On the Effective Parallelization and Near-Optimal Deployment of Service Function Chains

Jianzhen Luo, Jun Li, Lei Jiao, and Jun Cai

Abstract—Network operators compose Service Function Chains (SFCs) by tying different network functions (e.g., packet inspection, flow shaping, network address translation) together and process traffic flows in the order the network functions are chained. Leveraging the technique of Network Function Virtualization (NFV), each network function can be "virtualized" and decoupled from its dedicated hardware, and therefore can be deployed flexibly for better performance at any appropriate location of the underlying network infrastructure. However, an SFC often incurs high latency as traffic goes through the virtual network functions one after another. In this paper, we first design an algorithm that leverages virtual network function dependency to convert an original SFC into a parallelized SFC (p-SFC). Then, to deploy multiple p-SFCs over the network for serving a large number of users, we model the deployment problem as an Integer Linear Program and propose a heuristic, ParaSFC, based on the Viterbi dynamic programming algorithm to estimate each p-SFC's occupation of the bottleneck resources and adjust the processing order of the p-SFCs in order to approximate the optimal solution. Finally, we conduct extensive trace-driven evaluations and exhibit that, compared to the Greedy method and the state-of-the-art CoordVNF method, ParaSFC reduces the average service latency of all the deployed p-SFCs by about 15% through parallelization while accommodating more SFC deployment requests over resource-limited networks.

Index Terms—Network function virtualization, service function chain, parallelization, deployment, quality of service

1 Introduction

While conventional network functions are often implemented as dedicated hardware middleboxes and appliances, Network Function Virtualization (NFV) has been introduced to move the functionalities of traffic flow processing into software in the form of virtual Network Functions (vNFs), thus decoupling the functionalities from the underlying hardware and enabling them to run on any general-purpose servers [1], [2]. Further, as each vNF implements a distinct functionality of processing traffic flows, such as packet inspection, traffic flow shaping, network address translation (NAT), or virtual private network (VPN), often many vNFs are chained altogether in order to process traffic flows in sequence, thus forming a Service Function Chain (SFC). SFC provides flexibility in that the vNFs in an SFC can be replaced, reordered, upgraded, and maintained separately without interfering with each other [3].

A critical concern with using SFC, however, is that if an SFC is composed of many vNFs, every flow must be processed by each vNF in order and could incur a high latency to traverse the entire SFC [4]. A high SFC latency is particularly detrimental to latency-sensitive applications. While one can try to reduce a flow's latency in traversing an SFC, such as using routing techniques to reorder vNFs in an SFC [5], it is subject to the minimum latency limit that one can achieve with a single SFC. Some research began to introduce parallel SFCs, such as that in [6], but it is essentially about making multiple copies of an SFC and dividing flow load among them, which would require more resources (every vNF needs to be copied multiple times) and does not shorten the latency with any single SFC. Some work, like ParaBox [7] and NFP [8], use order-dependency between vNFs to identify vNFs within an SFC that can run in parallel, distribute packets to vNFs in parallel, and then merge the outputs. These solutions are mainly deployed in a single server where vNFs can share memory and reduce the overhead of copying and moving data.

We therefore design a new method to convert a traditionally sequential SFC, or **s-SFC**, into a **parallelized SFC**, or **p-SFC**, where the p-SFC is deployed over different servers in a network. The p-SFC has the same number of vNFs as the original s-SFC and has a non-linear topology in the form of a directed acyclic graph where some vNFs process packets in parallel. Furthermore, as many different SFCs are often requested concurrently from different locations, it is important to consider how to deploy a mass of concurrent SFCs where each SFC is converted to a p-SFC. The problem of constructing and deploying network-deployed p-SFC is challenging, at least from the following perspectives:

(1) When converting an s-SFC into network-deployed p-SFC, we will change the positions of vNFs in the workflow of packet processing and disturb the I/O relationship between vNFs. It is critical to guarantee the processing correctness of p-SFCs, i.e., the processing result of p-SFC is the same as that of the original SFC.

(2) When deploying multiple concurrent p-SFCs, some SFCs may abusively run out of resources that are critical or the only option for other SFCs to satisfy the required QoS. Actually, the SFCs deployed first may use some scarce resources, while the SFCs deployed later may have poor

- *J. Luo and J. Cai are with the School of Cyber Security, Guangdong Polytechnic Normal University, Guangzhou 510665, China. E-mail: luojz@{gpnu.edu.cn, cs.uoregon.edu}, caijun@gpnu.edu.cn.*
- *J. Li and L. Jiao are with the Department of Computer and Information Science, University of Oregon, Eugene, OR, 97403-1202, USA. E-mail: {lijun, jiao}@cs.uoregon.edu.*

Manuscript received 25 Feb. 2020; revised 9 Sept. 2020; accepted 25 Nov. 2020. Date of publication xx xx 2020; date of current version xx xx 2020. (Corresponding author: Jun Cai.)

performance if they loss the only options. Thus, besides satisfying the required resource and QoS, the allocation of scarce resources should also be handle reasonably.

