times within a reasonable range (we tried 0 to R). Each trail formulates the IP problem as in §V-A, relaxes the variables in function $Relax\_vars()$, and solves it with LP relaxation in function $LP()$. After getting all real value for the variables, we need to start a new loop to round the variables from real numbers to integers (function $Round\_vars()$) until the rounding result is verified to satisfy the constraints (function $Verify\_vars()$) and get the result. The optimal result is selected between trials as the final output.

D. Greedy Algorithm as Baseline

We construct another heuristic algorithm as a baseline in the evaluation. The heuristic algorithm is a greedy algorithm that can execute quickly. Intuitively, we prefer offloading SFCs whose throughput (objective) is high but resource occupancy is small (less cost). Therefore, we define a metric for each SFC:

$$Metric = \frac{T_l}{\sum J_l \times F_{jl}}. \tag{13}$$

Algorithm 2 Greedy Algorithm

Input: constants in TABLE I, recirculation times R
Output: variables x, z

$ordered\_sfcs = Order\_SFCs()$;
for $sfc \in ordered\_sfcs$ **do**
  $Try\_placement()$;
  $Resource\_recompute()$;
end for

The outline of greedy algorithm is shown in Algorithm 2. It first sorts all SFC candidates with the metric in Equation 13. SFCs with higher metric have priority to try the placement (function $Try\_placement()$). In detail, given an SFC, the algorithm would place the NFs in the chain one by one. To install one NF, the algorithm always finds the nearest next physical NFs with enough resource capability. If failed, it will require new physical NF installation in the nearest next stages (if the resource allows) and then settle down the NF. If the placement succeeds, function $Resource\_recompute()$ is invoked to compute the remaining resources. The greedy algorithm cannot always get an optimal replacement since it merely reflects on the total number of rules/entries without considering larger search spaces such as NF rule distribution.

E. Runtime Update

SFCs can dynamically arrive and leave. When SFCs arrive/leave, SFP should modify the data plane (i.e., the rules in match-action tables) to reconcile all SFCs in the switch. In detail, SFP deletes old SFCs by removing their rules and installs new ones by adding their rules to the corresponding tables assigned by the control plane. If new global `tenant IDs` are assigned to existing SFCs, the rules belonging to those affected ones should be modified as well.

To target the runtime update, the control plane needs to update the switch resource usage when SFC leaves, maintain the SFCs who do not leave in previous placement, and try to place new SFCs to the pipeline with newly released resources.

The new SFC placement uses the same algorithm above, with the updated network topology and resource availability. The result may not be global optimal since it only considers incremental SFCs. To this end, we can set up a threshold. Once the distance between the current configuration and the optimal one exceeds the threshold, the whole SFCs and pipeline would be automatically re-configured. In such case, extensive changes of the rules are desired to apply in the data plane, which incurs high overhead or possibly a reboot of the switch.

If the tenants require to adjust their SFCs (e.g., the processing order or the type of the internal NFs), we regard them as SFC departure and arrival, and apply the aforementioned principles.

VI. Evaluation

Our evaluation of SFP answers the following key questions:

1) To what extent can SFP improve the SFC processing throughput and latency compared to software SFC deployed in servers with DPDK? (§VI-B)

2) What throughput and resource utilization can SFP achieve in various SFC provision scenarios? Would the recirculation of SFCs improve the performance or incur latency overhead? (§VI-C)
3) What if we attempt to early terminate SFP IP solver rather than waiting for a full stop? How is the intermediate result approaching the optimal solution? (§VI-C)
4) How much performance loss would linear relaxation of SFP and greedy algorithm cause compared to IP approach? (§VI-C)
5) How well does runtime update work? What impact would it have on the objective given that we replace partial SFCs? (§VI-D)

A. Experiment Settings

Implementation. We implement SFP control plane placement solver in Python, involving 1.5k lines of codes. We use a well-known powerful optimization solver Gurobi[23] to handle integer and linear programming problems in our algorithms. We implement 4 NF programs (firewall, load balancer, traffic classifier, and router) in P4 language[24] as a proof-of-concept.

