pSFC: Fine-grained Composition of Service Function Chains in the Programmable Data Plane

Xiaoquan Zhang
Department of Computer Science
Jinan University
Guangzhou, China
zhangxiaoquan547@gmail.com

Lin Cui
Department of Computer Science
Jinan University
Guangzhou, China
tcuilin@jnu.edu.cn

Fung Po Tso
Department of Computer Science
Loughborough University
UK
p.tso@lboro.ac.uk

***Abstract*—Dynamic service function chains (SFC) are enabled by network function virtualization on general purpose servers. The emergence of programmable data planes (PDP) has offered a new way for the deployment of SFC. However, the implementation of network functions is constrained by resource limitations in PDPs (e.g., compute and memory resource). Moreover, most of existing works do not consider the optimization of state information (e.g., registers), which is essential for stateful network functions. In this paper, we propose *pSFC* which provides a fine-grained SFC deployment scheme in the PDP to tackle the problem. We first model network functions as control flow graphs (CFG) and the process of deployment as a one big switch (OBS) problem, and then propose an ILP (Integer Linear Programming) model for resource optimization for the OBS problem, which is NP-hard. To solve this problem efficiently, *pSFC* first composes multiple SFCs for eliminating redundant resources, decomposes the compound CFG based on the resource limitation per stage, and finally maps OBS into the substrate network. We have implemented *pSFC* in both bmv2 software switch and P4 hardware switch (i.e., Intel Tofino). Evaluation shows that *pSFC* reduces switch costs 45.7% and average latency 15% while providing the correctness of the process of SFC.**

***Index Terms*—P4, Service function chains, Programmable data plane, One big switch**

I. Introduction

Service function chains (SFCs), underpinned by network function virtualization (NFV), play an important role in datacenters and ISP networks because of its intrinsic support for creating dynamic network services [10]. The emergence of programmable data planes such as Protocol-Independent Switching Architecture (PISA) [2] attracts great attention from both academic and industry due to their flexible programmability and high performance that NFV can not achieve (e.g., 1TB/s throughput). This also opens up the possibility of deploying SFCs on the programmable data planes.

Due to the limited resources and capability of a single programmable switch, it is of paramount importance to optimise the resource usage such that as many SFCs as possible can be embedded on a given number of data planes. By doing so, there is also a possibility to save cost by eliminating the need for purchasing additional hardware. There have been a number of schemes proposed for the deployment of SFCs in the programmable data plane [12], [15]–[17], [20], [24], but they either only offer coarse-grained resource optimization or do not consider multiple data planes across multiple switches.

To achieve the deployment of multiple SFCs, some works use the recirculation function[1] provided by PISA [15], [20]. The recirculation allows packets re-entering the pipeline so that they can be processed by specific order of sequences in SFC. Thus, deployment of SFCs determines the number of times a packet reenters the pipeline, which can increase dramatically when the number of SFCs increases, degrading the overall performance. In addition to this potential performance issue, recirculation-based approach only works on a single switch.

In comparison, work carried out in [9] proposes merging duplicated network functions among SFCs into a P4 program implemented in the target switch. While this solution can effectively reduce resource usage by removing duplication at the function level (i.e., coarse-grained), it does not optimise resource usage at the table level of granularity (i.e., fine-grained), meaning that resource cannot be shared or aggregated on tables. Recent advancement of programmable data plane permits exploitation of more fine-grained resource optimization in the pipeline [19], [21]. For example, a P4 program can be split into a set of tables and dependencies between tables, and then merge as many same tables (redundant tables) as possible for reducing resource usage [23]. Unfortunately, this technique yet again only works on a single switch and does not consider synergistic optimization across multiple switches.

Worse still, existing literature also rarely takes stateful network functions, which are common in real networks [22], into consideration as current abstraction for P4 programs are inadequate to deal with stateful network functions. This is because P4 programs adjust their behaviour depending on the flow information saved in registers. However, the abstraction of above optimization works simply treats the register as the usage of resources without considering their properties such as size and action set, and lacks the definition of conditional statements to describe how the register is accessed [13] [23].

Based on the above analysis, we argue that *there is a need to create a fine-grained resource optimization scheme that can support the deployment of SFCs on data planes across multiple switches with consideration of stateful network functions*. Realizing this is challenging as it not only calls for

[1]Recirculation is a default operation in Intel Tofino switch ASIC.

a distributed architecture but also requires a new abstraction to express properties of registers and condition statements for supporting *register level* resource optimization.

In this paper, we propose such a scheme called *pSFC* (programmable SFC). In this architecture, *pSFC* first offers a fine-grained analysis for stateful network functions via CFG (Control Flow Graph) to find out potential composable logic and resources in the preparation of composition and placement. Then, *pSFC* composes network functions of multiple SFCs to a big compound CFG, in which it removes redundant logic and resources, as opposed to just functions, for reducing resource usage. Afterwards, *pSFC* abstracts the substrate network into a OBS (One Big Switch). The placement of the compound CFG is based on the objective of minimizing pipeline resource usage and network latency, constrained by requirements of SFCs and switch resources. Finally, *pSFC* designs corresponding metadata for inter-switch communication in P4 programs and deploys these P4 programs in the substrate network.

In short, this paper has made the following contributions:

1) An abstraction CFG is proposed to abstract stateful network functions with registers; ILP and OBS mapping models are proposed to model the resources optimization and scalability problem.
2) A two-steps composition algorithm is proposed to compose and remove redundant logic and resources, and a decomposition algorithm is proposed to schedule the stage usage [2] in the OBS environment with the objective and constraints.
3) A prototype of *pSFC* has been implemented on both hardware P4 switches (equipped with Intel Tofino ASIC) and software switches of bmv2. Extensive experiments show that *pSFC* can reduce 45.7% of stage usage in the implementation of five SFCs.

The rest of the paper is organized as follows. Section II models the system and formulates the problem. Section III describes the two algorithms and OBS mapping algorithm of *pSFC*. Section IV discusses the implementation and Section V shows the experiment results. Section VI gives an overview of related works and Section VII concludes the paper.

II. System Model and Problem Formulation

This section will describe the model of *pSFC*. Notations that will be used are listed in the Table I.

