FlexChain: Bridging Parallelism and Placement for Service Function Chains

Sihao Xie, Junte Ma, Jin Zhao, *Senior Member, IEEE*

***Abstract*—A Service Function Chain (SFC) is an ordered sequence of network functions (NFs). With the emerging Network Function Virtualization (NFV) paradigm, NFs can be deployed as software instances on commodity servers, leading to a more flexible provision of network service. However, the flexibility of NFV comes with considerable compromises since virtual network functions (VNFs) introduce significant processing latency overheads. In this paper, we design a flexible SFC parallel system called FlexChain, enabling the parallelism among VNFs to reduce the processing latency of SFCs. To leverage the benefits of parallelism, we study the problem of joint optimization over SFC parallelism and placement with the objective of accepting as many requests as possible. Since the problem is proved to be NP-hard, we propose a parallelism-aware approximation placement algorithm with performance guarantees, and an efficient heuristic algorithm for large-scale data center networks. Our simulation results show that FlexChain combined with our placement algorithm can substantially improve the number of accepted flows in the latency-sensitive scenario, and significantly reduce the average latency of accepted flows at the same time.**

***Index Terms*—NFV, network function parallelism, SFC, placement**

I. Introduction

NETWORK Function Virtualization (NFV) was recently introduced to address the limitations of dedicated middleboxes. By decoupling network functions (NFs) from the underlying dedicated hardware and implementing them in the form of software, NFV offers the potential for both enhancing service provision flexibility and reducing overall costs [1]. Network flows are often required to be processed by multiple NFs in an ordered sequence. For example, a flow needs to be filtered by a Firewall before it is processed by a Load Balancer. This notion is known as Service Function Chaining (SFC) [2].

However, the benefits of NFV come with considerable compromises especially on packet processing latency. For example, a software-based stateful firewall, implemented with Data Plane Development Kit (DPDK) [3], processes packets 6.4 times slower than the hardware-based implementation on average [4]. Moreover, the SFC latency grows linearly with the length of the chain. Such long latency may be unacceptable for latency-sensitive network applications, such as algorithmic stock trading and virtual reality online collaboration.

Manuscript received June 1, 2020; revised October 23, 2020; accepted December 21, 2020. The work was supported in part by Natural Science Foundation of China under Grant 61972101. The associate editor coordinating the review of this article and approving it for publication was G. Schembra. (Corresponding author: Jin Zhao.)

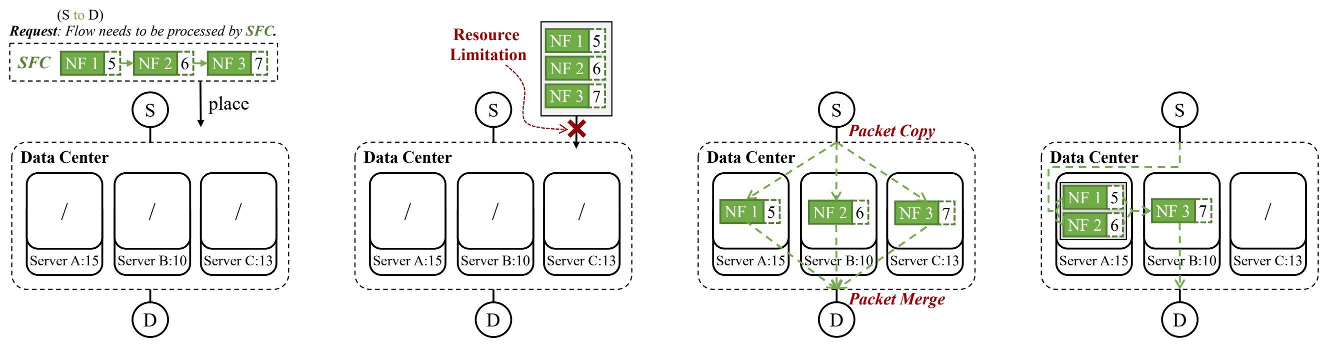
S. Xie, J. Ma, and J. Zhao are with the School of Computer Science, Fudan University, Shanghai 200438, China, and also with Shanghai Key Laboratory of Intelligent Information Processing, Shanghai 200438, China. (e-mail: xiesh19@fudan.edu.cn; jtma19@fudan.edu.cn; jzhao@fudan.edu.cn)

To solve this problem, some recent efforts aimed to apply parallelism among Virtual Network Functions (VNFs) to accelerate SFC [5], [6]. Zhang *et al.* [5] are the first to explore parallel packet processing among different VNFs. They propose Parabox which reduces SFC latency by parallelizing independent VNFs in an SFC. Sun *et al.* [6] only consider the scenario where all VNFs of an SFC are placed on the same server. They present a high-performance VNF parallelism system NFP, which can identify parallelizable VNFs in a chain intelligently and construct high-performance service graphs to reduce SFC latency effectively.

The above pioneering efforts explore the design space for VNF parallelism within SFCs. However, as we elaborate in Fig. 1, they still remain limited in flexibility and efficiency. NFP is applicable only if the whole chain is placed on one server, which greatly limits the flexibility of chain placement. As shown in Fig. 1(b), though the total resources are sufficient, the chain cannot be accommodated because no single server has enough resources. NFP's inflexibility in chain placement leaves fragmented resources unused in the system. Parabox allows a chain to span multiple servers, and ensures correctness with the help of packet copying and packet merging among different servers. One possible solution for Parabox is shown in Fig. 1(c). This method indeed reduces the latency of the SFC, but it consumes extra bandwidth. In the system with limited bandwidth resources, the benefit of SFC latency reduction may be negated by extra bandwidth consumption.

To address the above challenges and limitations, we propose a more flexible and efficient SFC parallel system, FlexChain. FlexChain allows splitting an SFC into multiple sub-chains and place them on different servers. Meanwhile, we only apply parallelism to each sub-chain separately. In other words, FlexChain does not support VNFs placed on different servers to work in parallel. Therefore, the extra bandwidth overheads introduced by distributing packet copies to multiple servers can be eliminated. Actually, we make a trade-off between the reduction of SFC latency and extra bandwidth consumption. For example, our solution for Fig. 1(a) is illustrated in Fig. 1(d). In this solution, we only parallelize NF1 and NF2. In Table I, we compare FlexChain with the state-of-the-art SFC parallel systems.

As we only consider the parallelism among VNFs on the same server (i.e., sub-chain), the reduction of SFC latency achieved by FlexChain depends heavily on how the SFC is placed on the underlying topology. Specifically, if we put as many VNFs that can run in parallel on the same server as possible, we can minimize the processing latency of SFC and maximize the benefits of parallelism. Therefore, to differentiate from existing SFC placement algorithms [7], [8], [9], [10],



(a) SFC (NF1 → NF2 → NF3) and available servers. (b) Placement on a single server. (c) Parallel across multiple servers. (d) Joint optimization of parallelism and placement.

Figure 1. An SFC placement example, where NF1, NF2 and NF3 can run in parallel. The numbers in the dotted box of NFs represent the resources required by each NF and the numbers on servers represent the available resources. NFP [6] cannot accommodate this chain because no single server has enough resources (in (b)). Parabox [5] can accommodate this chain but consume extra bandwidth due to parallelism among multiple servers (in (c)). FlexChain can accommodate this chain without extra bandwidth consumption (in (d)).

Table I
SFC Parallel System Comparison

	Parabox[5]	NFP[6]	FlexChain
Allowing SFC placed across multiple servers	✓	✗	✓
Efficient utilization of fragmented server resources	✓	✗	✓
Eliminated extra overheads of packet copying and merging in parallelism without conflicts	✗	✓	✓
Efficient utilization of bandwidth resources	✗	✓	✓

[11] that all assume that VNFs on a chain are run sequentially, we bridge the parallelism and the placement for SFCs by designing two parallelism-aware SFC placement algorithms to jointly consider SFC placement and SFC parallelism. The first one is an approximation algorithm named Recursive One-Rounding-based placement (ROR) with explicit performance guarantees. However, the time complexity of ROR is high, say, $O(mN^{3.5})$, which is only appropriate for small-scale networks. Therefore, we further design a more efficient heuristic algorithm named Parallelism-Aware Residual Capacity first placement (PARC). PARC has $O(m \log m + ml \log |V|)$ time complexity, making it suitable for large-scale data center networks.

We summarize our contributions as follows:

- We propose an SFC parallel system FlexChain which leverages VNF parallelism to accelerate packet processing in SFCs. By allowing placing an SFC on multiple servers and only applying parallelism to those VNFs placed on the same server, FlexChain is more flexible and efficient compared with existing SFC parallel systems.
- We design two parallelism-aware SFC placement algorithms, namely an approximation algorithm ROR with explicit performance guarantees and an efficient heuristic algorithm PARC for large-scale data center networks, to optimize resource utilization efficiency and reduce SFC latency when placing SFCs.
- Evaluation results show that in the latency-sensitive scenario, FlexChain combined with PARC can substantially improve the number of accepted flows by 20% on average compared with the traditional placement algorithm that does not take SFC parallelism into consideration. Meanwhile, the average latency of accepted SFC requests is also reduced by 28% on average in our system.