Testbed. Our testbed consists of a 32-port Tofino switch, each port with 100Gbps bandwidth and three mid-range servers (4 Intel(R) Xeon(R) Gold 5120T CPU @ 2.20GHz 16 cores, 192RAM with Ubuntu 18.04, 100Gbps Mellanox ConnectX-5 NICs, 22TB disk) that run the client, server and SFCs respectively.

Dataset. We synthesize the SFC dataset to evaluate SFP control plane algorithms. We observe that a top-of-rack switch that connects a rack of servers that could have tens of SFCs. Other principal parameters are implied by some work[25, 26]. We add randomization to generate five sets of SFC data for each experiment in §VI-C. In detail, each SFC randomly chooses different NFs to compose the chain, and the number of rules for each NF uniformly ranges from 100 to 2100; the bandwidth requirement of each NF follows the long-tail distribution. In each experiment, we use synthetic traffic workload and trace[27] to measure the performance.

Baseline. In the data plane, we have SFP's P4-based implementation comparable to the state-of-the-art software solutions[1]. And we choose four basic NFs(firewall, load balancer, traffic classifier, and router) with DPDK acceleration as the baseline. In the control plane, we compare several placement algorithms: SFP, SFP without consolidation (i.e., using Equation 12 instead of 11), ILP, and greedy algorithm.

Metrics. In each experiment, we measure the throughput and the latency to show SFP's QoS. And we measure the execution time of SFP algorithms to evaluate their efficiency.

B. SFC Performance and Overhead

We deploy 4 NFs on both P4 switch and server, and send traffic at 100Gbps, with packet size varying from 64 to 1500 Bytes that cover most packet size[27]. In the DPDK baseline, 16 (of 56) CPU cores are used to run the client, SFCs, and the receiver, and DPDK master uses one extra core. In each DPDK experiment, the average memory utilization of SFC is 722 MB, and the CPU utilization is 30.35% (17/56).

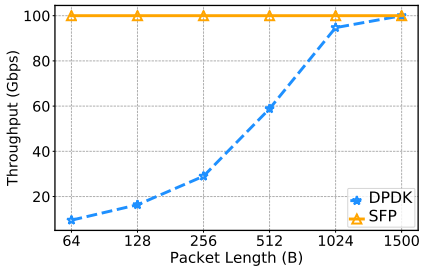


Fig. 4: Throughput Comparison between SFP and Software SFC Deployment

Fig. 4 shows the processing throughput of SFP and DPDK-based SFC. SFP always outperforms DPDK for packets of any size(from 64B to 1500B). SFP keeps saturating the 100Gbps bandwidth bound by the sender, whereas DPDK method can reach 100Gbps only when the packet size is increased large enough as 1500B or more. In the worst case of 64 Byte packets, SFP performs at least 10 times better than DPDK in terms of packet processing rate.

Fig. 5 shows the processing latency of SFP and DPDK SFC. With various packet sizes, the average latency of SFP and DPDK is 341ns and 1151ns respectively. SFP is 0.3X to DPDK. In practical systems, SFP would have more significant advantages in RTT because SFP processes traffic "on-path" in the switch instead of routing the packets to NF servers with more hops (shown in Fig. 1).

C. SFC Placement Performance

We configure the switch with 8 stages and 20 memory blocks (each for an NF) in each stage, and each block has 1000 entries of rules. We configure the switch backplane speed to be 400Gbps in the control plane model.

Impact of the Number of SFCs. We set the types of NFs to be 10, the average chain length to be 5, and the maximum recirculation time to be 3. We tune the number of SFCs L from 10 to 50, and measure the throughput, block utilization, and entry utilization of SFP and SFP without consolidation ('Baseline'). Fig. 6a shows block utilization and throughput varying in L. (1) With a small number of SFCs (about 15), blocks in both algorithms are quickly occupied. With L increasing, the block utilization is close to the upper bound of 20 and does not show a significant increase. (2) But the throughput increases proportionally with L because larger L allows for more candidates to choose to place on the switch, and SFCs with higher throughput would be finally selected. (3) SFP always has slightly higher throughput than SFP without consolidation but may consume a little more blocks, e.g., when $L = 30$, the throughput is 247.1 Gbps v.s. 227.0 Gbps, and the block utilization is 20 and 19.7 for both algorithms.