(3) SFC parallelism causes extra resource consumption. Whenever a p-SFC graph splits up at a node, duplicated packets are forwarded to two or more parallel nodes, so that more traffic will go in parallel and higher burden will be forced at the network. The question is how to bound the proportion of the extra resource consumption and make a trade-off between extra resource consumption and SFC latency reduction.

(4) The p-SFC deployment problem is NP-hard [9], [10], [11]. Due to high computational complexity, existing algorithms aiming at optimal solution are not practical to solve the p-SFC deployment problem [12], [13], [14]. Thus, several works [15], [16], [17], [18] proposed heuristic algorithms with lower complexity to obtain approximate solution. However, as lack of coordination for the behavior of resource competition between multiple SFCs, existing heuristic algorithms were not well designed to approach the optimal solution.

This paper aims to fill these missing gaps. For challenge (1), to guarantee the processing correctness of p-SFCs, we analyze the vNF dependencies from the perspective of deploying SFCs distributedly at the network level and force the resultant p-SFCs to comply with the vNF dependencies. Next, in response to challenge (2), a mechanism is needed to coordinate p-SFCs' behavior of competing critical and scarce resources. Specifically, we estimate each p-SFC's occupation of bottleneck resources and adjust the processing order of p-SFCs to approximate the optimal solution in resource-limited networks. Towards challenge (3), the extra resource consumption of p-SFC is mainly the bandwidth resource consumption. To bound this overhead, when deploying a p-SFC, we enforce the ratio of the increased bandwidth requirement due to SFC parallelism over the bandwidth requirement of the original SFC not to exceed a specified threshold that can be tuned flexibly. To handle challenge (4), we combine the resource competition coordination, the extra resource consumption control, and the dynamic programming paradigm to design a heuristic algorithm with wide applicability and high scalability to achieve near-optimal deployment of multiple concurrent p-SFCs.

In this work, we first propose several rules for parallelizing SFCs, and design a light-weight approach based on such rules to convert s-SFCs into p-SFCs. Then, we formulate an ILP-based p-SFC deployment problem while bounding the extra resource utilization and minimizing the average SFC latency. To solve the p-SFC deployment problem in acceptable execution time, we design a dynamic-programming-based approach [19] that deploys p-SFCs one after another while searching the optimal deployment path for each p-SFC. As scarce resource may be used up by the previously deployed p-SFCs, the subsequent p-SFCs may be unable to provide required QoS due to lack of sufficient resources. Thus, the processing order of the p-SFCs to be deployed will affect the overall average SFC latency. To obtain a good processing order of p-SFCs, we estimate each p-SFC's occupation of the bottleneck resources and identify the p-SFC abusiveness, i.e., the degree to which each p-SFC needs the scarce resources, and the node competitiveness, i.e., the degree to which each node is needed by the p-SFCs. To obtain an approximate optimal solution in resource-limited networks, we adjust the processing order of the p-SFCs via two policies: (1) the most abusive p-SFC first drop policy; (2) the least competitive node first allocate policy.

We implement the proposed algorithm as a tool called *ParaSFC* and conduct a series of trace-driven simulations to evaluate its performance. We also implement two comparison algorithms, i.e., the Greedy algorithm and the CoordVNF [15], [16] algorithm. The Greedy algorithm calculates the shortest path from the ingress to the egress of a p-SFC and selects nodes as close to the shortest path as possible to deploy the p-SFC. The CoordVNF algorithm exploits backtracking: it recursively tries to select valid nodes for the VNF instances step by step, and if it fails at some steps, it discards the last deployment steps and iteratively tries to select alternative nodes. We generated a range of synthetic SFC requests and run 100 simulations on three real-world network topologies: Internet2, an Indian network, and a German national research and education network. We found the following evaluation results: (1) ParaSFC can reduce the average SFC latency by about 15% through parallelization; (2) compared to Greedy and CoordVNF, ParaSFC achieves the average SFC latency closer to the optimal solution achieved by standard ILP solvers, deploying 70%$\sim$90% of the p-SFCs in their optimal paths; (3) while the execution time of the ILP solvers is much longer, i.e., more than one day, ParaSFC finishes execution within a few seconds, comparable to Greedy and CoordVNF; (4) as the link capacity increases, ParaSFC can accommodate more p-SFCs and reduce the average latency to a larger extent accordingly.

In summary, our main contribution are as follows.

- We design an algorithm based on the vNF dependency to convert s-SFCs into p-SFCs that are adaptable for network deployment. The resulted p-SFCs can guarantee the processing correctness and shorten the SFC latency.
- We formulate the problem of deploying concurrent p-SFCs in networks. The problem formulation uses the objective of minimizing the overall SFC latency and is able to control the extra resource consumption.
- We propose a heuristic algorithm to near-optimal deployment of many multiple p-SFCs of concurrent users. The proposed algorithm is based on dynamic programming and is scale well in different size of network.
- We conduct simulations to evaluate the effectiveness and efficiency of our proposed algorithm. The proposed algorithm outperforms Greedy and CoordVNF algorithms in term of SFC latency reduction, closeness to the optimal solution and acceptance rate of SFC requests.