A. Overview

The system architecture of *pSFC* is shown in Figure 1. In the control plane, first, the controller translates P4 programs, i.e., network functions, to corresponding CFGs, which capture dependencies of P4 programs while considering state information such as registers (see Section II-B). Second, based on the SFC requests from the network administrator, the controller uses the composition algorithm of *pSFC* to compose the corresponding CFG diagrams of SFCs. Third, the composed CFG is decomposed based on constraints of resource per stage and dependencies between nodes (detailed information of stage constrains can be seen in Section IV). Finally, *pSFC* transfers the composed CFG as a OBS and leverages the deployment strategy to search appropriate switch resources for OBS deployment in the substrate network. As a result, the composed CFG is decomposed into multiple smaller CFGs each of which can be implemented in one target P4 switch. After that, the controller adds metadata information for inter-switch communication, and compiles corresponding P4 code, and configures to the P4 data plane.

[2]The pipeline of hardware switch is split into multiple stages.

B. CFG Abstraction

Given a P4 program, *pSFC* can detail its execution information (e.g., match-action table, registers and conditional statements), in the code of control flow through tools such as p4c-graph [6]. Most prior works translate a P4 program into TDG (Table Dependency Graph), which only displays dependencies between tables in the program [13] and is not enough to abstract various programs. For example, TDG is unable to express the information of registers since it only treats the registers as the consumption of SRAM in a match-action table [13]. Hence, we introduce CFG (Control Flow Graph), which can provide accurate abstraction of P4 programs for stateful network functions by considering information such as registers. A CFG is defined as a directed acyclic graph $G = \{V, E\}$, where V is the set of nodes and E is the set of directed edges which describe execution directions of the P4 program.

1) ***Nodes****:* There are three types of nodes in CFG:

- *Table node* V^t is the set of table nodes, each of which, say u, is comprised of a match-action table and contains: (i) match fields f_u^m, (ii) action set f_u^a , (iii) required resources H_u, which is a three-tuples consisting of resource types (key) and the amount (value) (e.g., resource types include SRAM for exact match, TCAM for ternary match and ALU for operations of registers), and (iv) set of registers N_u;
- *Action node* V^a is the set of action nodes, each of which has action set and required sources like table node, but does not have match fields;
- *Conditional node* V^c is the set of conditional nodes, each of which consists of a conditional statement (e.g. *if* statement in P4) whose result determines the direction of execution.

The action nodes are important because they are used to represent actions of registers in the table which usually do not have match fields. For example, some frequent use of operations like hash operation can be represented by the action node. By distinguishing between table nodes and action nodes in CFG, *pSFC* can perform more fine-grained analysis and improve the performance effectively by reducing redundancies.

2) ***Dependency****:* The dependencies between nodes in CFG are inevitable since different nodes may read/write the same states or data. Dependencies should be maintained during operations on CFGs. In addition to previous study [13], we further investigate the dependency of all three types of nodes

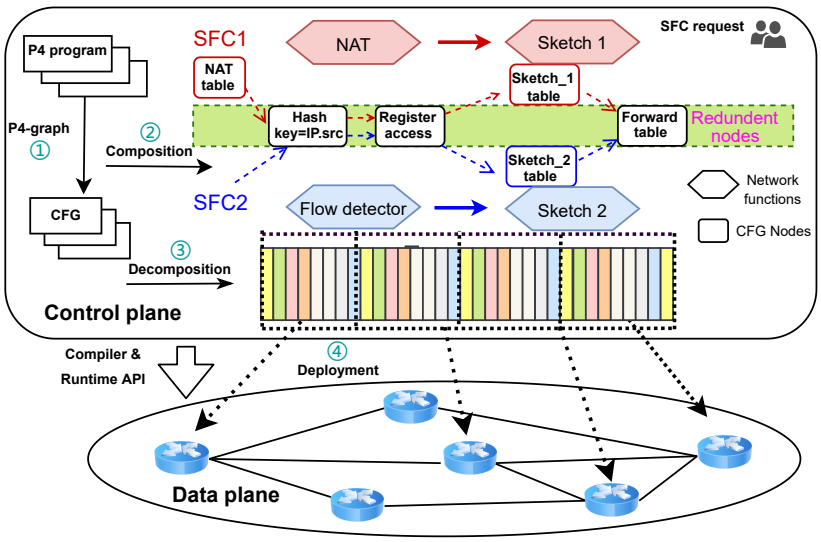


Fig. 1. Overview of *pSFC*. There are two SFCs, i.e., SFC1: *NAT→Sketch1* and SFC2: *Flow detector→Sketch2*. Their CFGs contains several redundant nodes (green rectangle) which can be composed together. Only part of CFG is shown in the figure due to space limitation.

in CFG. The following types of dependencies are defined in *pSFC* to accurately describe the dependencies between two nodes (say $u \to v$) in CFG:

(1) *match dependency*: u modifies fields $f \in f_u^a \cap f_v^m$;
(2) *action dependency*: u and v modify $f \in f_u^a \cap f_v^a$;
(3) *reverse match dependency*: u modifies $f \in f_u^m \cap f_v^a$;
(4) *successor dependency*: execution of v depends on the result of u;
(5) *condition dependency*: the result of a table/action node u affects the conditional statement in the conditional node v; and
(6) no dependency.

The dependencies (1)∼(3) may exist among table nodes and action nodes. The dependency (4) can be used to represent the relation of a conditional node and its successor nodes. The dependency (5) indicates how the result of register actions affect the processing direction. To simplify models, *pSFC* requires that a node must be directly connected to the conditional node if it contains operations which affect the result of the conditional node. For example, an action node containing the operation of fetching value from register is connected to the conditional node which compares the value with the threshold.

Let D represent the dependency matrix saving all dependencies among nodes in a CFG, where $D(u, v)$ represents the dependency of nodes $u \to v$. $D(u, v) = 0$ if there is no dependency between them, otherwise, $D(u, v) = 1$.

3) ***Register****:* In addition to tables which are usually read-only on data plane, state information can also be implemented with extern objects in P4, such as registers. We define the set of registers R, and each register $r_i \in R$ has a set of action blocks[3] B_{r_i}, which contains a series of actions, size of input and output. Two blocks are equal when they have the same actions and the size of input and output. Since register is more general than other extern objects, for the simplicity of description, we will mainly focus on registers in the following sections of this paper. However, similar ideas can be easily extended to other extern objects, such as counter and meter.

Access of registers, e.g., read and write, can affect the logic execution in the network function. They usually are associated to conditional nodes. For example, the value of a flow counter resulting from read operation in register r is subsequently used to compare with threshold in a conditional node (say value of $r > threshold$), thus the conditional node is associated to register r. In order to guarantee the correct logic execution, composition of two registers must ensure that their related conditional nodes are the same. *pSFC* uses Q_{r_i} to represent the set of all conditional nodes associated to register $r_i \in R$.