Observing the entry utilization in Fig. 6, SFP without NF consolidation has a lower entry utilization because there are internal fragments within each block. Such under-utilization of

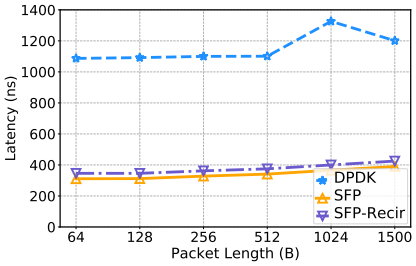
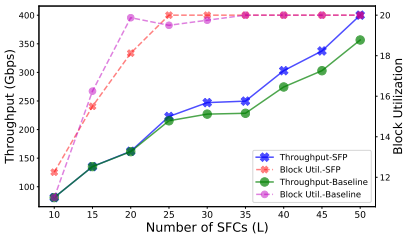
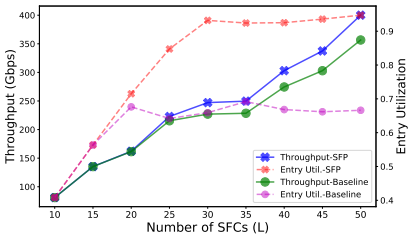


Fig. 5: Processing Latency of SFP and Software SFC

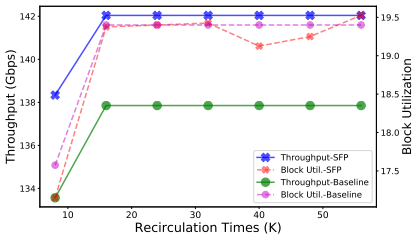


(a) Block Utilization and Throughput

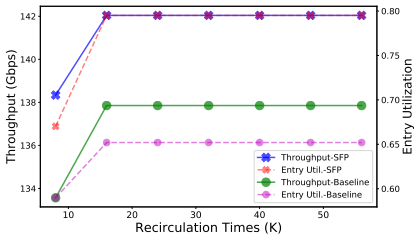


(b) Entry Utilization and Throughput

Fig. 6: Throughput and Resource Utilization Varying the Number of SFC Candidates



(a) Block Utilization and Throughput



(b) Entry Utilization and Throughput

Fig. 7: Throughput and Resource Utilization Varying the Recirculation Times

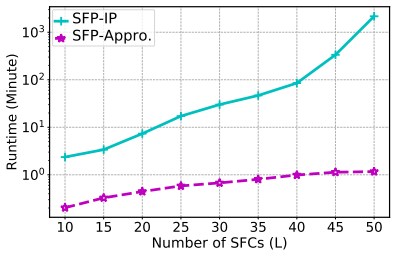


Fig. 8: SFP Runtime Varying the Number of SFCs

switch memory leads to a slightly lower throughput compared with SFP.

Impact of Recirculation Times. We set the number of candidate SFC to 15[2], each with a length of 8 NFs chosen randomly from 10 NF types. We tune the number of stages in the virtual pipeline, from 8 to 56 (i.e., 0 to 6 times of recirculation). As shown in Fig. 7, we observe that (1) one time of recirculation can improve the throughput (i.e., from 138.3 Gbps and 133.6 Gbps to 142.0 Gbps and 137.6 Gbps for both algorithms), but more than one recirculation cannot further improve the throughput, because the one-time recirculation has already enlarged the search space of candidate SFCs enough to saturate the backplane capacity. (2) Block utilization is roughly the same for both algorithms, but SFP achieves better entry utilization similar to Fig. 6.

However, allowing recirculation could raise some concerns about the processing latency. To manifest the impact on latency, we use 'recirculate' primitive in P4 to redirect the packets to go through the ingress pipelines repeatedly. In each pipeline pass-through, we apply only one NF to process the packets so that the whole packet processing logic is the same as in §VI-B. We measure the processing latency and show the result as 'SFP-Recir' in Fig. 5. It shows that compared with SFP, three recirculations within 4-NF SFC only incur 35 ns overhead. And we conclude that the processing latency is substantially dependent on the processing complexity of the whole SFC but not the recirculation times.