II. Related Work

A. VNF Parallelism

Some recent work recognized the benefits of VNF parallelism to reduce the latency of SFC. ParaBox [5] is the first to propose the concept of VNF parallelism within an SFC. It parallelizes VNF execution if they are independent of each other. For correctness, it distributes different packet copies to VNFs running in parallel and gets the final result by merging outputs from these VNFs. NFP [6] presents a comprehensive system with more detailed VNF parallelism analysis and less resource overheads compared with ParaBox. One can describe his VNF chaining intentions through a policy specification scheme provided by NFP, and then NFP will run these SFCs in an efficient way. Different from the previous two systems, Microboxes [12] uses modular μStacks to construct VNFs and redundant processing across a chain can be eliminated by sharing μStacks. Besides, to achieve higher performance gains, Microboxes also exploits parallelism for both VNFs and μStacks.

These efforts indeed reduce SFC latency and improve system performance. However, they mainly consider the case of placing the whole SFC on one server, which greatly reduces system flexibility. In contrast, we propose a more flexible and efficient SFC parallel system FlexChain, which allows placing SFC on multiple servers and eliminates extra bandwidth overheads among servers.

B. VNF Chaining and Placement

Many previous research efforts have been focused on the area of VNF placement, chaining and routing by considering different metrics to increase network efficiency, including minimizing the total cost [7], [11], [13], [14], [15], [16], [17], [18], [19], [20], minimizing the overall latency [21], [22], maximizing the total throughput [8], [19], maximizing the number of accepted SFC requests [9], [10], and others [23], [24], [25], [26]

Moens *et al.* [13] take the first step toward VNF placement within a hybrid NFV scenario where NFs are provided in either hardware or software. They formulate the VNF-P problem as an Integer Linear Programming (ILP) problem with the objective of minimizing the number of used servers. Sang *et al.* [18] also consider the VNF placement problem but under a different environment where all NFs are software instances. They discuss the problem of how to meet the demand from all of flows with a minimum cost (e.g. in terms of the number of instantiated instances). For this purpose, they prove the NP-hardness of this problem through a reduction from the set cover problem. Then, they develop a heuristic algorithm for the general problem and also identify a special case where the problem becomes submodular, which can be solved efficiently.

Instead of considering VNF placement only, some efforts consider the VNF placement and chaining jointly. Luizelli *et al.* [14] formalize the network function placement and chaining problem and propose an ILP model to solve it. They aim at minimizing the number of VNF instances mapped on the infrastructure. In order to deal with large infrastructures, they also propose a heuristic procedure for efficiently guiding the ILP solver towards feasible, near-optimal solutions. Tomassilli *et al.* [7] extend the work of Sang *et al.* [18] by considering the setting where each flow must traverse a chain of NFs instead of only one single NF. Their optimization task is to minimize the total deployment cost for all demands, which is similar to the objective of [18]

D'Oro *et al.* leverage the game theory to solve the VNF placement and chaining problem in a distributed way [15], [16]. In [15], they formulate the service chain composition problem as an atomic weighted congestion game with unsplittable flows and player-specific cost functions, and player's objective is to minimize the cost function. In [16], they exploit game theory to model interactions between users requesting network functions and servers providing these functions. The interactions among VNF servers and users requesting VNFs have been modeled as a two-stage Stackelberg game, where servers try to maximize their utility and users try to imitate other user's decisions to improve their benefit.

Xiao *et al.* [19] propose an online SFC deployment approach called NFVdeep. NFVdeep can jointly minimize the operation cost of NFV providers and maximize the total throughput of requests with the help of deep reinforcement learning. Pei *et al.* [17] also try to solve the optimal VNF placement problem via deep reinforcement learning, but with a different objective of minimize a weighted cost. The weighted cost includes the VNF placement cost, VNF instance running cost, and penalty of reject SFC requests.

Besides minimizing the deployment cost, many other efforts consider different optimization objectives. In [22], Gouareb *et al.* consider the problem of placement and routing in the edge clouds, aiming to minimize the overall latency defined as the queuing delay within the edge clouds and in network links. In [9], Sallam *et al.* consider the deployment budget and the VNF-node capacity additionally, proposing a joint VNF-nodes placement and capacity allocation algorithm that can maximize the total amount of network flows which are fully processed by the VNF-nodes. in [23], Tajiki *et al.* propose a novel resource allocation architecture to jointly manage the VNF placement and the routing in an SDN-based network, with the goal of minimizing the energy consumption. Alhussein *et al.* [25] study the online provisioning of NFV-enabled network services by considering both unicast and multicast NFV-enabled services. They propose a primal-dual based online approximation algorithm that allocates both processing and transmission resources to maximize a profit function, subject to resource constraints on physical links and NFV nodes.

There are some other work considering the placement of backup VNF. In [10], Yang *et al.* consider the fault tolerance of the network by deploying stand-by instances for each active instance. With considering request routing paths and state transfer paths to stand-by instances jointly, this work devises an efficient heuristic algorithm aiming to accept as many requests as possible, and an approximation algorithm for a special case. A similar problem is considered in [11], but with a different objective of minimizing the backup resource consumption and more comprehensive constraints of the heterogeneous resource demands for different VNFs.

Different from previous work, we consider the joint optimization problem over the parallelism and placement for SFC in data centers, with the objective of maximizing the total number of accepted SFC requests. By jointly considering SFC parallelism and SFC placement, we have the opportunity to further reduce the latency of SFC compared with the existing SFC placement algorithms, which is beneficial to latency-sensitive SFC requests.

III. FlexChain System Design

This section first describes the high-level overview of FlexChain via an illustrative example in Section III-A. We then describe the FlexChain orchestrator in Section III-B and the FlexChain parallel subsystem in Section III-C, respectively.

A. Overview

The high-level architecture of FlexChain consists of an orchestrator which is responsible for management and coordination, and parallel subsystems running in servers, which are responsible for running VNFs. The orchestrator makes decisions of SFC request acceptance and SFC deployment. The parallel subsystems receive VNF deployment commands from the orchestrator and run corresponding VNFs.

To understand how FlexChain works, consider a simple network consisting of three servers connected to the orchestrator as shown in Fig. 2. Assume that a network operator wants to deploy an SFC set consisting of 3 SFCs (as shown in Step 1 of Fig. 2). The orchestrator runs parallelism-aware placement algorithm (see Section IV and Section V) to determine which SFCs are accepted and how these accepted SFCs are placed. In order to take VNF parallelism into consideration

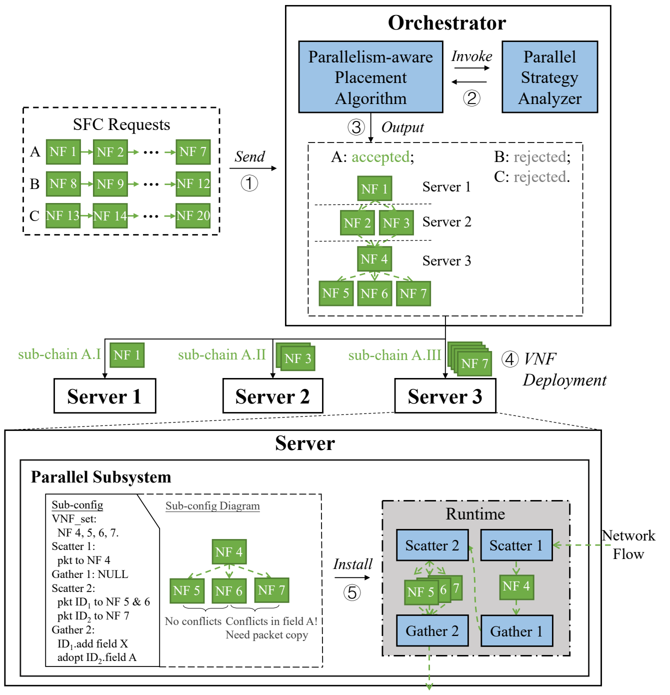


Figure 2. FlexChain overview using an example SFC request.

simultaneously, the algorithm will invoke the parallel strategy analyzer (see Section V-A) to obtain the optimal parallel strategy for each placement configuration (in Step 2 of Fig 2). According to the placement configuration generated for the accepted SFC (in Step 3), the orchestrator delivers deployment instructions (in the form of sub-configuration) to each parallel subsystem (in Step 4). Finally, based on the sub-configuration received, the parallel subsystem runs corresponding VNFs and configures the behavior of the packet scatter module and the packet gather module (in Step 5).

B. Orchestrator

The orchestrator is responsible for receiving SFC requests from network operators and placing SFCs into the underlying network. Due to the limited network resources, it is possible that some requests cannot be accepted. Therefore, the orchestrator needs to decide which SFC requests to accept and how to place these accepted SFCs into the underlying network. To solve this problem, the orchestrator uses the parallelism-aware placement algorithm, whose objective is to maximize the number of accepted requests. During the processing of the algorithm, a large number of placement configurations are generated for each SFC, of which one will be selected as the final result if the SFC is accepted. It is worth noting that a placement configuration for an SFC is a way to place the SFC in the network, consisting of the servers that each VNF is placed on and the routing path for the chain. To incorporate VNF parallelism into the placement configuration, during the generation of placement configurations, the parallel strategy analyzer is invoked to generate the optimal parallel strategy for those VNFs placed on one server. More details about the formal definition of placement configurations, the parallel

Table II
POPULAR NFS AND THEIR OPERATIONS.