4) ***Principles of CFG composition****:* Composing multiple CFGs to a compound CFG are subject to the following constraints:

- *Node equivalence*: two nodes $u, v \in V^t \cup V^a$ are equal only if $f_u^m = f_v^m$, $f_u^a = f_v^a$, $H_u = H_v$ and $N_u = N_v$.
- *Register equivalence*: two registers $r_i, r_j \in R$ are equal if they have the same action blocks and set of conditional nodes associated to them, i.e., $B_{r_i} = B_{r_j}$ and $Q_{r_i} = Q_{r_j}$
- *Loop-free*: The compound CFG should be loop free[4] since P4 switch does not support loop operations [2]:

$$\nexists u, v \in G, \quad p(u, v) \neq \emptyset \ \wedge \ p(v, u) \neq \emptyset \tag{1}$$

- *Correctness*: When two CFGs G_i and G_j are composed into one compound CFG G_c, dependencies should be preserved:

$$\begin{cases} D_c(u, v) = D_i(u, v), \forall u, v \in G_i \\ D_c(u, v) = D_j(u, v), \forall u, v \in G_j \end{cases} \tag{2}$$

[3]The definition is based on RegisterAction [5] in $P4_{16}$ Language in Intel Tofino ASIC, and it can be easily applied to other pipeline architecture.

[4]The recirculation operations is also loop-free since the P4 program will execute in sequence eventually.

TABLE I
Notation of symbols

Definition of CFG	
G	$G = \{V, E\}$, node set $u, v \in V$ and edge set E
V	Table node V^t, action node V^a and conditional node V^c
f_u	Match fields f_u^m and action fields f_u^a of node u
H_u	Amount required resource in node u
N_u	Register set in node u
$D(u, v)$	Represents dependency of two nodes $u \rightarrow v$
R	Set of registers
B_{r_i}	Action blocks in registers $r_i \in R$
Q_{r_i}	Set of conditional nodes that are associated to register r_i
$p(u, v)$	Path from node u and v
Resource optimization in OBS	
$n_{i,j}$	The j-th redundant node of node n_i
$w(n_{i,j}, m)$	Required resources of redundant node $n_{i,j}$
U_m	Amounts of resource of type m per stage
Resource optimization in OBS: Binary variables	
$x_{i,j}^s$	Whether redundant node $n_{i,j}$ is placed on stage s
$z_{i,j}^s$	Whether redundant node $n_{i,j}$ incurs multiple placement on stage s
Resource optimization in OBS: Variables	
I_u	Input stage that node u places on
O_u	Output stage that node u places on
$C(n_{i,j}, s, m)$	Required resource of type m of redundant node $n_{i,j}$ placed on stage s
OBS mapping	
G_d	$G_d = \{V_d, E_d\}$, vertex set $u_d, v_d \in V_d$ and link set $u_d v_d \in E_d$ in the virtual network
G_c	$G_c = \{V_c, E_c\}$, vertex set $u_c, v_c \in V_c$ and link set $u_c v_c \in E_c$ in the substrate network
K_s	Total required resources of stage s in OBS
ϕ_{u_d}	Resources of vertex u_s in the substrate network
$x_{u_c}^{u_d}$	Whether virtual vertex u_d placed on substrate vertex u_c
$x_p^{u_d v_d}$	Whether virtual link $u_d v_d$ placed on path p
π_p	Latency of path p

C. Resource optimization in OBS

In order to compile multiple SFCs into the pipeline of target switches, *pSFC* uses an ILP model for resource optimization in the OBS, i.e., *pSFC* defines the network as one big switch with network-wide constraints. Based on constraints in target switches and requirements of SFCs, all constraints are linear constraints on either integer (e.g., table resources) or binary variables (e.g., placement policies). An objective function is set for minimizing stage usage. We now explain how to formulate constraints of the problem and the objective function.

Node assignment constraint: Different P4 programs may have the same nodes (satisfying the constraint of node equivalence) which are called redundancies. For example, the forward table usually appears in many network functions. Suppose node set has λ of nodes with different structures and each node has μ_i of redundancies. We use $n_{i,j}$ to represents the j-th redundant node of the node n_i. At the beginning of the algorithm, it tags specific ID for each unique node n_i derived from SFC requests, and then checks equivalent nodes of n_i and tags them to $n_{i,j}$ in order. For example, in Fig 2, node a appeals in $SFC1$ and $SFC2$, thus tagging the two nodes as $n_{a,1}$ and $n_{a,2}$. However, if a node need to be isolated with its equivalent node, the algorithm specifies another unique ID for the node. Thus, each node $n_{i,j}$ in the node set must be assigned to at least one stage $s \in S$ in the pipeline of target switches:

$$1 \leq \sum_{s \in S} x_{i,j}^s \leq \xi, \forall i \in [1, \lambda], j \in [1, \mu_i] \tag{3}$$

where the set of available stages in the pipeline is S.

A binary variable $x_{i,j}^s$ is used to indicate whether the node $n_{i,j}$ is placed on stage s. The total of $x_{i,j}^s$ in each stage of the node $n_{i,j}$ ranges from 1 to a threshold $\xi \in [1, |S|]$, where $|S|$ is the number of stages. This means that a node which may require more resources can be placed on multiple stages.

Let the integer variable $C(n_{i,j}, s, m)$ be resource of type $m \in M = \{TCAM, SRAM, ALU\}$ allocated to the node $n_{i,j}$ in stage s. If $x_{i,j}^s$ equals 1, the resource allocation on the node must be greater than 0, otherwise it must be equal to 0. And the total resources among stages must be equal to the corresponding resources requirement $w_{i,j,m}$ of node $n_{i,j}$:

$$\sum_{s \in S} C(n_{i,j}, s, m) = w(n_{i,j}, m), \forall i \in [1, \lambda], j \in [1, \mu_i], m \in M \tag{4}$$

Dependency constraint: The placement result must respect dependencies between nodes:

$$\forall D(u, v) = 1, O_v \leq I_u, \forall u, v \in V \tag{5}$$

Since a node can be placed in different stages, it must have a start stage I and an end stage O. When the two nodes u and v have a dependency ($D(u, v) = 1$), the latest stage O_v of node v must precede or equal to the earliest stage I_u of node u.