Comparison between Placement Algorithms. Fig. 8 shows the execution time of SFP integer programming ('SFP-IP') and SFP IP with linear relaxation ('SFP-Appro.'). The switch has 8 stages, the recirculation time is allowed to be 2, and the average chain length is 5. We tune the number of SFCs and measure the execution time that both algorithms need. We observe that the SFP-IP approach shows a trend of increasing faster than exponentiation, and SFP-Appro. (with relaxation) shows a polynomial time increase where finding a solution for 50 SFCs only takes 70s.

Fig. 9 shows the throughput and resource utilization of SFP-IP, and we can tune the solver to early terminate the computation with the current optimal solution. There are 25 SFCs in the experiment. With the shortest runtime limit (5s), the algorithm cannot get any solutions and its performance is 0. Given a little more time (10s), the throughput approaches approximately optimal and keeps going up slightly as the time lasts, and the throughput reaches the optimal threshold at 30s. Compared with SFP-Appro. that gets the result within 35.0 seconds, using early termination in IP is another possible approach to efficiently get a near-optimal result.

Fig. 10 shows the objective throughput of SFP-IP, SFP-Appro., and the greedy algorithm. In the setting of 8 stages, 2 recirculation times, 10 NF types, and average chain length of 5, the SFP-IP almost saturates the switch capacity with about 50 SFCs. The SFP-Appro. and the greedy algorithm show the same increasing trend when SFC is fewer than 40, but cannot show a saturation compared to the IP approach (e.g., 398 Gbps v.s. 377 Gbps v.s. 367 Gbps for 60 SFCs). And SFP-Appro. outperforms the greedy algorithm. In practical use, SFP-Appro. is efficient enough to output a near-optimal solution, and greedy algorithm can be considered when the scenario needs prompt deployment but less optimality.

[2]We apply small number of candidate SFC to eliminate the contention between SFCs and manifest the impact of recirculation times.

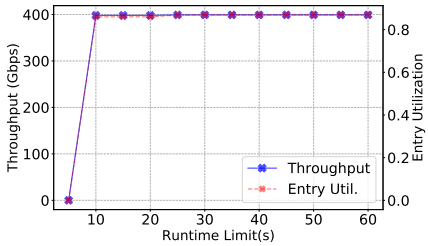


Fig. 9: Throughput and Resource Utilization Varying Runtime Limit

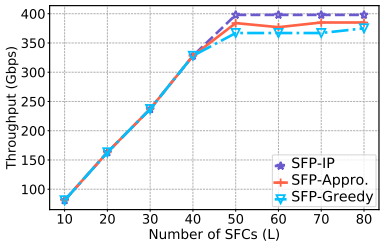


Fig. 10: Throughput of Greedy Algorithm Compared with IP

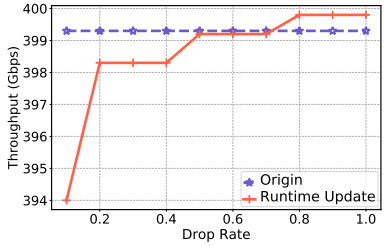


Fig. 11: Throughput of Runtime Update Compared to That Before Update

D. Runtime Update

We run experiments with 8 stages, 2 recirculation times, 5 average chain length, 10 NF types, 20 allocated SFCs and 50 SFC candidates. We allocate SFCs first, and then drop the allocated ones with a drop rate, and choose SFCs from candidates to fill in the switch pipeline again. Fig. 11 shows the throughput after the update compared with the ones before the update ('Origin'). We observe that the throughput has a very slight increase with higher drop rate, i.e., from 394.0 with 0.1 drop rate to 399.8 with 1.0 drop rate. The high saturation results from the extensive possible combinations from 50 SFCs, and a nearly saturated result always exists. The reason for the slight increase is that dropping more old SFCs makes more available resources for new SFC combination/selection, and the more candidate combinations that can fill in the free resource can lead the algorithm to find the one with higher saturation.