NF	Products	Operation			
		R	W	A/R	D
Firewall	iptables	SIP, SPORT, DIP, DPORT	—	F	T
VPN	OpenVPN [27]	SIP, DIP, Payload	Payload	T	F
Load Balancer	Beamer [28]	SIP, SPORT, DIP, DPORT	SIP, DIP	F	F
NAT	iptables	SIP, SPORT, DIP, DPORT	SIP, SPORT, DIP, DPORT	F	F
IPS/IDS	Bro [29]	SIP, SPORT, DIP, DPORT, Payload	—	F	T

R: reading; W: writing; D: dropping packets;
A/R: adding or removing header fields.

Table III
WHETHER TWO VNF OPERATIONS ARE PARALLELIZABLE.

Parallel Case	<NF1 Operation, NF2 Operation, FieldConflict>
Parallelizable (No packet copying)	<R, R, *>; <R, W, N>; <W, R, N>; <W, W, N >; <*, D, *>
Parallelizable (Need packet copying)	<W, W, Y>; <R, W, Y>; <R, A/R, *>; <W, A/R, *>; <A/R, A/R, *>
Non-parallelizable	<W, R, Y>; <A/R, R, *>; <A/R, W, *>; <D, R, *>; <D, W, *>; <D, A/R, *>

FieldConflict: whether two VNFs perform operation on the same field.
Y: Yes; N: No; *: Any.

strategy analyzer and the two parallelism-aware placement algorithms we provided will be introduced in Section IV and Section V. After obtaining the placement configurations for all accepted SFC requests, the orchestrator will divide each configuration into sub-configurations according to the servers and send them to the corresponding parallel subsystems to deploy the SFCs.

C. Parallel Subsystem

The parallel subsystem is designed to run VNFs according to the sub-configuration received from the orchestrator. After receiving the sub-configuration, the parallel strategy needs to launch the VNFs specified in the *VNF_set* and configure the packet scatter/gather module accordingly. Next, we first introduce the details about the VNF parallelism and then the packet scatter/gather module.

1) VNF Parallelism: To run VNFs parallelly, the main challenge is to decide whether two VNFs are allowed to run in parallel. A closer look into various NFs shows that NFs may perform various actions on packet including reading (R) or writing (W) packet data, adding or removing (A/R) header fields, and dropping (D) packets. We show the popular NFs and their operations in Table II.

Inspired by the idea of Instruction-Level Parallelism, an acceleration technology widely adopted in modern CPUs [30], we determine whether two VNFs are allowed to work in parallel by analyzing various operations VNFs perform on packets and the order of operations. We use a tuple <NF1 Operation, NF2 Operation, FieldConflict> to describe the operation relationship between NF1 and NF2, where NF1/NF2 Operation represents the operation performed by NF1/NF2 and FieldConflict represents whether these two VNFs perform operations on the same field. We summarize all the possible combinations in Table III. For example, NF1 precedes NF2 in the SFC. If NF1 reads a certain packet field and NF2 writes a different field, the combination will be <R, W, N>. Clearly, NF1's read operation and NF2's write operation do not interfere with each other. Therefore, NF1 and NF2 can work in parallel without copying packets, i.e., they can perform their operations simultaneously through the same packet pointer. In contrast, if NF2 writes the same packet field read by NF1 (i.e., <R, W, Y>), in order to avoid that NF1 reads the dirty data written by NF2, we need to distribute different packet copies to them and merge outputs from them later for correctness. When multiple operations are taken by VNFs, their operation combinations will hit multiple tuples in Table III, among which the tuple closest to the bottom is used to determine their parallelism. For instance, Firewall cannot work in parallel with NAT because one of their operation relationship is <D, R, *>, which hits the *Non-parallelizable* case.

2) Packet Scatter and Packet Gather: The packet scatter module is mainly responsible for copying and distributing packets among VNFs in a server. If two VNFs are running sequentially (corresponding to the *Non-parallelizable* situation in Table III), then the packet scatter module will deliver the packet to them sequentially. Similarly, the packet scatter module will deliver the packet to those VNFs running in parallel simultaneously. Note that some VNFs running in parallel may conflict with each other (corresponding to the *Parallelizable (Need packet copying)* situation in Table III), e.g., two VNFs with relationship of *write-after-read* at the same packet field. To resolve the conflicts, we need to copy packets. In contrast, for those VNFs that can work in parallel without conflicts (corresponding to the *Parallelizable (No packet copying)* situation in Table III), we only need to delivery the same packet pointer which points to the actual packet data to them so that the overheads of copying steps are eliminated. To distinguish the above two situations, we use the configuration statement *distribute packet ID to VNF x* to instruct the packet scatter module on how to distribute packets among VNFs. We tag each packet copy with a unique id, thus, the packet scatter module can know how to copy packets and distribute them.

For correctness, when these packet copies are processed by their corresponding VNFs respectively, the processing results will be merged into a final output by the packet gather module.

In terms of the merging process, we first discuss the case where VNFs only modify packet data without adding/removing header fields or dropping packets. Assuming that P_O represents the original packet, and P_A represents the output of a certain VNF, the packet gather module uses XOR operation to get the modified bits, i.e., $P_{MA} = P_O \bigoplus P_A$. Once all the modified bits are collected, the packet gather module will XOR P_O with these modified bits to apply the modification to the original packet and obtain the final result. If there are multiple VNFs modifying the same field of a packet, we will adopt the modification made by the VNF closest to the end of the original chain for correctness. To achieve it, we use the configuration statement *adopt ID_x.field_name* to explicitly tell the packet gather module which version of modifications should be adopted when conflicts arise.

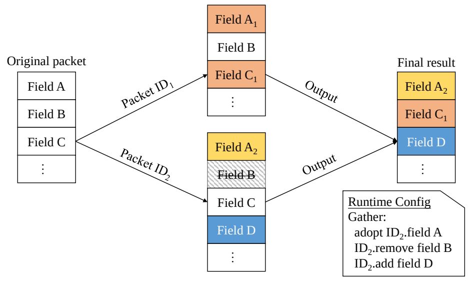


Figure 3. An example of packets merging. When VNFs add/remove packet fields or multiple VNFs modify the same packet field, we use the configuration statements instead of XOR to obtain the correct result.

In the case that VNFs perform adding or removing header fields on packets, we use the configuration statement *ID_x add field_name* or *ID_x remove field_name* to inform the packet gather module, so that the packet gather module can hide the adding or removing fields during the XOR process and apply them to the final result to guarantee correctness. As for VNFs that drop packets, we require a packet with a content length of 0 to be transmitted so that the packet gather module can recognize it and drop the corresponding packet finally. All the runtime configurations needed by the packet scatter module and the packet gather module will be configured before the system is up and remains static when system is running. An example of merging packets is shown in Fig. 3.

Comparing with traditional NFV systems that run VNF sequentially, FlexChain needs extra packet copying for VNFs running in parallel with conflicts and needs packet merging for VNFs running in parallel, which may increase the processing latency. To address this challenge, we use the high-performance Data Plane Development Kit (DPDK) [3] to implement the packet scatter/gather module. Evaluation in Section VI shows that the latency incurred by packet copying and merging in our system is extremely low compared with the processing latency of VNFs. Besides that, the packet copied for those VNFs running in parallel with conflicts will occupy extra memories. However, the situation where two VNFs run in parallel with conflicts is very rare (about 10%) according to [6], which means that the overall extra memory overheads is very small. In a word, using VNF parallelism to accelerate SFCs in our system is feasible.

IV. Placement Problem Formulation

Benefiting from the VNF parallelism, FlexChain has the opportunity to further reduce the latency of SFC by placing as many parallelizable VNFs on one server as possible. This motivates us to design parallelism-aware placement algorithms. In this section, we first present our network model and formulate the problem of joint optimization over the placement and parallelism for SFC in data centers (JPP-SFC) as an ILP problem. The key notations are summarized in Table IV. Then, the proposed algorithms are presented in Section V.

We begin to model our JPP-SFC problem as $I = (G, R)$, where G is a data center network, and R is a set of SFC requests which are known at the beginning since we only consider the offline setting. The data center network G is

Table IV
DESCRIPTION OF VARIABLES

Symbol	Description
I	An instance of JPP-SFC problem, $I = (G, R)$
	Topology
G	The underlying network topology, $G = (V, E)$
V	The set of nodes, including servers and switches
S	The set of servers, $s \in S, S \subseteq V$
E	The set of links, $e = (s_1, s_2) \in E$
t_e	The latency on link e
b_e	The available bandwidth resource of link e
b_s	The available computing resource of server s
	SFC Requests
R	The set of SFC requests, $r \in R$
λ_r	The ingress throughput of request r
τ_r	The latency constraint of request r
F_r	The network function chain of request r
C_r	The set of configurations for SFC request r
x_{ri}	1, if the configuration i is selected for request r; 0, otherwise
t_{ri}	The latency of configuration i for request r
u^s_{ri}	The computing resource consumption of server s when choosing configuration i for request r
u^e_{ri}	The bandwidth consumption of link e when choosing configuration i for request r

described by a directed graph (V, E), where V is the set of nodes (including commercial servers and switches) and E is the set of communication links. Then we denote an SFC request as $r = (s_r, d_r, F_r, \lambda_r, \tau_r)$. Each request r requires its traffic flow to be routed from a source node s_r to a destination node d_r at a given packet rate λ_r, such that its traffic flow passes all the VNFs of its required service chain F_r in order and the flow latency should be lower than τ_r. Denoting the set of all VNF types by F, the network function chain F_r is defined to be a n-tuple $(f_1, f_2, ..., f_{|F_r|})$, where $|F_r|$ is the length of F_r and $f_i \in F$ for all i.