Redundancy placement constraint: *pSFC* allows same nodes with the same resource consumption (*Node equivalence*) to be placed in the same stage to share allocated resources and they only consume the resource of one node in the stage. A binary variable $z_{i,j}^s$ is used to indicate whether the node $n_{i,j}$ is the first node of the type of node n_i placed on stage s. *pSFC* uses the variable to avoid repeated calculation of resource consumption in the placement of redundant nodes. The variable has extensive relations with variable $x_{i,j}^s$. First, if there is no node placed on stage s, $z_{i,j}^s$ equals 0:

$$x_{i,j}^s = 0, z_{i,j}^s = 0, \forall i \in [1, \lambda], j \in [1, \mu_i], s \in S \tag{6}$$

Second, if there is one or several same nodes placed on stage s, the relation between the total of $z_{i,j}^s$ and $x_{i,j}^s$ can be modeled as:

$$\sum_{j \in [1, \mu_i]} x_{i,j}^s \cdot \sum_{j \in [1, \mu_i]} z_{i,j}^s = \sum_{j \in [1, \mu_i]} x_{i,j}^s, \forall i \in [1, \lambda], s \in S \tag{7}$$

where the total of $z_{i,j}^s$ among redundant nodes (i.e., $\sum_j z_{i,j}^s$) can be defined as a binary variable. When there are redundancies of n_i placed in stage s, it is equal to 1. The aim is to only accumulate one table's resource in the case of the placement of the same nodes in the same stage s. For example, in Fig 2, we can set $z_{a,1}^1$ to 1 and $z_{a,2}^1$ to 0 when the two nodes are

placed in stage 1, and then we can only accumulate one table's resource according to Equation 9.

Furthermore, since a node may be split into several partitions placed in different stages, we needs to guarantee that the resource consumption of same partitions of redundant nodes in the same stage are the same:

$$\sum_{j\in[1,\mu_i]} z^s_{i,j} = 1, x^s_{i,u}\cdot C(n_{i,j},s,m) = x^s_{i,v}\cdot C(n_{i,j},s,m), \\ \forall i\in[1,\lambda], s\in S, u,v\in V, m\in M \tag{8}$$

If there are redundancies, any redundant nodes placed on the same stage must be allocated the same resources.

Stage resource constraint: Every stage has limited amounts of resource with different types U_m, and the total resource consumption of all nodes placed on the same stage must not exceed it:

$$\sum_{i\in[1,\lambda]}\sum_{j\in[1,\mu_i]} z^s_{i,j}\cdot C(n_{i,j},s,m) \le U_m, \forall s\in S, m\in M \tag{9}$$

As explained above, the calculation of total resource consumption can use $z^s_{i,j}$ to avoid repeated calculation of node redundancies.

Objective function: The objective function is to minimize the stage usage by nodes, i.e., minimizing the maximum number of occupied stages:

$$\min(\max(x^s_{i,j}\cdot s)), \forall s\in S, i\in[1,\lambda], j\in[1,\mu_i] \tag{10}$$

D. OBS placement in the network

The next step of *pSFC* is to deploy OBS in the substrate network. *pSFC* translates the OBS program from the ILP result into a virtual network $G_d = \{V_d, E_d\}$, in which a virtual link ($u_dv_d \in E_d$) connects two virtual vertices ($u_d, v_d \in V_d$) and each virtual vertex consists of a certain number of stages ϵ. The substrate network can be defined as $G_c = \{V_c, E_c\}$, where V_c is the set of vertices and E_c is the set of links. u_c and v_c are two vertices and u_cv_c is a link in the substrate network.

Then, the problem becomes a mapping problem. First, virtual vertex $u_d \in V_d$ must be only assigned in a vertex $u_c \in G_c$, and total required resources of u_d are less than resources provided by u_c:

$$\sum_{s\in[1,\epsilon]} x^{u_d}_{u_c}\cdot K_s \le \phi_{u_c}, \forall u_c\in V_c, u_d\in V_d \tag{11}$$

where K_s is total required resources of stage s, and ϕ_{u_d} is total resources of vertex u_c in the substrate network.

Second, during link mapping, each link $u_dv_d \in E_d$ will only be assigned to a path $p \in P(u_c, v_c)$ between two target P4 switches $u_c, v_c \in V_c$:

$$\sum_{p\in P(u_c,v_c)} x^{u_d}_{u_c}\cdot x^{v_d}_{v_c}\cdot x^{u_dv_d}_p = 1, \forall u_c, v_c\in V_c, u_dv_d\in E_d \tag{12}$$

where the binary variable $x^{u_d}_{u_c}$ indicates whether virtual vertex u_d is mapped to u_c, and $x^{u_dv_d}_p$ indicates whether link u_dv_d is mapped to path p.

Algorithm 1 Composing single SFC

Input: G_1, G_2, CFG G_1 and G_2
Output: G_c, result of composition
1: $L1 \leftarrow$ TopologySort(G_1)
2: $L2 \leftarrow$ TopologySort(G_2)
3: $S \leftarrow L1 \cap L2$
4: **for** each s in S **do**
5: **if** CheckDependency(s) **then**
6: S.PopNodes(s)
7: **end if**
8: **end for**
9: $G_c \leftarrow$ MergingRedundentNodes(S, G_1, G_2)

The objective of mapping is to minimize network latency:

$$\min(\sum_{u_dv_d\in E_d}\sum_{u_d,v_d\in V_d}\sum_{p\in P(u_c,v_c)} x^{u_d}_{u_c}\cdot x^{v_d}_{v_c}\cdot x^{u_dv_d}_p\cdot\pi_p) \tag{13}$$

where π_p is the latency of path p.

E. Complexity analysis

Both the two problems above can be easily proved to be NP-hard. Assuming all nodes have no dependencies among CFGs and there are no redundancies for all nodes, the problem reduces to a MKP [14] (Multiple Knapsack Problem) problem which has been proved to be a NP-Complete problem. Thus, problems in both Section II-C and Section II-D are NP-hard.

III. *pSFC* ALGORITHMS

A. Composing SFCs

Composing single SFC. Algorithm 1 shows operations for composition of single SFC. It first computes an intersection S from two topological orderings of $S1$ and $S2$ (Line 1∼3). Then the function $CheckDependency()$ analyzes whether a node in S violates dependency with other nodes before and after the composition. If so, the node will be removed from S. Afterwards, the function $MergingRedundenctNodes()$ merges the two CFGs by S, and sequentially connects other nodes to obtain an SFC execution logic. For example, as the composition of two CFGs shown in Figure 2, the two CFGs have the "e" node as their last node, and the Algorithm will sequentially connect other nodes based on their sequence in the SFC request and lastly compose the mergeable node. The merge operation in line 9 may change the execution order of CFG nodes within a single network function to achieve more optimization, as long as there is no dependency conflict among all affected nodes (i.e., $D(u, v) = 0$). The above Algorithm can be called repeatedly to compose all network functions to a SFC. The time complexity is at the level of $\mathcal{O}((n+e)\cdot m)$, in which n and e are the number of nodes and links in CFG respectively, and m is the number of network functions, since the Algorithm require to execute function $TopologySort$ by m times.