The paper is organized as follows. The related works are studied in Section 2. The system models are presented in Section 3. The SFC parallelization algorithm is introduced in Section 4. An ILP-based formulation for p-SFC deployment problem is provided in Section 5 and an heuristic solution to the p-SFC deployment problem is proposed in Section 6. The proposed method is evaluated in Section 7. Finally, conclusion is made in Section 8.

2 Related work

In this section, we review literatures related to SFC parallelization and deployment problem. We first discuss recent efforts aiming at improving the performance of SFCs. Then, we review the early studies of SFC parallelization. Next, we present the works making contribution to SFC deployment. Lastly, we study the related problem of resource scheduling in cloud computing. Besides, we summarize the related works as shown in TABLE 1.

2.1 SFC Improvement

As the problem of chaining and deploying SFCs has been widely studied in recent years [20], [21], the problem of SFC performance improvement is drawing more and more attention from both the academic and industrial area. Several works have proposed to leverage the advantages of flexible SFC structures to promote SFC QoS such as SFC delay and throughput. Ayoubi et al. [5] relaxed the order of vNFs inside the SFCs and adjusted the relative order of vNFs in each SFC so as to avoid the packet detouring among different servers. The simulation results showed that the flexible SFCs can increase about 10% revenue per unit cost compared with the traditional SFCs with fixed order of vNFs. Mehraghdam et al. [22] and Dräxler et al. [23] represent the SFC service graphs based on the YANG model [24] and calculate several vNF composition options for the SFCs. Additionally, they design an novel algorithm for the service composition selection and achieve near-optimal placement of flexible SFCs. Usually, the throughput of the deployed SFCs is limited to that of the physical machines provisioning the SFCs, which may not be adaptable for accommodating services that demand large data rates. In order to address this problem, Ghaznavi et al. [6] proposed distributed service function chaining to place duplicated vNF instances of the same function in a distributed manner.

2.2 SFC Parallelism

Although existing works have greatly enhanced the performance of SFCs and improved the throughput of SFCs, the essence of sequential processes in the traditional SFCs may still produce risk of degrading the quality of service of some latency sensitive applications. For example, as the SFC length increases, the SFC latency grows up linearly. To address this problem, some works proposed to apply the SFC parallelism in the SFC deployment by converting the sequential SFCs with linear structure into the graph-based SFCs with graph structure, and thus enable parallel processing across NFs within the SFC. For instance, Zhang et al. [7] implemented a prototype called ParaBox on top of the DPDK-enabled Berkeley Extensible Software Switch to exploit opportunities for parallel packet processing across vNFs. Sun et al. [8] presented a policy specification scheme to describe sequential or parallel NF chaining intents and proposed a vNF orchestrator called NFP to perform lightweight packet copying for vNF parallelism. ParaBox and NFP investigated the feasibility, the effectiveness and the efficiency of SFC parallelism for service latency reduction and realized SFC parallelism at the server level, i.e., the parallel vNFs are deployed upon one single server. Nevertheless, the SFC parallelism can also be realized at the network level by deploying parallel vNFs over a network such that SFCs can exploit service diversity provided by multiple servers and maintain high stability in case of single point of failure.

The technique of SFC parallelism shares commonalities with parallel graph composition of workflows. For instance, to address the problem of parallelizing activities in scientific workflows in datacenters, Ogasawara et al. [25] proposed an algebraic approach to model the parallel execution of activities and took into account the dependency between activities within a workflow when representing the parallel computational graph. The algebraic could also be used to represent the computational graph of p-SFCs. However, the vNFs of network-level p-SFCs were deployed at multiple servers, and it is not practical to facilitate data access among different vNFs by saving intermediate results on disk or in memory, as what is done for workflows in the cloud environment. Since vNFs often read and write the processing results directly through network packets [26], the dependencies of vNFs are determined not only by the input/output data schema but also by the position in a particular SFC.

2.3 SFC Deployment

Previous works [27], [28] presented several mathematical formulations for the SFC deployment problem and leveraged optimization algorithms to solve the problem by considering different objective functions. Li et al. [29] applied the ILP model to formulate the VNF placement problem and used the objective function of minimizing the resources consumption. Zhang et al. [30] jointly optimized SFC deployment and request scheduling by minimizing the total latency of all requests. Liu et al. [10] formulated an ILP model to solve the joint optimization problem of deploying new users' SFCs and readjusting in-service SFCs while considering the trade-off between resource consumption and operational overhead. However, due to the massive consumption of computation and memory, the optimization algorithms are limited in scalability for large-scale networks.