Then given an SFC request r, depending on the network function chain it requests, there is a set of available placement configurations C_r, which are defined to be combinations of a routing path from s_r to d_r and a mapping from all $f_i \in F_r$ to the set of servers in the routing path. For the placement configuration $c_{ri} \in C_r$, the optimal parallel strategy will be generated by using Algorithm 1 (introduced in Section V-A) for those VNFs placed on the same server. According to the optimal parallel strategy and the routing path of c_{ri}, the total latency of c_{ri}, denoted as t_{ri}, including routing latency and processing latency, can be calculated. If the length of service chain is n and m routing paths are available with an average length of l, then a trivial upper bound for the size of C_r (coming from brute-force enumeration over all possibilities) is ml^n. In practice, the set C_r is smaller because some servers may be optimized for certain VNFs and some routing paths may be limited for privacy. Even in an unlimited situation, we use a k-th method to reduce the size of set C_r, by only considering the first k-th optimal configurations in order of latency. The method only makes a small difference to the final result.

The goal of JPP-SFC is, given SFC requests and the network model, to determine i) the set of requests accepted and ii) how the accepted SFCs are placed in the network topology in order to accept as many SFC requests of R as possible while meeting the requests' latency constraints and the network resource constraints. These constraints are described as follows.

Given a request r, each configuration $c_{ri} \in C_r$ has a binary variable x_{ri} to denote whether we choose the configuration c_{ri} for request r. Thus we have to ensure,

$$x_{ri} \in \{0, 1\}, \quad \forall r \in R. \tag{1}$$

$$0 \le \sum_i^{|C_r|} x_{ri} \le 1, \quad \forall r \in R. \tag{2}$$

We allow the case of $\sum_i x_{ri} = 0$, which corresponds to the rejection of request r.

Each configuration $c_{ri} \in C_r$ is also associated with resource usages over the network. For instance, if two VNFs from a service chain are placed on server s and s', the placed VNFs will incur CPU usage on s and s', and bandwidth cost on the routing path from s to s'. For ease of presentation, we use the CPU usage as a representative of the computing resource constraints. Other type of resources such as memory usage or disk I/O cycles can be similarly addressed. Thus, we need to ensure,

$$\sum_r^R \sum_i^{|C_r|} x_{ri} \cdot u^s_{ri} \le b_s, \quad \forall s \in S. \tag{3}$$

$$\sum_r^R \sum_i^{|C_r|} x_{ri} \cdot u^e_{ri} \le b_e, \quad \forall e \in E. \tag{4}$$

Besides, to fulfill the latency constraint of each request, we have:

$$\sum_i^{|C_r|} x_{ri} \cdot t_{ri} \le \tau_r, \quad \forall r \in R. \tag{5}$$

We are now ready to formulate JPP-SFC, a problem of maximizing the total number of accepted SFC requests in data centers, as an optimization problem:

$$\begin{aligned} \max \quad & \sum_r^R \sum_i^{|C_r|} x_{ri} \\ \text{s.t.} \quad & (1) \text{ - } (5). \end{aligned} \tag{6}$$

Theorem IV.1. *The JPP-SFC problem defined in Eq.(6) is NP-hard.*

Proof. First of all, coming back to the Zero-One Equation (ZOE) problem where given an input matrix $\boldsymbol{A}$ whose entries are zero-one, we want to decide whether there exist decision variables $\boldsymbol{x}$ consists of zero-one values such that the linear system

$$\boldsymbol{Ax} = \boldsymbol{1}$$

holds. It is known the ZOE problem is NP-complete [31].

ZOE problem can be reduced to the following formulation:

$$\begin{aligned} \max \quad & \boldsymbol{1}^T \boldsymbol{Ax} \\ \text{s.t.} \quad & x_i \in \{0, 1\}, \\ & \boldsymbol{Ax} \le \boldsymbol{1}, \end{aligned}$$

where x_i is the i-th entry of $\boldsymbol{x}$. After solving this problem, if the optimal objective value is $|\boldsymbol{x}|$, then the only possibility is $\boldsymbol{Ax} = \mathbf{1}$ holds for the optimal $\boldsymbol{x}$, which implies $\boldsymbol{Ax} = \mathbf{1}$ is feasible. If the optimal objective value is less than $|\boldsymbol{x}|$, then $\boldsymbol{Ax} = \mathbf{1}$ cannot be feasible.

We can find that the reduced ZOE is a special case of JPP-SFC without considering the constraints (3) - (5). As a result, ZOE can be reduced to JPP-SFC, which proves JPP-SFC is NP-hard. □

Due to the computational complexity of the proposed ILP model and given the available computing processing power, the optimization model imposes a limitation in the actual data center network environment. Therefore our goal is to design a practical approximation algorithm that solves JPP-SFC with an explicit performance guarantee and an efficient heuristic algorithm for large-scale data center networks.

V. Proposed Algorithms

In this section, we first introduce our parallel strategy analyzer which is used to generate the optimal parallel strategy for a given SFC (or a sub-chain). Then, for solving JPP-SFC, we present our approximation algorithm with its approximation ratio, and our heuristic algorithm for large-scale data center networks.

A. Parallel Strategy Analyzer

The parallel strategy analyzer takes each sub-chain placed in one server as input, analyzes dependencies among them, and then generates the optimal parallel strategy with minimum latency for them.

Based on the VNF parallelism analysis presented in Section III-C1, we maintain two tables for the parallel strategy analyzer. They are an *action table* which maps each VNF to its operation set (i.e., Table II) and a *parallel case table* that maps each <NF1 Operation, NF2 Operation, FieldConflict> tuple to its parallel case (i.e., Table III). For a given VNF pair, we can obtain each VNF's operation set from the *action table*, and then construct all their <NF1 Operation, NF2 Operation, FieldConflict> tuples. Once we have all the tuples, we can decide their parallelism according to the *parallel case table*. When a new VNF is added to our system, the *action table* needs to be updated by inserting the new VNF and its operation set, while the *parallel case table* remains fixed. Some VNF providers might not wish to disclose the implementation details of their VNFs, which means that we cannot obtain the operation set of these VNFs (i.e., they are black-box VNFs). In such a situation, we simply add a tag to the black-box VNF to tell the parallel strategy analyzer explicitly that the VNF cannot run in parallel with any other VNFs. Although this conservative method may miss some potential parallel VNFs, it can ensure the correctness of packet processing logic in the chain. And in this way, the latency of SFCs in our system is always not worse than that of all VNFs running in sequential. According to these two tables, the parallel strategy generator can generate a parallel strategy for a given SFC. Then, the generated parallel strategy will be converted into configuration

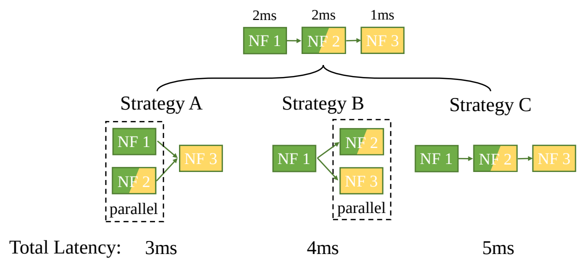


Figure 4. An example of obtaining optimal parallel strategy by traversing, where VNFs in the same color can work in parallel. After traversing all the strategies, strategy A will be chosen as the optimal one.

statements which are used to instruct the packet scatter module and the packet gather module.

Conflict when parallelizing VNFs: When generating a parallel strategy, it is desirable to parallelize VNFs as much as possible to minimize the processing latency. However, in the process of obtaining the optimal parallel strategy, there may exist situations as illustrated in Fig. 4, where NF1 can work in parallel with NF2 and NF2 can work in parallel with NF3, but NF1 cannot work in parallel with NF3. In this situation, there are multiple parallel strategies and finding the optimal parallel strategy with minimum processing latency is NP-hard as proven in Theorem V.1.

Theorem V.1. *Finding the optimal parallel strategy with minimum processing latency for an SFC is NP-hard.*

Proof. We start the proof by introducing the classic weighted mutually exclusive set cover problem. We consider a set of m elements $U = \{e_1, e_2, ..., e_m\}$ and a collection $S = \{s_1, s_2, ..., s_n\}$ of n subsets of U. The union of all the subsets in S equals U, i.e., $\cup_{i=1}^{n} s_i = U$. Each $s_i \in S$ has a real number weight denoted by $W(s_i)$. Then the weighted mutually exclusive set cover problem can be represented as:

$$\begin{aligned} \min \quad & \sum_{i=1}^{|X|} W(x_i) \\ \text{s.t.} \quad & X \subset S, \\ & \bigcup_{i=1}^{|X|} x_i = U, \\ & x_i \cap x_j = \emptyset, \quad \forall x_i, x_j \in X, i \neq j. \end{aligned}$$

It has been proven that the mutually exclusive set cover problem is NP-hard [32].