Composing multiple SFCs: The composition of multiple SFCs is described in Algorithm 2. It first calculates the topological ordering of compound CFGs S_1 and S_2 respectively,

Algorithm 2 Composing multiple SFCs

Input: S_1, S_2, composing SFC graph S_1 and S_2
Output: G_c, result of composition

1: $L1 \leftarrow$ TopologySort(S_1)
2: $L2 \leftarrow$ TopologySort(S_2)
3: $l \leftarrow$ LCS($L1$, $L2$)
4: **if** $l == \emptyset$ **then**
5: $G_c \leftarrow$ Parallel(S_1, S_2)
6: **else**
7: $G_c \leftarrow S_1 + S_2$
8: RemoveNode(G_c, l)
9: RemoveEdge(G_c, l)
10: **end if**

and then computes mergeable nodes of the two CFGs using LCS-based (Longest Common Subsequence) methods (lines of 1∼3). Since nodes vary in resource consumption, the benefits of merging different nodes are different. Merging the "big table" which occupies more resources is usually more effective in saving pipeline resources. For example, an ACL table may consume multiple stages due to the requirement of large TCAM memory, in contrast to a $drop()$ table only occupying few resources. Hence, *pSFC* adds weighted value for these "big tables" based on their resource consumption during the calculation of composable nodes l when *pSFC* computes the result of LCS (line 3). If l has no nodes, the two CFGs will be processed in parallel in the stage (line 5). Otherwise, *pSFC* connects the two CFGs according to nodes in l, and remove redundant nodes (line 7∼9). We also give an example of the composition in multiple SFCs as shown in Figure 2. The algorithm composes three mergeable nodes to form a new composing CFG.

The above Algorithm can be called repeatedly to compose all SFCs. The time complexity is at the level of $\mathcal{O}(p \cdot q \cdot m)$, in which p and q are the number of nodes of the two CFGs respectively and m is the number of SFCs. The time complexity of LCS algorithm is $\mathcal{O}(p \cdot q)$, and it is required to call m time as there are m of SFCs.

B. Decomposition algorithm

Algorithm 3 is a greedy heuristic approach to achieve per-stage placement, which aims to minimize the number of required stages without violating any capacity constraints and dependencies. Queue Q saves nodes that have been processed (i.e., have been placed in stage) and their rear nodes are going to be processed. The Algorithm is started from the "Start" node of the composed CFG, arranging the stage for rear nodes of the "Start" node and inserting them in Q. Afterwards, the Algorithm conducts an iteration that places nodes of the queue Q in stages until all nodes of the CFG are placed in stages.

There are two situations when nodes are being processed. First, if a node is destined to only one adjacent node (i.e., only one direction) and they do not violate dependency (e.g., $ifDependency()$), the algorithm will trigger the placement of the adjacent node by function $PlacingAdjacentNode()$

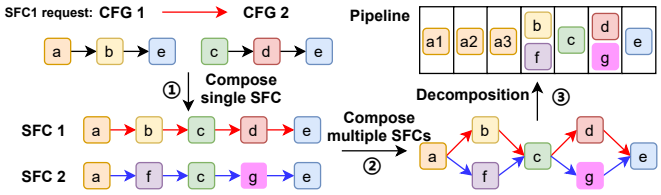


Fig. 2. Example of composition and decomposition

(line 12∼13). In function $PlacingAdjacentNode()$, only if all front nodes of the input node (e.g., *m_rear*[0]) have been processed, the input node can be placed in a stage. Next, the function executes an iteration that places the next adjacent node of the processed node until a node violates the above conditions.

Second, if the node has several adjacent nodes or has dependency with its adjacent node, the number of placement stages increases by 1 since they must be placed in different stages to avoid dependency conflict. All front nodes of each adjacent node must be processed in case of disorder of dependency, otherwise skipping the adjacent node. Then, the function $SearchNodeMaxStage()$ computes the maximum stage s_{max} among front nodes of the adjacent node, and $ArrangeStage()$ places the adjacent node in the $s_{max}+1$. In $ArrangeStage()$, if resources of the stage can not satisfy the requirement of the node, the function will return a new stage (e.g., $stageID + 1$) with sufficient resources for the node.

Observation from experiments shows that the LCS-based strategy may prolong the length of required stages in the pipeline of target switches. Assuming two SFCs, $A \rightarrow B \rightarrow C$ and $C \rightarrow D \rightarrow E$, each node has dependency with its adjacent node. The LCS-based strategy will compose the two chains into $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$, which consumes 5 stages. However, placing them in parallel needs only 3 stages. Thus, *pSFC* puts forward a reconstructing mechanism to revise the placement. In the reconstruction, each SFC may be extracted from the composed CFG obtained by the Algorithm 2, and its resource consumption will be returned back to corresponding stages. Then the SFC is re-installed in available stages by a heuristic algorithm. However, the mechanism can try all SFCs with different depths (e.g., it can extract multiple SFCs at a time). In order to reduce the searching scale, *pSFC* quantifies each SFC in advance and gives those top SFCs a priority treatment (e.g., based on the number of "big table"). Experiments show that the reconstructing mechanism can effectively improve *pSFC* performance (see Section V).

Algorithm 3 requires traversing all nodes of the CFGs, and the time complexity of searching front nodes (line 16) and $SearchNodeMaxStage()$ (line 17) are $\mathcal{O}(m)$, where m is the number of nodes in the composing CFG. Since the function $SearchNodeMaxStage()$ can directly use the result of searching front nodes, the complexity of Algorithm 3 is $\mathcal{O}(m^2)$.