VII. Discussion

We discuss the scope of this paper as follows.

Offloadability. It is reported that most NFs are offloadable in programmable switches[21]. The others that are not currently available in switch pipelines will be deployed in COTS servers as VNFs, completing the functionalities of SFCs. However, programmable switches would entail much more resource contention compared to commodity servers that are prevalent in the cloud.

Branches inside SFC. Using P4 programming abstractions, NFs are implemented in match-action tables and might be organized by if-else control flow as directed acyclic graph (DAG). In the programmable pipeline, dependent tables (with read/write dependency) should be placed in continuous stages and independent tables can be placed within the same stage. For the sake of simplicity, we regard NFs as sequential *virtual* tables where each table can be merged from different independent tables.

Multiple-table NFs. To the best of our knowledge, SOTA NFs [15, 28] usually consist of few big tables that define the main processing functionality and other tables that handle exceptions. In this paper, particularly, we assign each NF with one big table since most tables handling the exceptions contribute little to the resource contention.

Shared Parser/Deparser. Since programmable switches only parse the packets before the ingress pipeline and compose/deparse the packets after the egress pipeline, it is required that different NFs should operate on the same set of packets header fields. Fortunately, cloud traffic usually follows standard protocols.

NF States. NF states are stored in SRAM together with MATs in switch pipelines. Hence, SFP could be further extended to account for NF states whose size should be fixed as well as MATs before compilation.

VIII. Related Work

P4 Switch Compilation. Some works[26, 29] explore the compilation of P4 programs to specific switch target and present some optimizations. Lyra[19] and μP4[30] focus on the portability problem across multiple programmable devices. However, these works are not designed for the SFC problems, and SFP can provide some optimizations regarding our insights on SFC provision.

P4-accelerated Applications. Since the IC vendors enable the programmability atop traditional switches, some works emerge to utilize the fast switch and offload their functionalities from end-host to in-network. [10] devises fast layer-4 load balancing. [14, 31] propose heterogeneous gateways to alleviate the servers. [15, 32] provide in-network cache for large-scale storage systems. [33] performs aggregation in switch to accelerate neural network training. [34] enables data aggregation to accelerate data processing applications. Most works require modifications to applications or communication protocols. In contrast, NF or SFC deployment is transparent to high-level applications and can be migrated from servers to switch at low cost, making it fast applicable to most cloud networks.

Multi-tenancy in P4. Due to limited capacity for accommodating multiple P4 programs within a single switch, software virtualization is applied to support multi-tenancy in P4. HyperV[35], Hyper4[36], and HyperVDP[37] provide full or partial virtualization that uses software to emulate hardware to enable multi-programming in the exclusive data plane. P4visor[18] presents a set of abstractions and merging algorithms to incorporate two programs to achieve virtualization in the data plane, which mainly involves code merging techniques. [38] designs compile and run-time approaches to multi-tenancy. These methods provide new directions to deploy multi-tenancy SFCs in P4, but SFP can utilize the

correlations of individual SFCs to better optimize resource utilization.

SFC Deployment in Programmable Switches. Recent works propose offloading SFC to programmable switches. [39] provides primitives to deploy SFCs in switches, but it cannot ensure resource efficiency because it merges SFCs in a straightforward LCS method. [40] proposes data plane design to enable multi-tenancy in switches and other works[19, 21, 41–49] alleviate the progress of offloading SFCs to programmable switches but they lack the support for runtime reconfiguration or multi-tenancy compared to SFP.

IX. Conclusion

We designed SFP, a system for cloud tenant SFC provision on programmable switches. SFP virtualizes the packet processing pipeline to execute each tenant's SFC using shared programmable switch chips in the data plane, and leverages integer programming with relaxation to place multiple tenants' SFCs efficiently with limited switch resources in the control plane. SFP also supports runtime update. We prototype SFP and validate its good properties of low latency, low overhead, high saturation, and efficient execution time.

Acknowledgement

We would like to thank the anonymous reviewers for their thoughtful feedback. This project is supported by National Natural Science Foundation of China under Grant No.61802225 and 62002019 and the Beijing Institute of Technology Research Fund Program for Young Scholars.