Given an arbitrary instance (U, S, W) of the weighted mutually exclusive set cover problem, we construct an instance (F, O, L) of finding the optimal parallel strategy problem. Each VNF $f_i \in F$ corresponds to an element of $e_i \in U$, and the index i represents the execution order of the VNF. For each $s_j \in S$, we construct an execution strategy of o_j for all elements (i.e., VNFs) in s_j. In o_j, VNFs with consecutive indexes is executed in parallel, while others are executed in sequence in ascending order of the indexes. Meanwhile, the weight of s_j corresponds to the processing latency of o_j, i.e., $L(o_j) = W(s_j)$. The objective of finding the optimal parallel strategy is to find the execution strategy for all VNFs in F with minimum processing latency. It is easy to see that this

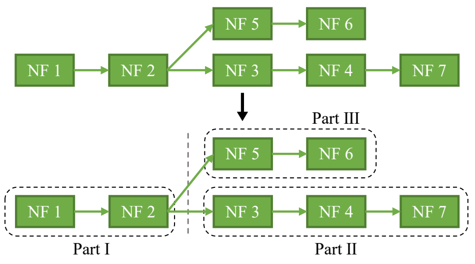


Figure 5. The method to deal with a bifurcated SFC. We divide the bifurcated SFC into multiple parts and apply parallelism to each part separately.

is equivalent to finding the subset X of S such that the union of X equals the universe set U and the sum of weights of subsets in X is minimal. Therefore, finding the optimal parallel strategy with minimum processing latency for an SFC is also NP-hard. □

Although finding the optimal parallel strategy for an SFC is NP-hard, the vast majority of current SFCs do not exceed 7 in length [33], [34]. Therefore, we can traverse all possible parallel strategies to get the optimal one. The pseudo-code of our traversal algorithm is articulated in Algorithm 1. For each VNF f, we check whether f can work in parallel with the next VNF f.next. If it can, we first consider the situation when they work in parallel (line 7-9), and then the situation when they run sequentially (line 10); Otherwise, we only consider the sequential situation and move to the next VNF (line 10). When we consider the situation where two VNFs run in parallel, we need to combine these two VNFs into one mock VNF logically by combining their operation sets (line 7-8) before we move on. More specifically, the combing process is done by setting the operation set of the new_nf as the union of the operation set of f and f.next, i.e., new_nf.operation_set = f.operation_set ∪ f.next.operation_set. Therefore, when we reach the end of the chain, the parallel strategy can be obtained from the structure of the chain. After we traverse all the possible parallel strategies, we can obtain the optimal parallel strategy with the minimum processing latency.

Considering the example shown in Fig. 4, where every VNF's packet processing latency is shown above the VNF and those VNFs painted in the same color can work in parallel. Our traversal algorithm will first consider parallel strategy A, then strategy B and finally strategy C. After all the parallel strategies have been compared, our algorithm will get the optimal one, which is strategy A in this example.

Note that an SFC may bifurcate (e.g. a Load Balancer or a Deep Packet Inspector) as shown in Fig. 5, where different packets may be delivered by NF2 to different sub-branches of the SFC. In such a situation, we will pre-process the chain before finding the optimal parallel strategy for it. Specifically, we divide the chain into three parts at the point that bifurcates and find the optimal parallel strategy for each part separately. In this way, we guarantee that the VNFs in Part I will not be run in parallel with the VNFs in Part II and Part III, and, the packets sent to Part II (III) will not be processed by the VNFs in Part III (II), which retains the semantics of bifurcation and ensures the correctness of our system.

Algorithm 1 Traversal algorithm for optimal parallel strategy

```
Input: sub-chain $c$ placed in one server
Output: optimal parallel strategy with minimum latency
 1: function GETOPTSTRATEGY(f, chain)
 2:     if f.next == NULL then
 3:         if chain.latency < optimal_strategy.latency then
 4:             optimal_strategy ← chain
 5:     else
 6:         if f can work in parallel with f.next then
 7:             new_nf ← combining f and f.next
 8:             new_chain ← replace f, f.next with new_nf
 9:             getOptStrategy(new_nf, new_chain)
10:         getOptStrategy(f.next, chain)
11: optimal_strategy ← $c$
12: getOptStrategy($c$.header, $c$)
13: return optimal_strategy
```

In the worst case, our algorithm needs to traverse 2^{n-1} cases to get the optimal parallel strategy, where n is the length of the chain. Although the computational complexity is exponential, our algorithm will not take a long time to find the optimal parallel strategy, since n is generally less than 7 as mentioned above.

B. Recursive One-Rounding-Based Placement Algorithm

It is intuitive to solve JPP-SFC, a classic integer linear programming problem by a two-phase procedure, Randomized Rounding [35] (denoted as RR). In the first phase of RR, the integral constraints are relaxed and the resulting linear program is solved to get the optimal fractional results. In the second phase, the optimal fractions obtained from phase one are used as probability to randomly choose a configuration for each request. However, it has been proven that the expected number of violated constraints achieved by RR is $O(\log n)$. In other words, assuming the network size n is 128, the expected probability of constraint violation is 99.925%, which may lead to network congestion or cluster failure in real-world scenario.

Therefore, we propose a more reliable and efficient approximation algorithm, Recursive One-Rounding-based placement (ROR) that ensures no constraint is violated. Furthermore, ROR achieves a better performance compared with RR, which is proved in Theorem V.2. The simplified pseudo-code of ROR is shown in Algorithm 2.

In phase one (line 1-4), we formulate the JPP-SFC as an ILP problem, and solve the relaxed LP problem. First, for a request r, we generate its configuration set by searching paths from s_r to d_r in the network topology, and enumerating how all VNFs $f \in F_r$ are placed on each searched path. Then, algorithm 1 (mentioned in Section III) will be used to generate the optimal SFC parallel strategy and compute the latency t_{ri} for each configuration. To avoid exponential explosion of the configuration set, we use the k-th optimal method (mentioned in Section IV). However, a certain level of performance loss is incurred because some configurations will not be considered. In general, it is a trade-off between performance and execution time. After generating configuration sets, we formulate the ILP

Algorithm 2 ROR algorithm

```
Input: An instance of JPP-SFC problem $I = (G, R)$, including a graph $G = (V, E)$, a set of requests $R$
Output: Accepted configurations $\overline{C}$
 1: Generate $k$-th optimal configurations $C_r$ for all $r \in R$
 2: $C \leftarrow \cup_{r \in R} C_r$
 3: Formulate $I$ to an ILP model with variables $X$.
 4: Solve the relaxed LP model for the optimal $\hat{X}$
 5: $\overline{C} \leftarrow \{c_{ri} | \hat{x}_{ri} = 1, \hat{x}_{ri} \in \hat{X}\}$
 6: if $\overline{C}$ is Empty then
 7:     $\overline{C} \leftarrow$ RandomizedRounding$(G, R, C, \hat{X})$
 8: else
 9:     $I' \leftarrow I$ excludes the resources and requests of $\overline{C}$.
10:     $\overline{C} \leftarrow \overline{C} \cup$ ROR$(I')$.
```

model, relax the integral constraints and solve it for optimal fractional $\hat{X}$.

In phase two (line 5-10), we determine which requests to accept and their corresponding configurations in a recursive fashion. Here we define that accepting a configuration is to accept its corresponding request and to choose this placement configuration for the accepted request. After obtaining $\hat{X}$ in phase one, to avoid constraint violation, we only accept $\hat{C}_1 = \{c_{ri} | \hat{x}_{ri} = 1, \hat{x}_{ri} \in \hat{X}\}$, a subset of configurations that will be definitely accepted if using RR, but discard those fractional x_{ri}. This is where "One-Rounding" comes from. In this way, there is a high probability that resources of the underlying topology remain under-utilized, so we run ROR recursively to fully use the network resources. We remove the resources utilized by $\hat{C}_1$ from the network topology and remove the corresponding requests from R to get the subproblem I', and then solve it by run ROR again (line 9-10). The recursion stops when there is no $\hat{x}_{ri} = 1$ for $\forall \hat{x}_{ri} \in \hat{X}$ and ends up with RR to further fully utilize the remaining network resources. During RR, we reject the configurations that lead to constraint violation.

Theorem V.2. *The approximation ratio of ROR is no greater than $O(\log n)$, where n is the number of constraints*

Proof. RR has been proved to guaranteed to achieve an objective value within $O(\log n)$ factor of the optimal [35], where n is the number of constraints. Later, we prove in Lemma V.3 that the objective value achieved by our ROR is never worse than RR. □

Lemma V.3. *For a JPP-SFC problem instance I,*

$$\kappa(\overline{C}^I_{ROR}) \geq \kappa(\overline{C}^I_{RR}), \tag{7}$$

where $\overline{C}^I_{ROR}$ represents the solution of ROR, $\overline{C}^I_{RR}$ represents the solution of RR, and $\kappa(C)$ represents the corresponding objective value of C.

Proof. We define that $\hat{X}^I$ is the optimal fractional solution calculated by an LP solver, and $\hat{C}^I_1 = \{\hat{c}_{ri} | \hat{x}_{ri} = 1, \hat{x}_{ri} \in \hat{X}^I\}$. In this way, we have $\overline{C}^I_{\text{ROR}} = \begin{cases} \overline{C}^I_{\text{RR}} & , \hat{C}^I_1 = \emptyset \\ \hat{C}^I_1 \cup \overline{C}^{I'}_{\text{ROR}} & , \hat{C}^I_1 \neq \emptyset, \end{cases}$ where I' represents the subproblem formulated by removing from I the network resources consumed by $\hat{C}^I_1$ and the accepted requests of $\hat{C}^I_1$.