Aiming to provide with practical solutions for time-sensitive applications, heuristic algorithms [31], [32] were designed to obtain approximate results within less time. Unlike optimization-based algorithms, heuristics run faster and often consume much less memory. Bari et al. [33] provided a dynamic-programming-based heuristic to solve the SFC deployment problem with large instances. Beck et al. [34] proposed a heuristic method called as CoordVNF to coordinate the composition of SFCs and the embedding into the networks. CoordVNF was able to quickly solve the allocation problem even in the network topologies with hundreds of nodes. Tomassilli et al. [11] designed two approximation algorithms that achieved a logarithmic approximation factor to solve the SFC deployment problem with the goal of minimizing the total setup cost. In this paper, we solve the problem of composing network-deployed p-SFCs and deploying concurrent p-SFCs in networks. We approximately provide minimal overall SFC delay for the concurrent SFCs by coordinating the resource competition between the concurrent SFCs.

The SFC deployment is also related to the topic of resource-optimal scheduling investigated in other areas,

such as resource scheduling in cloud computing [35], [36], [37]. Waibel et al. [38], [39] developed a resource scheduling technique called GeCo based on genetic algorithms to achieve fine-granular task scheduling and resource allocation optimization. Gawali et al. [40] utilized Bipartite graphs to map tasks to appropriate virtual machines and proposed divide-and-conquer methods to perform task scheduling and resource allocation. Alhubaishy et al. [41] formulated the task-scheduling problem in the cloud environment based on the adoption of the Best-Worst Method and provided a consumer-based task-oriented model for resource allocation to a prioritized set of tasks. Nevertheless, the problem of SFC deployment at the network level is beyond that of resource scheduling. The routing of packets, the locations of servers, and the diversity of resources in the underlying networks all need to be considered.

3 System Model

In this section, we introduce the network model and virtual network functions, and establish the joint problem of parallelization and deployment of SFCs.

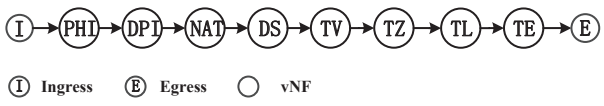
3.1 Network Setup

We represent the network topology as an undirected graph $G = (V, L)$, where V is the set of network nodes and L is the set of links connecting the network nodes. Each network node can be a router, a software-defined networking (SDN) switch, a proxy, or a general-purpose server with NFV resources such as CPU, memory, and disk. Every network node can forward every incoming packet to one or multiple outgoing ports toward the destination of the packet.

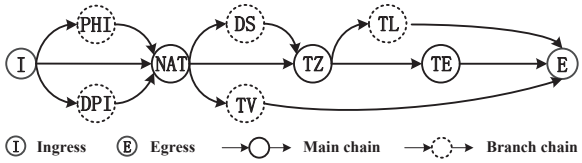
We further assume that there is an NFV orchestrator for the entire network that has a full view of the resource utilization and flow status in the whole network. (We discuss relaxing this assumption later in the paper.) Furthermore, it processes input from users regarding an SFC to be deployed in the network. It determines how to have vNFs to shape and monitor user flows, installs and configures different vNFs at selected network nodes, and steers flows toward one or more different directions to be shaped and monitored by vNFs *en route*.

3.2 vNF

Let $\mathfrak{F} = \{F_1, F_2, ..., F_K\}$ be a set of vNFs, where F_k, $\forall k = 1, 2, ..., K$ is the k-th vNF and K is the total number of vNFs. Some commonly-used vNFs are presented in TABLE 2. A vNF is either a **monitor** that monitors traffic flows without modification, or a **shaper** that processes and modifies traffic flows. As vNFs are organized in a predefined order in an SFC and process traffic flows in that order, they form a dependency relationship. If one vNF f provides its output to another vNF g, denoted as $f \prec g$ or $g \succ f$, we say that g **depends** on f, where f is the **input vNF** of g and correspondingly g is the **output vNF** of f. Each vNF also requires some amount of resources when instantiated. We denote the amount of resources required to instantiate vNF f as $r(f)$.



(a) Sequential SFC



(b) Parallelized SFC

Fig. 1: The illustration of SFC parallelization. Fig. 1a shows a sequential SFC with eight ordered vNFs. The vNFs as well as their usage purposes are defined in TABLE 2. Fig. 1b shows the corresponding parallelized SFC that is converted from the sequential SFC in Fig. 1a. It includes a main chain NAT→TZ→TE as well as five branch chains: ingress→PHI→NAT, ingress→DPI→NAT, NAT→DS→TZ, NAT→TV→egress, and TZ→TL→egress, where the first and last vNFs of the branch chains are their ingresses and egresses, respectively.