References

[1] W. Zhang *et al.*, "Opennetvm: A platform for high performance network service chains," in *HotMiddlebox'16*, 2016, pp. 26–31.

[2] C. Sun *et al.*, "Nfp: Enabling network function parallelism in nfv," in *SIGCOMM'17*. ACM, 2017, pp. 43–56.

[3] J. Martins *et al.*, "Clickos and the art of network function virtualization," in *NSDI'14*. USENIX Association, 2014, pp. 459–473.

[4] S. G. Kulkarni *et al.*, "Nfvnice: Dynamic backpressure and scheduling for nfv service chains," in *SIGCOMM'17*. ACM, 2017, pp. 71–84.

[5] S. Palkar *et al.*, "E2: A framework for nfv applications," in *SOSP '15*. New York, NY, USA: ACM, 2015, pp. 121–136.

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

[7] "https://www.alibabacloud.com/blog/the-network-architecture-and-network-management-system-behind-this-years-double-11_595615."

[8] F. engineering. Disaggregate: Networking recap. [Online]. Available: https://engineering.fb.com/2017/01/30/data-center-engineering/disaggregate-networking-recap/

[9] "https://scholar.harvard.edu/srivatsan-krishnan/publications/accelerating-recurrent-neural-networks-analytics-servers-comparison."

[10] R. Miao *et al.*, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *SIGCOMM'17*. New York, NY, USA: ACM, 2017, p. 15–28.

[11] Y. He *et al.*, "Scalable on-switch rate limiters for the cloud," in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, 2021.

[12] R. Datta *et al.*, "P4guard: Designing p4 based firewall," in *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*. IEEE, 2018, pp. 1–6.

[13] F. Hauser *et al.*, "P4-ipsec: Site-to-site and host-to-site vpn with ipsec in p4-based sdn," *IEEE Access*, vol. 8, pp. 139 567–139 586, 2020.

[14] K. Qian *et al.*, "Flexgate: High-performance heterogeneous gateway in data centers," in *APNet '19*. New York, NY, USA: ACM, 2019, p. 36–42.

[15] X. Jin *et al.*, "Netcache: Balancing key-value stores with fast in-network caching," in *SOSP' 17*. New York, NY, USA: ACM, 2017, p. 121–136.

[16] H. Huang *et al.*, "Nfd: Using behavior models to develop cross-platform network functions," in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, 2021.

[17] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM CCR*, vol. 44, no. 3, pp. 87–95, 2014.

[18] P. Zheng *et al.*, "P4visor: Lightweight virtualization and composition primitives for building and testing modular programs," in *CoNEXT'18*, 2018, pp. 98–111.

[19] J. Gao *et al.*, "Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics," in *SIGCOMM'20*, ser. SIGCOMM '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 435–450.

[20] "https://scholar.harvard.edu/srivatsan-krishnan/publications/accelerating-recurrent-neural-networks-analytics-servers-comparison."

[21] "Lightnf: Simplifying network function offloading in programmable networks," in *2021 IEEE/ACM International Symposium on Quality of Service*.

[22] D. Li *et al.*, "Virtual network function placement considering resource optimization and sfc requests in cloud datacenter," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 7, pp. 1664–1677, 2018.

[23] Gurobi. [Online]. Available: https://www.gurobi.com

[24] P4. [Online]. Available: https://p4lang.github.io/p4-spec/p4-14/v1.0.4/tex/p4.pdf

[25] G. P. Katsikas *et al.*, "Metron: NFV service chains at the true speed of the underlying hardware," in *NSDI'18*. Renton, WA: USENIX Association, 2018, pp. 171–186.

[26] L. Jose *et al.*, "Compiling packet programs to reconfigurable switches," in *NSDI'15*. Oakland, CA: USENIX Association, 2015, pp. 103–115.

[27] T. Benson *et al.*, "Network traffic characteristics of data centers in the wild," in *IMC '10*. New York, NY, USA: ACM, 2010, pp. 267–280.