Algorithm 3 Decomposition algorithm

Input: G, composing CFG
Output: D, result of decomposition with node placement

```
Q ← ∅
m_rear ← G.SearchRearNode('Start')
for each node in m_rear do
    insert node in Q and D if a node whose front node only is 'Start' node
end for
while Q ≠ ∅ do
    m ← Q.pop()
    m_rear ← G.SearchRearNode(m)
    if m_rear == ∅ then
        Continue
    end if
    if len(m_rear) == 1 and !IfDependency(m_rear[0]) then
        PlacingAdjacentNode()
    else
        for each node in m_rear do
            if any front node of node has been processed then
                stage ← G.SearchNodeMaxStage(node)
                ArrangeStage(D, node, stage + 1)
                Q.insert(node)
            end if
        end for
    end if
end while
```

C. One big switch mapping

Deployment in the substrate network: The next step is searching switch resources for deployment in the substrate network. *pSFC* leverages a greedy strategy algorithm which satisfies requirements of SFCs (e.g., minimizing latency). *pSFC* takes the stage ID as the searching step, in which it checks whether accumulated resources of stages exceed the limitation of single target P4 switch (e.g., the number of stages). If so, the current stage ID will be a new accumulated starting point, and the previous stages will be placed in an available switch in the substrate network.

Inter-switches communication: Since an SFC may be deployed in multiple switches, it is inevitable that a complete P4 program would be decomposed and deployed on different switches. Inter-switches communication is important for correct packet processing when the downstream switch requires the results of the previous table in the upstream switch as inputs. *pSFC* use *SFCID* as metadata in the initialization of packet headers to drive packets traversing specific tables among switches for correct processing. Intermediate data can also be carried as metadata in the packet, then they can be transmitted among switches. Previous works have shown that using metadata as data carries only occupies a small fraction of resources in the pipeline and bandwidth [8]. In the last step, *pSFC* provides metadata information to the compiler after the calculation of deployment, and generates configurations and installs rules in corresponding switches.

TABLE II
P4 PROGRAMS (NETWORK FUNCTIONS) AND THEIR REQUIRED RESOURCES

Name	# Stages	# Nodes	# Registers
HeavyHitter1	2	3	1
HeavyHitter2	4	5	1
ECMP	3	3	0
FlowletSwitch	5	7	2
SYNDetection	3	6	2
UDPFloodDetection	4	6	2
FlowSizeDetection	3	5	2
SpreadDetection	3	6	2
QoS	3	5	0
ACL	4	4	0
sFlow	3	3	0
Sketch1	6	6	16
Sketch2	6	6	16

IV. IMPLEMENTATION

We have implemented a prototype of *pSFC* based on both the software switch bmv2 [6] and the P4 hardware switch (Flnet S9180-32X with Intel Tofino ASIC). $P4_{16}$ is adopted as the programming language in the implementation. *pSFC* translates $P4_{16}$ programs to CFGs via *p4c-graph* [6], outputting *.dot* files to analyze and define their own properties in CFGs. All algorithms are implemented in the control plane via Python. The runtime control of bmv2 is achieved by *P4Runtime* [6] (e.g., configuration).

Target switch model: In our experiment, we use the Intel Tofino ASIC [2] as the model of the pipeline. The pipeline includes 12 stages, each which assembles 10Mbits of SRAM, 528Ktrits of TCAM and 8 ALUs (each register occupies two ALUs for write and read) . Besides, in order to provide high speed operation on Tofino ASIC, dependencies are defined in the pipeline as following: (1) match dependency, i.e., upstream table modifies downstream table; (2) action dependency, i.e., two tables mutually modify the same field; (3) successor dependency, i.e., the conditional operation happens.

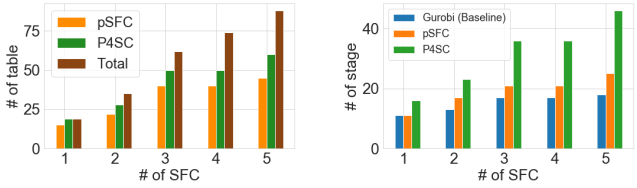
V. EVALUATION

In this section, we evaluate the performance of *pSFC* on both software switches (i.e., bmv2) and P4 hardware switches (i.e., Intel Tofino ASIC).

A. Experiment setup

Environment: The test-bed consists of two servers (equipped with 8 Intel Core i7-4771 CPU @ 3.50GHz), each configured with a 10Gbps NIC, and two P4 hardware switches (Flnet S9180-32X with Intel Tofino switch ASIC) connecting the servers.

In simulation experiments, we built a topology that includes P4-programmable switches implementing network functions and legacy switches with the function of L3 routing to evaluate OBS placement in the network. Mininet [4] is used to build the network environment with bmv2 [6] as P4 software switches. We randomly produce the network topologies, which includes



(a) Performance of table optimization (b) Comparison of stage usage

Fig. 3. Comparison of resource optimization

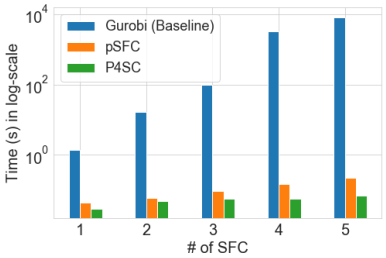


Fig. 4. Comparison of running time

more than 20 switches (half of switches are set to be P4 switches) and each link's latency ranging from 10ms to 15ms.

P4 programs: We have also implement thirteen P4 programs (i.e., network functions), as listed in Table II. These programs are from P4 tutorials [6], example programs of Intel Tofino ASIC [2] and stateful network function works [7]. Note that we simply distinguish stateful network function with stateless by the usage of registers among the programs we used [22]. The definition is not the same as state-intensive network functions. For example, NAT is a typical state-intensive network function and its *states* can be constructed as a table without the usage of registers.

Comparison schemes: We have implemented P4SC [9] for comparison. P4SC is a coarse-grain solution without any deep analysis in the programmable pipeline. Since the deployment of multiple SFCs may require resources of several P4 switches in the experiments, we have also employed the same inter-switch communication as *pSFC*.

We also use Gurobi [1], mathematical optimization solver, to solve the ILP model as a baseline to compare with *pSFC*. Gurobi can find an optimal solution but it is also very time-consuming.

B. Performance of resource optimization

Correctness. To check whether *pSFC* can guarantee that packets are processed in the correct sequence of each SFC., we implement five SFCs with network functions mentioned above. For the purpose of correctness verification, we also implement the same SFCs without any optimization. Then, requests of SFCs are sent in one server, and receive them in the other server. By the comparison of packet information (e.g., pcap files), we found that packets in *pSFC* have the same processing results with SFCs without optimization.