Since $\overline{C}^I_{\text{ROR}}$ has two cases, we do our proof in two cases too. For the first case, i.e., $\overline{C}^I_{\text{ROR}} = \overline{C}^I_{\text{RR}}$, we have $\kappa(\overline{C}^I_{\text{ROR}}) = \kappa(\overline{C}^I_{\text{RR}})$. Thus Eq.(7) always holds for the first case.

For the second case, i.e., $\overline{C}^I_{\text{ROR}} = \hat{C}^I_1 \cup \overline{C}^{I'}_{\text{ROR}}$, we first prove that when $\overline{C}^I_{\text{ROR}} = \hat{C}^I_1 \cup \overline{C}^{I'}_{\text{RR}}$, Eq.(7) always holds. Then Eq.(7) can be proven to be true in the second case by mathematical induction. In a general ILP problem, intuitively we have $\overline{C}^I_{\text{RR}} = \hat{C}^I_1 \cup \overline{C}^{I'}_{\text{RR}}$. However, with the k-th optimal method, the equation does not always hold. In I', as resources in the network become increasingly scarce, the configurations with more complex routing paths and placement strategies will be explored as the first k-th configurations. In other words, I' increases the feasible solution space. If we set the determining variables x_{ri} of these newly joined configurations to 0, we can still get the original problem setting and get the original optimal solution, i.e., $\overline{C}^I_{\text{RR}} - \hat{C}^I_1$. Therefore, the final objective value will not get worse, or even get better due to the expanded configuration set, i.e., Eq.(7) always holds for the second case.

In conclusion, the recursively one rounding steps improves the empirical performance as a result of exploring more detailed configuration in subproblems, although the guaranteed approximation ratio remains the same. □

Algorithm Complexity. The highest order of magnitude in ROR is the subroutine of solving the LP model. If an interior point method with complexity $O(N^{3.5})$ is used for solving the LP problem, the complexity of ROR is $O(mN^{3.5})$, where N is the number of constraints and m is the number of requests which represents the number of recursions in the worst case. In JPP-SFC, the number of constraints N is obtained by suming up the number of links $|E|$, the number of servers $|V|$ and the number of requests m.

C. Parallelism-Aware Residual Capacity First Placement Algorithm

Given the limited computing processing power, the computational complexity of ROR imposes a limitation on scaling it to large-scale data center networks. Therefore, we further propose an efficient heuristic algorithm, Parallelism-Aware Residual Capacity first placement (PARC), which takes both the VNF parallelism and the residual resource capacity of the network into consideration. We show the pseudo-code of PARC in Algorithm 3.

Recalling that our objective is to maximize the total number of accepted SFC requests and a request can be accepted only when all its constraints can be fulfilled, PARC tries to reach this goal through two aspects: a) make full use of network resources to accommodate more requests; b) make VNFs in an SFC run as parallel as possible to increase the probability of meeting its latency constraint. In detail, PARC works as follows:

Step 1: Sort all requests in ascending order according to their resource consumption. For each request r, we generate a new request r^* by applying the optimal parallel strategy

Algorithm 3 PARC Algorithm

```
Input: An instance of JPP-SFC problem $I = (G, R)$
Output: Accepted configurations $\overline{C}$
1:  $\overline{C}$ = []
2:  sort $R$ by its consuming resources
3:  for all $r \in R$ do
4:      $r^* \leftarrow$ apply the optimal parallel strategy to $r$
5:      $c \leftarrow$ try to put $r^*$ on $G$ by residual capacity first
6:      if $c$ is not feasible then
7:          $c \leftarrow$ try to put $r$ on $G$ by residual capacity first
8:          if $c$ is not feasible then
9:              continue                          ▷ reject the request
10:     $\overline{C}$.append($c$)
11:     $G \leftarrow G$ - the resources consumed by $c$
    return $\overline{C}$
```

analyzed by Algorithm 1 to r. More specifically, in the request r^*, the VNFs that run in parallel according to the strategy must be placed in the same server to ensure the optimal strategy is applied.

Step 2: For each request r, we first check whether the corresponding r^* can be placed into the topology. More specifically, we adopt a residual capacity first greedy algorithm. That is, we repeatedly select the server with the most residual computing resources and place as many VNFs as possible in the server until all VNFs in r^* is placed and no constraint is violated. If r^* can be placed into the topology, we accept the corresponding request r.

Step 3: Otherwise, we will check whether the original request r can be placed in the topology by the same algorithm mentioned in Step 2. If r can be placed, accept r. Otherwise, reject r. Note that when we decide to accept a request r, the resources consumed by r need to be trimmed from the topology before we consider the next request.

Step 4: Repeat Step 2 and 3 until all the requests are checked.

Algorithm Complexity. The time complexity of PARC is divided into two parts. The first part is the subroutine of sorting m requests in the order of resource utilization, of which the complexity is $O(m \log m)$. The second part is the subroutine of checking whether the requests r^* and r can be placed in the topology, of which the complexity is $O(ml \log |V|)$. Here, $|V|$ represents the number of servers and l represents the average length of an SFC. Therefore, the total computational complexity of PARC is $O(m \log m + ml \log |V|)$, which is much lower than that of ROR.

VI. Evaluation

In this section, we first evaluate the parallel overheads introduced by FlexChain parallel subsystem, and then we observe and analyze the behavior of the proposed ROR algorithm and PARC algorithm.

We perform our experiments over three typical data center architectures: VL2 [36], Fat-Tree [37] and BCube [38]. In these networks, the link latency is set to $0.05ms$ and the bisection link bandwidth is 1Gbps. The SFC requests are generated according to realistic SFC in data centers [34], in which the length of SFC is ranging from 3 to 7. The VNFs in SFC are uniformly chosen at random from a set of 30 commonly-deployed VNFs including 5 typical VNFs (i.e., firewall, NAT, IPS/IDS, load balancer, VPN) in Table II and other service-customized VNFs. According to [39], the processing latency of a packet for each VNF is randomly drawn from $0.045ms$ to $0.3ms$. We simulate different constraints for different requests in terms of latency and throughput according to real-world trace [40]. The latency constraints are ranging from $0.5ms$ to $1.5ms$. The running time is obtained based on a server with a 3.40GHz Intel i7 Quad-core CPU and 8 GB RAM.

A. Parallel Overhead

It is first critical to understand the parallel overheads of FlexChain as it relates to how we calculate the total latency when VNFs running in parallel and the potential performance improvement our algorithms can achieve. Compared with sequential execution, the parallel overheads come mainly from two aspects: packet copying and packet merging introduced by the packet scatter module and the packet gather module. Therefore, we measure the average time spent on this two operations by running two VNFs in parallel and each VNF has its own CPU core.

As shown in Fig. 6, time overheads introduced by copying and merging are positively related to the size of packets. The average time for copying one packet is between 206~755 ns, while the average time for merging two packets is between 184~441 ns. Compared with VNF's processing latency (between 0.045~0.3 ms), the parallel overheads are so insignificant that we can simply ignore them. Therefore, for those VNFs running in parallel, the total latency is equal to the maximum processing latency of all VNFs. This result guarantees the effectiveness of our system and algorithms.

B. Parallelism-Aware Placement Algorithm Evaluation

We first evaluated the impact of k on the performance of our approximation algorithm (i.e., ROR). As mentioned in Section IV, we use a k-th method to reduce the number of configurations, by only considering the first k-th optimal configurations in order of latency. Given a set of SFCs and a VL2 network topology, we vary the value of k between 16 and 2048 and evaluate both the algorithm running time and the number of accepted SFCs. The results are shown in Fig. 7. Since the results are similar for the three topologies, we only show the result for VL2 topology due to space limitations. From Fig. 7 we can see that when k is equal to 256, the running time of ROR is minimal and the number of accepted requests is maximal. Therefore, we set k to 256 in subsequent experiments.

We also measured the impact of the parallel probability between VNFs on our algorithms. The parallel probability between VNFs can be calculated by dividing the number of parallelizable VNF pairs by the total number of VNF pairs. To vary the parallel probability, we simulated other 30 VNFs and use a map to judge whether two VNFs can run in parallel, so we can change the total parallel probability easily by

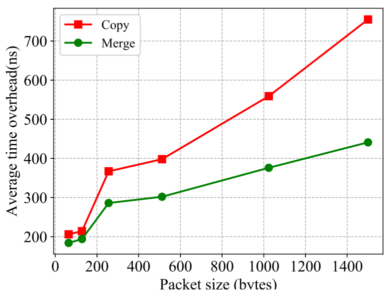


Figure 6. Copying and Merging overheads of parallelism.

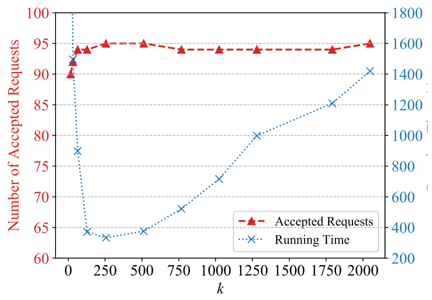


Figure 7. The effect of k on the performance of ROR.