[28] X. Jin *et al.*, "Netchain: Scale-free sub-rtt coordination," in *NSDI'18*. Renton, WA: USENIX Association, Apr. 2018, pp. 35–49.

[29] P. Bosshart *et al.*, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *SIGCOMM '13*. New York, NY, USA: ACM, 2013, p. 99–110.

[30] H. Soni *et al.*, "Composing dataplane programs with up4," in *SIGCOMM'20*, ser. SIGCOMM '20. New York, NY, USA: ACM, 2020, p. 329–343.

[31] T. Pan *et al.*, "Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches," in *SIGCOMM'21*, ser. SIGCOMM '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 194–206.

[32] Z. Liu *et al.*, "Distcache: Provable load balancing for large-scale storage systems with distributed caching," in *FAST' 19*. Boston, MA: USENIX Association, Feb. 2019, pp. 143–157.

[33] C. Lao *et al.*, "ATP: In-network aggregation for multi-tenant learning," in *NSDI' 21*. USENIX Association, Apr. 2021, pp. 741–761.

[34] L. Mai *et al.*, "Netagg: Using middleboxes for application-specific on-path aggregation in data centres," in *CoNEXT' 14*. New York, NY, USA: ACM, 2014, p. 249–262.

[35] C. Zhang *et al.*, "Hyperv: A high performance hypervisor for virtualization of the programmable data plane," in *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2017, pp. 1–9.

[36] D. Hancock and J. van der Merwe, "Hyper4: Using p4 to virtualize the programmable data plane," in *CoNEXT'16*. New York, NY, USA: Association for Computing Machinery, 2016, p. 35–49.

[37] C. Zhang *et al.*, "Hypervdp: High-performance virtualization of the programmable data plane," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 556–569, 2019.

[38] T. Wang *et al.*, "Multitenancy for fast and programmable networks in the cloud," in *HotCloud'20*. USENIX Association, Jul. 2020.

[39] X. Chen *et al.*, "P4sc: Towards high-performance service function chain implementation on the p4-capable device," in *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 2019, pp. 1–9.

[40] D. Wu *et al.*, "Accelerated service chaining on a single switch asic," in *HotNets '19*. New York, NY, USA: ACM, 2019, p. 141–149.

[41] K. Zhang *et al.*, "Gallium: Automated software middlebox offloading to programmable switches," in *SIGCOMM '20*. New York, NY, USA: Association for Computing Machinery, 2020, p. 283–295.

[42] N. Sultana *et al.*, "Flightplan: Dataplane disaggregation and placement for p4 programs," in *NSDI' 21*. USENIX Association, Apr. 2021, pp. 571–592.

[43] H. Liu *et al.*, "Sra: Switch resource aggregation for application offloading in programmable networks," in *GLOBECOM 2020*, 2020, pp. 1–6.

[44] X. Chen *et al.*, "Speed: Resource-efficient and high-performance deployment for data plane programs," in *ICNP' 20*, 2020, pp. 1–12.

[45] J. Ma *et al.*, "P4sfc: Service function chain offloading with programmable switches," in *2020 IEEE 39th International Performance Computing and Communications Conference (IPCCC)*, 2020, pp. 1–6.

[46] D. Moro *et al.*, "A framework for network function decomposition and deployment," in *2020 16th International Conference on the Design of Reliable Communication Networks DRCN 2020*, 2020, pp. 1–6.

[47] Y. Zhou *et al.*, "Flexmesh: Flexibly chaining network functions on programmable data planes at runtime," in *2020 IFIP Networking Conference*, 2020, pp. 73–81.

[48] Y. Xue and Z. Zhu, "Leveraging heterogeneous nfv platforms to upgrade service function chains in dcns," in *NetSoft' 21*, 2021, pp. 283–287.

[49] D. Moro *et al.*, "Network function decomposition and offloading on heterogeneous networks with programmable data planes," *IEEE Open Journal of the Communications Society*, vol. 2, pp. 1874–1885, 2021.