Comparison of resource usage. To compare the capability of removing redundancies, we calculate the number of required tables in results of *pSFC* and the P4SC. Note that, there exists big tables which need to occupy several stages, so that a big table may be counted in our result multiple times. The conditional nodes are not counted in the results since they consume no table resources. Results are shown in Figure 3(a). The total number refers the number of required tables in SFCs before composing. *pSFC* outperforms the P4SC in terms of the capability of composing redundancies by up to 25% in the composition of five SFCs, due to the fine-grained composition *pSFC* provides in the level of tables instead of network functions. More specifically, the abstraction CFG enhances the composition possibility of the same resource usage (table and register), thereby saving resources for the decrease of stage usage.

Figure 3(b) shows performance results on resource optimization by stage usage among the three methods. *pSFC* outperforms the P4SC in term of the capability of composing redundancies by up to 45.7% in the composition of five SFCs. In comparison, the performance of P4SC degrades with the increase of the number of SFCs. Especially in the experiment of composing five SFCs, its stage usage is three times more than Gurobi.

The main reason for a small performance gap between Gurobi and *pSFC* is that the latter tends to compose as many redundancies as possible, and may ignore more optimal composition. Although *pSFC* has proposed reconstruction of SFCs to tackle the problem, it only works in a complete SFC as *pSFC* do not consider the correctness of individual SFC but all of them in one request. Furthermore, we also noticed a surge of stage usage in the P4SC but a steady increase by a small fraction between *pSFC* and Gurobi in the third SFC composition experiment. Two factors contribute to the result. The LCS strategy may expand the length of the compound CFG in P4SC. Another factor is the reconstruction strategy of *pSFC*, which composes the third SFC to the compound CFG of the first two SFCs in parallel so as to maximize the use of available resources in the pipeline. The final observation is that the three methods can compose redundancies efficiently. Network functions in the fourth SFC are the duplication of that of the first three SFCs, and hence the stage usage has little change among the three methods.

Comparison of running time. We measure the running time of the three methods. As shown in Figure 4, the running time increases with the number of SFCs for all three methods. We have set a maximum stage ID (a constant) in the ILP model, and tuned the constant will affect the searching times in Gurobi. In the experiment, there is no solution if the constant is too small, but if it is too large the searching time prolongs. A proper value of the constant can accelerate the searching speed. However, the value is in fact unpredictable. Thus, we set the maximum stage ID to 20, and the experiment shows that Gurobi takes near one hour in the fourth SFC experiment and more than two hours in the fifth SFC experiment. In contrast

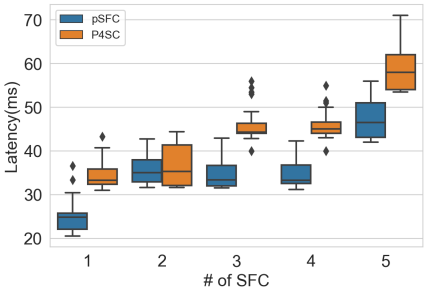
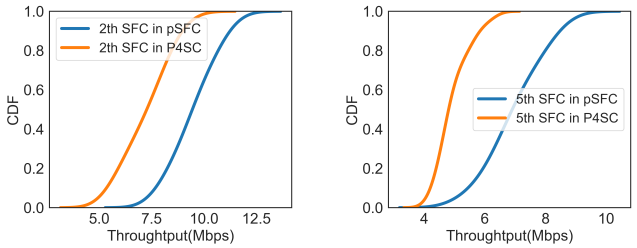


Fig. 5. Comparison of latency



(a) Comparison in the composition of the 2nd SFC

(b) Comparison in the composition of the 5th SFC

Fig. 6. Comparison of throughput

to Gurobi, *pSFC* is practical as it has low execution time even though *pSFC* has an apparent upward trend with the increased number of SFC as it reconstructs each SFC.

C. Performance of SFCs deployment

In the following simulation experiments, we implement P4 configurations in software switches bmv2 according to the result of OBS mapping, and establish a random topology in Mininet to measure the latency and throughput of *pSFC* and P4SC.

Comparison of latency. Figure 5 shows that average latency in *pSFC* is lower than P4SC under different numbers of SFCs by up to 15%. The major reason is that the stage usage ultimately implies the number of switches: The average usage of P4 switches in *pSFC* is lower than P4SC, and thus the number of switches that packets traverse is less. Besides, since we randomly tag *SFCID* to packets for requests of SFC in the experiment, packets are randomly processed by SFCs. Hence, the latency has a wide distribution in values

Comparison of throughput. As shown in Figure 6, *pSFC* outperforms P4SC in the composition of SFCs on throughput (we give results of the composition of the second and fifth SFC). There are two factors that can affect the results in Mininet. First, since complicated logic will decrease throughput in bmv2, *pSFC* outperforms P4SC when composing the same number of SFCs. Second, the increase of the number of switches leads to impairment of performance when the number of SFCs increases in both methods.

TABLE III
RESOURCE USAGE ON P4 HARDWARE SWITCH (TOFINO ASIC)

	pSFC	P4SC
Per stage resource		
SRAM	19.4%	16.1%
TCAM	29.4%	29.4%
ALU	46.4%	45.8%
The amount of resource		
Stage	21	36
Table	40	50
Number of switches	2	3

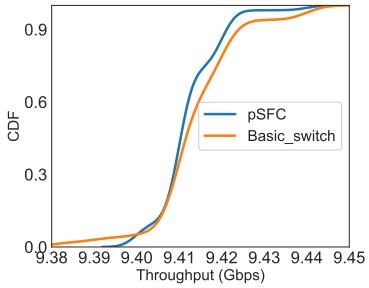


Fig. 7. Performance on P4 hardware switches (Tofino ASIC)

D. Performance with P4 hardware switches

In this experiment, we implement three SFCs using *pSFC* and P4SC with P4 hardware switches.

Resource usage. We compare the average resource usage in the switch pipeline as shown in Table III. *pSFC* and P4SC have very close resource usage in the pipeline. However, the demand of stage and table usage of *pSFC* is less than that of P4SC, as shown in Figure 3(b). In other words, the amount of consumption of SRAM, TCAM and ALU in P4SC is far more than in *pSFC* . The phenomenon results from the mechanism of eliminating redundant logic in *pSFC* . It demonstrates that *pSFC* not only is effective to reduce stage usage, but also can reduce resource usage instead of exhausting resources per stage. This is important because resources in a hardware switch is split into multiple stages (e.g., Intel ASIC has 12 stages per pipeline), using less number stages will save more cost with few number of switches.

Throughput. We compare *pSFC* with a simple switching program (baseline) in the P4 hardware switch by iperf [3]. As shown in Figure 7, *pSFC* achieve an average throughput of 9.412Gbps, which is very close to the baseline of 9.4136Gbps.This clearly demonstrates that *pSFC* does not compromise line rate performance.