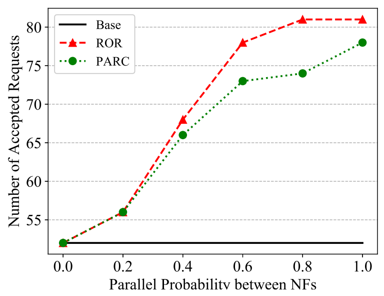
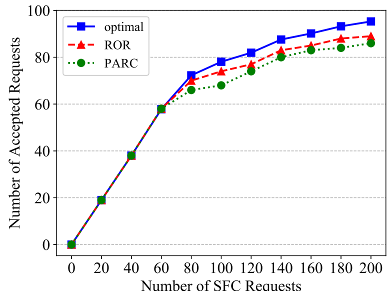
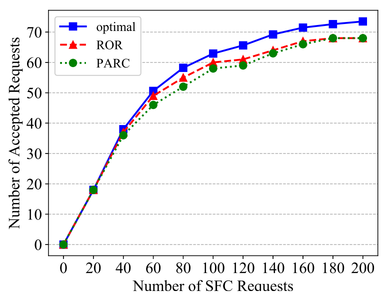


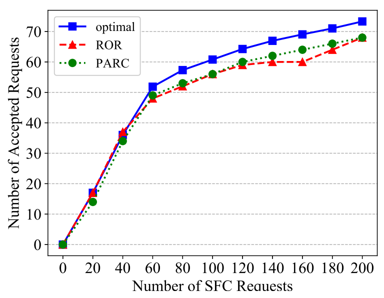
Figure 8. The effect of parallel probability between VNFs on the performance of our algorithms



(a) VL2 (96 servers).



(b) FatTree (63 servers).



(c) BCube (64 servers).

Figure 9. Accepted requests for our algorithms in three different data center topologies.

changing the map. During the evaluation, we guarantee that the algorithms always receive the same 100 SFC requests, but vary the total parallel probability from 0.0 to 1.0. As shown in Fig. 8, we can find that the greater the parallel probability between VNFs is, the higher profit our algorithm can achieve, compared with the method that places VNFs without considering parallelism (i.e., the Base line in Fig. 8). However, in real scenarios, the parallel probability cannot reach 100%. In the subsequent evaluations, we use the original VNF set and judge whether any two VNFs can work in parallel by analyzing their operation relationships. The total parallel probability of the original VNF set is 56.3%.

Next, we evaluate the performance of our proposed two algorithms, ROR and PARC, on different network topologies. In each topology, we vary the number of incoming SFC requests between 20 and 200. Fig. 9 presents the experimental results. The optimal result obtained by $\hat{X}$ in our Algorithm 2 is included to indicate an upper bound of this problem. We remark from the figures that when the total demand saturates the network, the gain achieved by our algorithms compared with the optimal is more than 85% in all cases. Moreover, the ROR algorithm is usually closer to the upper bound. As expected, our PARC algorithm cannot always achieve the performance of the ROR algorithm, but the performances of the two algorithms are very close. In particular, when the network is saturated, the PARC algorithm always achieves at least 90% of the ROR optimal. We also notice that in BCube, PARC performs better than ROR in some cases. The reason is that BCube is a server-centric network topology where each server has sufficient bandwidth resources, thus, the residual computing capacity first feature of PARC makes it achieve a nearly ideal placement.

Table V
RUNNING TIME OF OUR ALGORITHMS

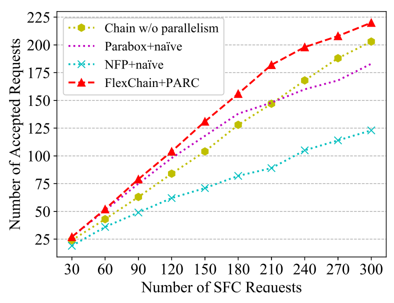
Topology Size	16	54	128	250
ROR	53.4s	195.8s	941.2s	5592.8s
PARC	0.038s	0.54s	1.49s	3.78s

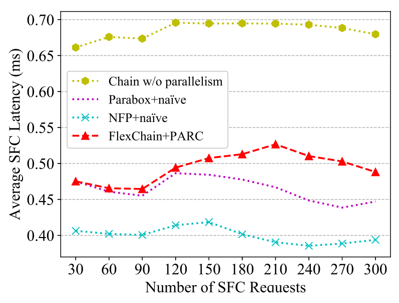
Finally, we examine the running time for our algorithms. The evaluation here focuses on the VL2 topology and similar results can be observed for other topologies. As shown in Table. V, the running time of both ROR and PARC increases in polynomials as network size grows, and PARC is always three orders of magnitude faster than ROR, indicating its applicability in large-scale networks.

C. Performance Comparison

In this section, we compare our deployment approach (i.e., **FlexChain+PARC**) with other approaches:

- **Chain w/o parallelism:** a traditional placement algorithm that does not take SFC parallelism into consideration.
- **Parabox+naïve:** since there is no placement algorithm for Parabox at present, we implement a naïve placement algorithm for it by modifying our PARC algorithm. In this


(a) Total number of accepted requests


(b) Average latency of accepted requests

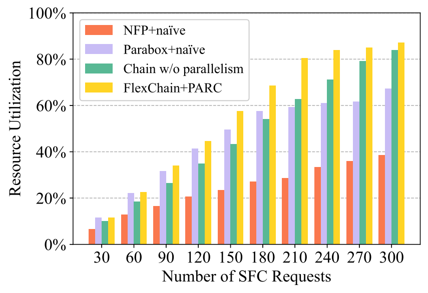

(c) Resource utilization of the network

Figure 10. Comparison between our deployment approach and other approaches in VL2 topology.

placement algorithm, VNFs placed on different servers are also running in parallel as assumed in Parabox [5].

- **NFP+naïve:** similarly, we implement a naïve placement algorithm for NFP by modifying PARC. It only allows placing an entire SFC on one server as required by NFP [6].

The evaluation here focuses on a larger and more realistic VL2 topology, consisting of 344 nodes (including 256 servers).

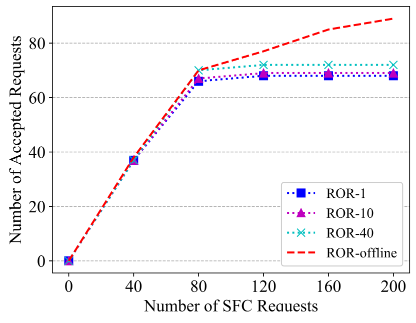
Fig. 10 presents the experimental results. We can see from Fig. 10(a) that our approach consistently accepts more SFC requests than other approaches. In a latency-sensitive scenario where most SFC requests are required to be processed in a very short time, if we place SFCs without taking SFC parallelism into consideration, some requests will be rejected due to latency constraint violation. Therefore, *Chain w/o parallelism* accepts about 20% fewer SFC requests than our approach. In *Parabox+naïve*, the VNF parallelism among different servers may introduce extra bandwidth consumption, which leads to the low SFC request acceptance compared with our *FlexChain+PARC* (11% on average). As for *NFP+naïve*, accepting fewest SFC requests among all approaches reflects its defect that an SFC can only be entirely placed on one server. There may exist servers whose remaining resources are enough to host a few VNFs but not a whole chain, leading to a waste of fragmented resources. In contrast, *FlexChain+PARC* allows splitting an SFC into multiple sub-chains and places them on different servers to make full use of these fragmented resources. Our flexibility contributes to a higher acceptance ratio of SFC requests.

We also calculate the average latency of those accepted SFC requests in Fig. 10(b). The average latency of *Flex-Chain+PARC* is about 28% lower than the *Chain w/o parallelism*, because we consider parallelism among VNFs. Compared with *NFP+naïve* and *Parabox+naïve*, the average latency of *FlexChain+PARC* is slightly higher (22% on average and 7% on average respectively). The reason is that some SFCs are split into multiple sub-chains and placed on different servers in our *FlexChain+PARC* approach, and those VNFs placed on different servers cannot work in parallel. However, in our setting, meeting the latency constraints of SFC requests is enough and it is unnecessary to pursue the minimum. The simulation result that *FlexChain+PARC* accepts more SFC requests suggests our proposed system and algorithms achieve a good trade-off between the average SFC latency and the number of accepted SFC requests.

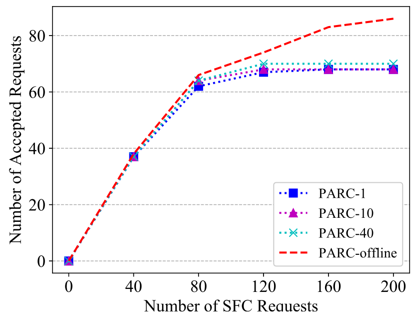
Another interesting metric to investigate is the network resource utilization of each deployment scheme. Here we choose the CPU usage as the representative of all network resources. Fig. 10(c) shows the network resource utilization as the number of SFC requests increases. From the figure, we can see that when the number of accepted requests is close to saturation, the resource utilization of our approach (*FlexChain+PARC*) is 87.2%, while the resource utilization of *NFP+naïve* and *Parabox+naïve* are only 38.5% and 67.3%. A prominent improvement of resource utilization is made by *FlexChain+PARC* among all the approaches that consider VNF parallelism, because *FlexChain+PARC* accepts more SFC requests with high resource requirements and makes full use of fragmented resources. Even compared with *Chain w/o parallelism*, the resource utilization of our deployment approach is also improved. This is because *FlexChain+PARC* accepts more requests which have rigorous latency constraints to occupy the network resource.