APPENDIX

A. Appendix to Mathematical Formalization of Placement Problem

Objective. As stated in the motivation, SFP aims to replace the server SFCs to save computation resources (and the overhead/cost consequently), thus SFP's objective is to offload as much traffic processing as possible, which is proportional to the traffic $d_{j=*,l}$ and the length of the chain J_l. So the objective is

$$\max \sum_l d_{j=*,l} \times T_l \times J_l. \tag{14}$$

Constraints. *Placement Constraints.* The variables about physical/logical NF placement are 0 or 1, which implies whether to place the physical/logical NF of type i in stage k.

$$x_{ik} \in \{0,1\}, \forall i,k. \tag{15}$$

$$z_{ijkl} \in \{0,1\}, \forall l,k,i,j. \tag{16}$$

Each type of physical NF must be assigned to at least one stage. Hence,

$$\sum_k x_{ik} \geq 1, \forall i. \tag{17}$$

Each logical NF can be allocated to at most one stage. Alternatively, it would be deployed in software.

$$d_{jl} \leq 1, \forall j,l. \tag{18}$$

If a logical NF is allocated, its type should be compliant with the actual type in the SFC (f_{jl}).

$$\sum_{i,k} z_{ijkl} \times i = f_{jl} \times d_{jl}, \forall j,l \tag{19}$$

All or none of NFs of the same SFC should be placed together, which implies we deploy SFC standalone in either hardware or software.

$$d_{j=1,l} = d_{j=2,l} = \cdots = d_{j=J_l,l}, \forall l \tag{20}$$

NFs in an SFC should be placed in pipeline stages in order. If the SFC is placed, it should satisfy

$$g_{j=1,l} < g_{j=2,l} < \cdots < g_{j=J_l,l}, \forall l \in \{l | d_{jl} = 1, \forall j\}.$$

If an SFC is not placed, all its NFs' stages are zero as following,

$$g_{j=1,l} = g_{j=2,l} = \cdots = g_{j=J_l,l} = 0, \forall l \in \{l | d_{jl} = 0, \forall j\}.$$

These two cases can be summarized as

$$g_{j=1,l} \leq g_{j=2,l} - d_{jl} \leq \cdots \leq g_{j=J_l,l} - d_{jl} * (J_l - 1). \tag{21}$$

Consistency Constraints. There are two consistency specific constraints in SFP. First, if a "logical" NF of a type is placed on a stage, the "physical" NFs of the same type must be allocated on the same physical stage.

$$z_{ijkl} \leq x_{ik}, \forall i,j,k,l. \tag{22}$$

Second, the virtual pipeline is extended from physical pipeline, thus, every S stages on the virtual pipeline repeat the same physical NF placement.

$$x_{ik} = x_{i,k+S}, \forall i, 1 \leq k, k+S \leq K. \tag{23}$$

Memory Constraints. Each stage has limited memory space to allocate NFs. And in the virtual pipeline, every S stages share the same physical pipeline's stage. On the k-th stage, all type-i NFs on all chains (each consumes $F_{jl} \times b$ memory) would consume $\lceil \frac{\sum_{k=s+nS}^{K} \sum_{j,l} (z_{ijkl} \times F_{jl} \times b)}{E} \rceil$ pages of memory. And the memory consumption on stage s ($1 \leq s \leq S$) of all types of NFs is constrained as follows

$$\sum_i \left\lceil \frac{\sum_{k=s+nS}^{K} \sum_{j,l} (z_{ijkl} \times F_{jl} \times b)}{E} \right\rceil \leq B, \quad \forall s \leq S. \tag{24}$$

Note that we consolidate the configuration of same-type NF across different SFCs into the same physical NFs, which eliminates the internal fragment of each NF. This is a memory efficient optimization. If consolidation across logical NFs is not allowed, the memory constraint should be

$$\sum_i \sum_{k=s+nS}^{K} \sum_{j,l} \left\lceil \frac{z_{ijkl} \times F_{jl} \times b}{E} \right\rceil \leq B, \quad \forall s \leq S. \tag{25}$$

Processing Capacity Constraints. The total traffic traversing the switch backplane is constrained by the backplane speed. It's noted that any recirculated traffic would compete with new inbound traffic.

$$\sum_l (R_l + 1) \times T_l \leq C. \tag{26}$$