3.3 SFC Parallelization

The primary job of the NFV orchestrator of any network is to process and deploy the requested SFCs from users. The requested SFC from any user is usually a traditional SFC with all the necessary vNFs ordered in sequence, as shown in Fig. 1a. We call this SFC a **sequential SFC**, or **s-SFC** in short. In order to improve the performance, the NFV orchestrator can parallelize the sequential SFC into a directed acyclic graph with a **main chain** and multiple **branch chains** of vNFs, as shown in Fig. 1b. Every branch chain merges into the main chain either at a shaper or the egress. We call the main chain, together with its branch chains, a **parallelized SFC**, or **p-SFC** in short. Note for every vNF on the main chain of a p-SFC, it may have more than one input vNF; in this case, the vNF will not process an incoming packet until all its input vNFs have processed the packet. Moreover, assuming the user specifies that the NAT vNF is dependent on the DPI vNF and PHI vNF, the TZ vNF is dependent on the NAT vNF, and the TE vNF is dependent on the TZ vNF, Fig. 1b shows it preserves all the dependency relationships between the vNFs in Fig. 1a, as the DPI vNF and the PHI vNF go before the NAT vNF, the NAT vNF before the TZ vNF, and the TZ vNF is before the TE vNF.

4 SFC Parallelization

In this section, we present three important principles to conduct the SFC parallelism, and further design an algorithm to convert s-SFCs into p-SFCs based on the rules.

4.1 Parallelization Principles

As we convert an s-SFC into a p-SFC composed of a main chain and multiple branch chains, not only will every packet

TABLE 1: Summary of relevant literatures

Literatures	SFC acceleration	SFC parallelism	SFC placement	SFC routing	Resource usage	SFC priority	Algorithms
[6]			✓	✓			heuristic
[7], [8]	✓	✓					heuristic
[10]			✓		✓		ILP
[11]			✓				ILP and greedy algorithm
[12]			✓		✓		heuristic ILP
[13]			✓	✓			MILP
[14], [20]			✓		✓		heuristic
[15], [16]			✓		✓		ILP
[17]			✓				heuristic
[18]			✓		✓		Markov approximation
[21]	✓						ILP
[22], [23]	✓		✓	✓			heuristic
[28]				✓	✓		heuristic
[29]			✓				ILP
[30]			✓				heuristic
[31]			✓				dynamic programming
[32]				✓			congestion game
[33]			✓	✓	✓		dynamic programming
[42]			✓	✓	✓		heuristic MILP
ParaSFC	✓	✓	✓	✓	✓	✓	heuristic

TABLE 2: vNF Examples.

Category	Virtual Network Function	Dependency Rules	Purpose
Monitor	Traffic Logger (TL)	$\succ$ {prec-shapers}	Record packets and their statistics
Monitor	Traffic Visualizer (TV)	$\succ$ {prec-shapers}	Visualize the statistics of traffic flows
Monitor	Packet Header Inspector (PHI)	$\succ$ {prec-shapers}, $\prec$ {succ-shapers}	Check packet headers and decide to forward or drop any packet
Monitor	Deep Packet Inspector (DPI)	$\succ$ {prec-shapers}, $\prec$ {succ-shapers}	Check packet contents and decide to forward or drop any packet
Monitor	DDoS Scrubber (DS)	$\succ$ {prec-shapers}, $\prec$ {succ-shapers}	Identify and remove malicious traffic flows
Shaper	Network Address Translator (NAT)	$\succ$ {prec-shapers, PHI, DPI, DS}, $\prec$ {succ-shapers}	Modify IP header to map one IP address space into another
Shaper	Traffic Zipper (TZ)	$\succ$ {prec-shapers, PHI, DPI, DS}, $\prec$ {succ-shapers}, $\prec$ TU	Compress packets to shrink the traffic volume
Shaper	Traffic Unzipper (TU)	$\succ$ {prec-shapers, PHI, DPI, DS}, $\prec$ {succ-shapers}	Uncompress the compressed packets into the original format
Shaper	Traffic Encryptor (TE)	$\succ$ {prec-shapers, PHI, DPI, DS}, $\prec$ {succ-shapers}, $\prec$ TD	Convert packets into cipher text
Shaper	Traffic Decryptor (TD)	$\succ$ {prec-shapers, PHI, DPI, DS}, $\prec$ {succ-shapers}	Recover the cipher text into plain text

Note: "prec-shapers" stands for preceding shapers in original SFC, while "succ-shapers" stands for succeeding shapers in original SFC.

of a traffic flow travel along the main chain and get processed by vNFs on the main chain, a copy of the packet will also be forwarded to every branch chain to be processed by vNFs on the branch chain.

The SFC parallelization follows three principles:

(1) Parallel processing equivalency principle.

The most important principle of SFC parallelization is to guarantee the parallel processing equivalency between a p-SFC and the corresponding original s-SFC. The results of processing any traffic flow with a p-SFC should be the same as processing the same flow with the original s-SFC. To this end, we define every s-SFC as having a **modification vector** that is only composed of its shaper vNFs in the order of their appearance in the s-SFC (note that vNFs of an SFC is either a monitor or a shaper, but the monitor vNFs will *not* modify any packet in the traffic flow). When converting an s-SFC to a p-SFC, if the main chain of the p-SFC maintains the same modification vector as the s-SFC, the SFC parallelization will guarantee the parallel processing equivalency.

(2) vNF dependency preserving principle.