VI. RELATED WORKS

Multiple programs optimization. Many recent researches have optimized multiple programs placement in hardware switch. On the one hand, an appropriate table scheduling can provide correct packet processing among programs while avoiding the waste of available resources [11], [18], [19], [21]. On the other hand, merging redundant tables is an effective way to reduce the usage of resources in the pipeline [8], [23].

However, they do not consider chained network functions. *pSFC* combines and improves the strength of these two to solve the deployment problem of multiple SFCs.

NFV in the programmable data plane. Prior works have discussed the deployment of chained network functions in the programmable data plane [12], [16], [17], [24]. These methods usually set up a forward table in the beginning of the pipeline in order to indicate in which stages the packet will enter. Furthermore, optimal techniques in resource usage can be categorized in two methods. First, packets are able to enter the pipeline multiple times so as to be processed in the correct sequence of network functions via recirculation operations [15], [20]. Second, merging the same network functions between SFCs can also reduce deployment cost in the case of the deployment of multiple service function chains [9]. Different from these works, *pSFC* avoids latency caused from multiple entries in the pipeline, and proposes a fine-grained scheme to merge redundant logic.

VII. Conclusion

In this paper, we propose *pSFC* which provides a fine-grained SFC deployment scheme in the programmable data plane. *pSFC* uses CFG to model P4 programs for fine-grained optimization, models the deployment of the SFCs as a OBS problem and proposes an ILP model for resource optimization in OBS network. *pSFC* proposes several algorithms to solve the problem. First, the composition algorithm is to compose multiple SFCs while diminishing redundant resources for cost reduction. Second, the placement algorithm decomposes the compound CFG based on the resource limitation per stage, and finally mapping OBS into the substrate network. Experiments show that *pSFC* reduces switch costs 45.7% and average latency 15% while providing the correctness of the process of SFC.

Acknowledgments

This work has been partially supported by National Natural Science Foundation of China (NSFC) No. 62172189 and 61772235; Natural Science Foundation of Guangdong Province No. 2020A1515010771; Science and Technology Program of Guangzhou No. 202002030372; The UK Engineering and Physical Sciences Research Council (EPSRC) grants EP/P004407/2 and EP/P004024/1, and InnovateUK grant 106199-47198.

References

[1] Gurobi - The Fastest Solver. https://www.gurobi.com/. Accessed on: Feb. 20, 2022.

[2] Intel Tofino switch ASIC. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html. Accessed on: Feb. 20, 2022.

[3] iperf - the ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr/. Accessed on: Feb. 20, 2022.

[4] Mininet: An instant virtual network on your laptop. http://mininet.org/. Accessed on: Feb. 20, 2022.

[5] Open-Tofino. https://github.com/barefootnetworks/Open-Tofino. Accessed on: Feb. 20, 2022.

[6] P4language. https://github.com/p4lang. Accessed on: Feb. 20, 2022.

[7] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. SNAP: Stateful network-wide abstractions for packet processing. In *Proceedings of the ACM SIGCOMM Conference*, pages 29–43, 2016.

[8] Xiang Chen, Hongyan Liu, Qun Huang, Peiqiao Wang, Dong Zhang, Haifeng Zhou, and Chunming Wu. SPEED: Resource-efficient and high-performance deployment for data plane programs. In *IEEE 28th International Conference on Network Protocols (ICNP)*, pages 1–12. IEEE, 2020.

[9] Xiang Chen, Dong Zhang, Xiaojun Wang, Kai Zhu, and Haifeng Zhou. P4SC: Towards high-performance service function chain implementation on the p4-capable device. In *IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 1–9. IEEE, 2019.

[10] Lin Cui, Fung Po Tso, Song Guo, Weijia Jia, Kaimin Wei, and Wei Zhao. Enabling heterogeneous network function chaining. *IEEE Transactions on Parallel and Distributed Systems*, 30(4):842–854, 2019.

[11] 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.

[12] Mu He, Arsany Basta, Andreas Blenk, Nemanja Deric, and Wolfgang Kellerer. P4nfv: An NFV architecture with flexible data plane reconfiguration. In *14th International Conference on Network and Service Management (CNSM)*, pages 90–98. IEEE, 2018.

[13] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling packet programs to reconfigurable switches. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 103–115, 2015.

[14] Hans Kellerer, Ulrich Pferschy, and David Pisinger. Multidimensional knapsack problems. In *Knapsack problems*, pages 235–283. Springer, 2004.

[15] Jaewook Lee, Haneul Ko, Hochan Lee, and Sangheon Pack. Flow-aware service function embedding algorithm in programmable data plane. *IEEE Access*, 9:6113–6121, 2020.

[16] Junte Ma, Sihao Xie, and Jin Zhao. P4SFC: Service function chain offloading with programmable switches. In *IEEE 39th International Performance Computing and Communications Conference (IPCCC)*, pages 1–6. IEEE, 2020.

[17] Tomasz Osiński, Halina Tarasiuk, Lukasz Rajewski, and Emil Kowalczyk. DPPx: A P4-based data plane programmability and exposure framework to enhance NFV services. In *IEEE Conference on Network Softwarization (NetSoft)*, pages 296–300. IEEE, 2019.

[18] Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doenges, and Nate Foster. Composing dataplane programs with μP4. In *Proceedings of the ACM SIGCOMM Conference*, pages 329–343, 2020.

[19] Hardik Soni, Thierry Turletti, and Walid Dabbous. P4Bricks: Enabling multiprocessing using linker-based network data plane architecture. 2018.

[20] Dingming Wu, Ang Chen, TS Eugene Ng, Guohui Wang, and Haiyong Wang. Accelerated service chaining on a single switch ASIC. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, pages 141–149, 2019.

[21] Cheng Zhang, Jun Bi, Yu Zhou, Abdul Basit Dogar, and Jianping Wu. Hyperv: A high performance hypervisor for virtualization of the programmable data plane. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9. IEEE, 2017.

[22] Xiaoquan Zhang, Lin Cui, Kaimin Wei, Fung Po Tso, Yangyang Ji, and Weijia Jia. A survey on stateful data plane in software defined networks. *Computer Networks*, 184:107597, 2021.

[23] 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.

[24] Yu Zhou, Jun Bi, Cheng Zhang, Mingwei Xu, and Jinaping Wu. FlexMesh: Flexibly chaining network functions on programmable data planes at runtime. In *IFIP Networking Conference (Networking)*, pages 73–81. IEEE, 2020.