D. Online Scenarios

Though our proposed algorithms (ROR and PARC) are designed for offline settings, we can add batch processing to our algorithms to make them applicable in online settings. The way of batch processing is: our placement system does not process all incoming SFC requests in time, but temporarily stores the requests. When the number of stored SFC requests reaches the batch threshold, we use our algorithm to place these SFCs. In this way, the size of the batch matters a lot, so we evaluated the impact of the batch size on the performance of our online algorithms, and compared the performance of our online algorithms and offline algorithms. We performed this evaluation on topology VL2, and the SFC requests arrive in an online manner (i.e., the information of all the SFC requests is unknown in advance). The results in Fig. 11 show that as the batch size grows, the performance of our online algorithms increases slightly. This is because that the larger the batch size is, the more SFCs can be considered simultaneously by our online algorithms, so that a set of placement configurations with higher resource utilization can be chosen. As more resources are used, more SFC requests can be accepted. Not



(a) ROR online



(b) PARC online

Figure 11. Accepted requests for our algorithms in online settings. The number after algorithm name represents the batch size it uses.

surprisingly, the performance of our offline algorithms is much better than that of the online algorithms when the total number of SFC requests is large. Therefore, the network operator can use the offline algorithms to reallocate SFCs to accept more SFC requests when the remaining network resources are insufficient.

VII. Conclusion

In this paper, we propose a flexible SFC parallel system FlexChain, which allows VNFs on the same server to work in parallel, to reduce the processing latency of SFCs. FlexChain also allows placing SFCs on multiple servers for elastic scaling. To leverage the benefits of VNF parallelism, we further consider the problem of joint optimization over SFC placement and parallelism in data centers. Due to its NP-hardness, we proposed a parallelism-aware approximation placement algorithm with performance guarantees, and an efficient heuristic algorithm especially for large-scale data center networks. Our evaluation shows that the performance obtained by FlexChain combined with our algorithms outperforms existing solutions that consider placement and parallelism separately.

References

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

[2] P. Quinn and T. D. Nadeau, "Problem statement for service function chaining," *RFC*, vol. 7498, 2015.

[3] Intel, "Data plane development kit." [Online]. Available: https://www.dpdk.org/

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

[5] Y. Zhang, B. Anwer, V. Gopalakrishnan, B. Han, J. Reich, A. Shaikh, and Z. Zhang, "Parabox: Exploiting parallelism for virtual network functions in service chaining," in *Proc. ACM SOSR*, 2017.

[6] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu, "NFP: enabling network function parallelism in NFV," in *Proc. ACM SIGCOMM*, 2017.

[7] A. Tomassilli, F. Giroire, N. Huin, and S. Pérennes, "Provably efficient algorithms for placement of service function chains with ordering constraints," in *Proc. IEEE INFOCOM*, 2018.

[8] L. Guo, J. Z. T. Pang, and A. Walid, "Joint placement and routing of network function chains in data centers," in *Proc. IEEE INFOCOM*, 2018.

[9] G. Sallam and B. Ji, "Joint placement and allocation of virtual network functions with budget and capacity constraints," in *Proc. IEEE INFOCOM*, 2019.

[10] B. Yang, Z. Xu, W. K. Chai, W. Liang, D. Tuncer, A. Galis, and G. Pavlou, "Algorithms for fault-tolerant placement of stateful virtualized network functions," in *Proc. IEEE ICC*, 2018.

[11] J. Zhang, Z. Wang, C. Peng, L. Zhang, T. Huang, and Y. Liu, "RABA: resource-aware backup allocation for a chain of virtual network functions," in *Proc. IEEE INFOCOM*, 2019.

[12] G. Liu, Y. Ren, M. Yurchenko, K. K. Ramakrishnan, and T. Wood, "Microboxes: high performance NFV with customizable, asynchronous TCP stacks and dynamic subscriptions," in *Proc. ACM SIGCOMM*, 2018.

[13] H. Moens and F. De Turck, "Vnf-p: A model for efficient placement of virtualized network functions," in *Proc. IEEE CNSM*, 2014, pp. 418–423.

[14] M. C. Luizelli, L. R. Bays, L. S. Buriol, M. P. Barcellos, and L. P. Gaspary, "Piecing together the nfv provisioning puzzle: Efficient placement and chaining of virtual network functions," in *Proc. IFIP/IEEE IM*, 2015, pp. 98–106.

[15] S. D'Oro, L. Galluccio, S. Palazzo, and G. Schembra, "Exploiting congestion games to achieve distributed service chaining in nfv networks," *IEEE J. Sel. Areas Commun.*, vol. 35, no. 2, pp. 407–420, 2017.

[16] ——, "A game theoretic approach for distributed resource allocation and orchestration of softwarized networks," *IEEE J. Sel. Areas Commun.*, vol. 35, no. 3, pp. 721–735, 2017.

[17] J. Pei, P. Hong, M. Pan, J. Liu, and J. Zhou, "Optimal vnf placement via deep reinforcement learning in sdn/nfv-enabled networks," *IEEE J. Sel. Areas Commun.*, vol. 38, no. 2, pp. 263–278, 2019.

[18] Y. Sang, B. Ji, G. R. Gupta, X. Du, and L. Ye, "Provably efficient algorithms for joint placement and allocation of virtual network functions," in *Proc. IEEE INFOCOM*, 2017.

[19] Y. Xiao, Q. Zhang, F. Liu, J. Wang, M. Zhao, Z. Zhang, and J. Zhang, "Nfvdeep: adaptive online service function chain deployment with deep reinforcement learning," in *Proc. IEEE IWQoS*, 2019.

[20] B. Farkiani, B. Bakhshi, and S. A. Mirhassani, "A fast near-optimal approach for energy-aware sfc deployment," *IEEE Trans. Netw. Serv. Manag.*, vol. 16, no. 4, pp. 1360–1373, 2019.

[21] H. Hawilo, M. Jammal, and A. Shami, "Network function virtualization-aware orchestrator for service function chaining placement in the cloud," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 3, pp. 643–655, 2019.

[22] R. Gouareb, V. Friderikos, and A. Aghvami, "Virtual network functions routing and placement for edge cloud latency minimization," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 10, pp. 2346–2357, 2018.

[23] M. M. Tajiki, S. Salsano, L. Chiaraviglio, M. Shojafar, and B. Akbari, "Joint energy efficient and qos-aware path allocation and vnf placement for service function chaining," *IEEE Trans. Netw. Serv. Manag.*, vol. 16, no. 1, pp. 374–388, 2018.

[24] Z. Wang, J. Zhang, T. Huang, and Y. Liu, "Service function chain composition, placement, and assignment in data centers," *IEEE Trans. Netw. Serv. Manag.*, vol. 16, no. 4, pp. 1638–1650, 2019.

[25] O. Alhussein and W. Zhuang, "Robust online composition, routing and nf placement for nfv-enabled services," *IEEE J. Sel. Areas Commun.*, vol. 38, no. 6, pp. 1089–1101, 2020.

[26] P. Jin, X. Fei, Q. Zhang, F. Liu, and B. Li, "Latency-aware vnf chain deployment with efficient resource reuse at network edge," in *Proc. IEEE INFOCOM*, 2020, pp. 267–276.

[27] M. Feilner, *OpenVPN: Building and integrating virtual private networks*, 2006.

[28] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu, "Stateless datacenter load-balancing with beamer," in *USENIX NSDI*, 2018.

[29] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Computer networks*, vol. 31, no. 23-24, pp. 2435–2463, 1999.

[30] D. E. Culler, J. P. Singh, and A. Gupta, *Parallel computer architecture - a hardware / software approach*, 1999.

[31] P. Toth and S. Martello, *Knapsack problems: Algorithms and computer implementations*, 1990.

[32] S. Lu and X. Lu, "An exact algorithm for the weighed mutually exclusive maximum set cover problem," *arXiv preprint arXiv:1401.6385*, 2014.

[33] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *USENIX NSDI*, 2012.

[34] S. Kumar, M. Tufail, S. Majee, C. Captari, and S. Homma, "Service function chaining use cases in data centers," *draft-ietf-sfc-dc-use-cases-06*, 2017.

[35] P. Raghavan and C. D. Thompson, "Randomized rounding: a technique for provably good algorithms and algorithmic proofs," *Combinatorica*, vol. 7, no. 4, pp. 365–374, 1987.

[36] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: a scalable and flexible data center network," in *Proc. ACM SIGCOMM*, 2009.

[37] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. ACM SIGCOMM*, 2008.

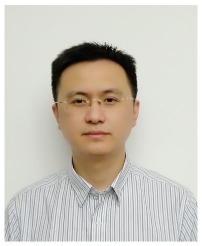
[38] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "Bcube: a high performance, server-centric network architecture for modular data centers," in *Proc. ACM SIGCOMM*, 2009.

[39] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *USENIX NSDI*, 2014.

[40] Alibaba, "Alibaba cluster trace." [Online]. Available: https://github.com/alibaba/clusterdata

Sihao Xie received his bachelor's degree in software engineering from Fudan University, China, in 2019. He is currently pursuing the master's degree in software engineering with Fudan University. His research interests include network function virtualization and programmable data plane.

Junte Ma received his bachelor's degree in software engineering from Fudan University, China, in 2019. He is currently pursuing the master's degree in computer science with Fudan University. His research interests include network function virtualization and programmable data plane.



Jin Zhao (M'10–SM'19) received the B.Eng. degree in computer communications from Nanjing University of Posts and Telecommunications, China, in 2001, and the Ph.D. degree in computer science from Nanjing University, China, in 2006. He joined Fudan University in 2006. His research interests include software defined networking and data center networks. He is a senior member of IEEE.