The dependency relationship between vNFs can be identified by (i) inspecting vNF documentation, (ii) utilizing vNFs' actions on packets derived from other studies [43], [44], [45] and analyzing extracting the vNF dependencies [7], [8] and (iii) systematically instrumenting test traffic by sending testing packets to vNF pairs and monitoring the results [46], [47].

The s-SFCs themselves, which are inputs to our algorithms, are supposed to already have dependency relationships accessible to us. Traditionally, network operators chain s-SFCs by assigning orders for vNFs, just as the workflow developers decide the ordering and the dependency of activities [8], [25]. For instance, when orchestrating an s-SFC "NAT-TZ-TE", network operators could do the following

assignments: Assign(NAT, 1), Assign(TZ, 2) and Assign(TE, 3), where "Assign(X, Y)" stands for assigning vNF X to position Y in the resulted s-SFC. Obviously, the resulted s-SFC has already contained with the desired dependency relationship between vNFs.

When converting an s-SFC to a p-SFC, the p-SFC must preserve the vNF dependency relationship implied in each original s-SFC. The third column of TABLE 2 in our manuscript shows the rules used to guarantee the preservation of such vNF dependency relationship. For instance, one of the dependency rules for the PHI vNF is "$\prec \{$succ-shapers$\}$" which denotes that PHI should stay before its succeeding shapers. Thus, as we convert an s-SFC that contains the PHI vNF as shown in Figure 1 to a p-SFC, the PHI vNF in the resulted p-SFC must be before the NAT vNF. Also, since a monitor vNF will not modify traffic flow, one monitor will not be dependent on another monitor.

(3) Shortest critical path principle.

Often there are multiple deployed paths from the ingress of a p-SFC to the egress of the p-SFC, each path is defined as a **service path** of the p-SFC, and the service path that has the largest delay is then the **critical path** of the p-SFC. The parallelization of an SFC should make the critical path of the p-SFC to be shortest. Note that the main chain may not necessarily be the critical path.

4.2 SFC Parallelization Algorithm

We design Algorithm 1 to convert an s-SFC, denoted as $s = f_1 \rightarrow f_2 \rightarrow ... \rightarrow f_n$) into a parallelized SFC (i.e., p-SFC, denoted as G_s). The algorithm takes an s-SFC and vNF dependency relationship as input and outputs a p-SFC with a main chain m and a set of branch chains B.

Algorithm 1 defines two variables to track the progress of parallelization: Q to track a set of outstanding monitor vNFs from the s-SFC and a to track the last vNF in the main chain. In the initialization phase (line 1), it initializes the main chain m to take the ingress of the s-SFC as its first element, B to be empty, Q to be empty, and a to be the ingress of the s-SFC.

Algorithm 1 then enters a loop that traverses the vNFs on the s-SFC one by one (line 4 to 21). For each current vNF f_i ($i = 1, 2, ..., n$) being processed, if it is a monitor vNF, the algorithm creates a new branch chain that starts at the last vNF in the main chain (i.e., m) and points to the current monitor vNF f_i, adds the branch chain to B, and also adds the current monitor vNF f_i to Q that tracks outstanding monitor vNFs. Otherwise, if the current vNF f_i is a shaper vNF, then the algorithm appends it to the main chain m and updates a to be f_i as well. After that, the algorithm searches Q for monitor vNFs that f_i is dependent on. For each such monitor vNF, say q, we have a branch chain currently ending at q; the algorithm then connects q to f_i, thus extending the branch chain to end at f_i. The algorithm also removes q from Q since q is no longer outstanding.

Once out of the loop, Algorithm 1 wraps up its processing by connecting the last vNF to the egress of the original s-SFC (line 19 to 22).

Algorithm 1 satisfies the principles from Section 4.1. First of all, the algorithm keeps all shaper vNFs on the main chain of the resulted p-SFC (line 10) such that the p-SFC meets the parallel processing equivalency principle. Recall that this principle is followed if the p-SFC maintains the modification vector of the input s-SFC, which is completely defined by shaper vNFs on the s-SFC. Since Algorithm 1 keeps all the shaper vNFs in the main chain of the p-SFC in the same order as in the original s-SFC, it preserves the modification vector and thus adheres to the parallel processing equivalency principle.

Also, Algorithm 1 meets the vNF dependency preserving principle that the output p-SFC maintains all the dependency relationships between the vNFs as the original s-SFC does. Specifically: (1) The relative order of shapers determine their dependency. Algorithm 1 keeps all the shaper vNFs in the main chain of the p-SFC (line 12) and maintains the relative order of the shaper vNFs the same as that in the original s-SFC. Thus, if a shaper vNF depends on another shaper vNF, the former will continue to depend on the latter in the p-SFC. (2) If a monitor vNF depends on a shaper vNF, the shaper vNF must be the last vNF in the main chain and Algorithm 1 will append the monitor vNF to the shaper vNF by creating a new branch chain that starts at the shaper vNF and connects to the monitor (line 5 to 10), such that the monitor will continue to depend on the shaper vNF in the p-SFC and Algorithm 1 preserves the dependency between them as that in the original s-SFC. (3) If a shaper vNF depends on a monitor vNF, Algorithm 1 connects the monitor vNF to the shaper to make sure the monitor vNF will continue to provide input for the shaper(line 14 to 19). (4) Note that no monitor depends on another monitor.

Finally, Algorithm 1 generates the shortest critical path for each p-SFC. When building the main chain, the algo-

Algorithm 1: SFC Parallelization Algorithm

Input: an s-SFC: $s = f_1 \rightarrow f_2 \rightarrow ... \rightarrow f_n$ and the dependency relationship of vNFs.
Output: a p-SFC composed of a main chain m and a set of branch chains $B = \{b_i\}, i = 1, 2, ...$

```
1  B ← φ, m ← ingress of s;
2  Q ← φ; // Q is a set of outstanding monitor vNFs
3  a ← ingress of s; // a is the last vNF on the main chain
4  for i = 1 : n do
5      if f_i is a monitor then
6          Create a branch chain b starting at a;
7          Append f_i to b;
8          B ← B ∪ {b};
9          Q ← Q ∪ {f_i};
10     end
11     else if f_i is a shaper then
12         Append f_i to main chain;
13         a ← f_i; // Record the last shaper of main chain
14         for any monitor q in Q do
15             if f_i depends on q then
16                 Connect q to f_i;
17                 Remove q from Q;
18             end
19         end
20     end
21 end
22 Connect a to egress of s;
```

rithm separates all of monitor vNFs that cause no modification on any flow from shaper vNFs and only keeps the shaper vNFs in the main chain, so that the resulted main chain is shortened at the most. Meanwhile, only if any two vNFs have dependency will the algorithm concatenate them according to their dependency relationship (line 12 to 17). Otherwise, the algorithm puts the two vNFs in separate new branch chains. Therefore, each branch chain is also most shortened. Since any service path between ingress and egress of p-SFC comprises the some branch chains or segments of the main chain, the resulting critical path is consequently shortest.

5 Problem Formulation of p-SFC Deployment

In this section, we firstly formulate the problem of deploying parallelized SFCs based on the ILP technique, and afterwards discuss the computational complexity of the problem.

5.1 Objective Function Formulation

We represent each p-SFC as a directed acyclic graph. Recall that there are multiple service paths within a directed acyclic graph of any p-SFC, which start from the ingress of the p-SFC and sink at the egress of the p-SFC. Each service path of any p-SFC has the same ingress and egress as that of the s-SFC, respectively. For any s-SFC, let's say s, we denote the directed acyclic graph of the corresponding p-SFC as g_s and the set of service paths of p-SFC as P_s.

In order to formulate the problem of deploying p-SFCs, we define two variables for each service path p as follows.

- $\alpha_v^p(i)$: A binary decision variable that indicates whether the i-th vNF of any service path p, denoted as f_i^p, is deployed on a specific node v. Here, the integer i varies from 1 to I_p, and I_p is the number of vNFs in service path p. The value of $\alpha_v^p(i)$ is determined by

$$\alpha_v^p(i) = \begin{cases} 1, & \text{if } f_i^p \text{ is deployed on } v; \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

- $\beta_l^p(i)$: A binary decision variable that indicates whether the segment from the i-th to (i+1)-th vNF of service path p, denoted as $p_{i:i+1}$, goes through a particular link l. The integer i ranges from 0 to I_p. The value of $\beta_l^p(i)$ is determined by

$$\beta_l^p(i) = \begin{cases} 1, & \text{If } p_{i:i+1} \text{ goes through link } l; \\ 0, & \text{Otherwise.} \end{cases} \tag{2}$$

Particularly, the variable $\beta_l^p(0)$ indicates whether the selected path from the ingress of service path p to the first vNF of service path p goes through link l. While $\beta_l^p(I_p)$ indicates whether the selected path from the last vNF to the egress of p goes through link l.

The objective of p-SFC deployment is to minimize the overall average SFC latency for all of the SFCs. We note that the SFC latency of a p-SFC is determined by the critical path that has the largest delay among all of the paths of the p-SFC. Therefore, for any p-SFC g_s, the SFC latency can be represented as

$$\max_{p \in P_s} \left(\sum_{i=1}^{I_p} \sum_{v \in V} \alpha_v^p(i) \tau_{f_i^p}^F + \sum_{i'=0}^{I_p} \sum_{l \in L} \beta_l^p(i') \tau_l^L \right), \tag{3}$$

where P_s is the set of service paths of g_s.

Accordingly, the objective function is defined as

$$\min \sum_{s \in S} \max_{p \in P_s} \left(\sum_{i=1}^{I_p} \sum_{v \in V} \alpha_v^p(i) \tau_{f_i^p}^F + \sum_{i'=0}^{I_p} \sum_{l \in L} \beta_l^p(i') \tau_l^L \right), \tag{4}$$

where $\tau_{f_i^p}^F$ is the processing latency of f_i^p and τ_l^L is the transporting latency of link l. The first term in Eq.(4) represents the summation of vNF processing latency, and the second term is the summation of transporting latency over all links of service path p.

5.2 Constraints Formulation

5.2.1 Node Constraint

For any service path p, each vNF in p should be deployed once and only once. Thus, we have

$$\sum_{v \in V} \alpha_v^p(i) = 1, \qquad \forall s \in S,\ p \in P_s,\ i \in [1, I_p]. \tag{5}$$

The total resources consumed by all vNFs deployed in any node should not be greater than the resource capacity of the node. In order to accommodate vNFs, the summation of requested resources submitted to any node cannot exceed the residual resource in the node. Thus, we have

$$\sum_{s \in S} \sum_{p \in P_s} \sum_{i=1}^{I_p} \alpha_v^p(i) r(f_i^p) \leq R(v), \forall v \in V, \tag{6}$$

where $R(v)$ is the residual resource in node v.

5.2.2 Link Constraint

In order to avoid network congestion due to lack of bandwidth resource in any links, the amount of bandwidth requirements for all SFCs whose flows go through link l should not be greater than the residual bandwidth of l, denoted as $U(l)$. That is

$$\sum_{s \in S} \sum_{p \in P_s} \sum_{i=0}^{I_p} \beta_l^p(i) w(p) \leq W(l), \quad \forall l \in L, \tag{7}$$

where $w(p)$ is the bandwidth requirement for service path p. Note that the bandwidth requirement for each service path in the p-SFC of any s-SFC s is equal to the bandwidth requirement of s.

5.2.3 Flow Constraint

We have two flow constraints as follows to guarantee that the selected links for each service path in a p-SFC can be concatenated together to form a service path in the network topology, we say routing path, starting from the ingress of the p-SFC, through all of the selected nodes, and sinking at the egress of the p-SFC.

First of all, for each routing path, the amount of traffic that arrives at a node on the routing path should be equal to the amount of traffic that leaves the node. In order to formulate the problem, we defined two variables as follows.

- $\mathbb{I}_v^p(l)$: A binary variable that indicates whether a particular link l delivers traffic of any service path p to a specific node v. The value of $\mathbb{I}_v^p(l)$ is determined by

$$\mathbb{I}_v^p(l) = \begin{cases} 1, & \text{If } l \text{ delivers } p\text{'s traffic into } v; \\ 0, & \text{Otherwise.} \end{cases}$$

- $\mathbb{O}_v^p(l)$: A binary variable that indicates whether the traffic of any service path p leaves a specific node v through a particular link l. The value of $\mathbb{O}_v^p(l)$ is determined by

$$\mathbb{O}_v^p(l) = \begin{cases} 1, & \text{If } p\text{'s traffic leaves } v \text{ through } l; \\ 0, & \text{Otherwise.} \end{cases}$$

For each service path p of a p-SFC g_s, we have the following constraint,

$$\sum_{l \in L} \sum_{i=0}^{I_p} \beta_l^p(i) \Big(\mathbb{I}_v^p(l) - \mathbb{O}_v^p(l) \Big) = 0, \quad \forall s \in S,\ p \in P_s,\ v \in V,\ i \in [1, I_p]. \tag{8}$$

Secondly, for each selected node v that deploys a vNF of any service path p, let say the i-th vNF of p, there must be at least one link that delivers traffic of p to the node v. Thus, we have

$$\sum_{l \in L} \sum_{i=1}^{I_p} \beta_l^p(i) \mathbb{I}_v^p(l) + \varphi^p(v) \geq \alpha_v^p(i), \quad \forall s \in S,\ p \in P_s,\ v \in V,\ i \in [1, I_p], \tag{9}$$

where $\varphi^p(v) = 1$ if v is the ingress of p, otherwise $\varphi^p(v) = 0$.

5.2.4 Bounding Extra Bandwidth Consumption

To bound the extra bandwidth consumption, we have to obtain the bandwidth consumption of deploying the corresponding s-SFC requests using existing algorithms such as [10], [48] and then add the following constraint for each p-SFC:

$$\sum_{p \in P_s} \sum_{i=0}^{I_p} \beta_l^p(i) w(p) \leq (1 + \kappa) \Upsilon(s), \quad \forall s \in \boldsymbol{S},\ l \in L, \tag{10}$$

where κ is the bound rate of extra bandwidth consumption and $\Upsilon(s)$ is the bandwidth requirement of s-SFC s.

5.3 Problem Complexity

The ILP-based approach has a limitation of scalability due to its high computation complexity. Similar with many traditional SFC deploying problems [48], [49] as well as some other combinatorial network optimization problems [50], the problem of deploying p-SFCs has been proved to be NP-hard [48].

6 Deployment Algorithm

Since the ILP-based solution has high complexity and unscalability in large scale problems, we propose a heuristic algorithm with low complexity and good scalability to achieve a near-optimal solution. Inspired by the observation that the SFC latency of a specific SFC is mainly determined by the performance of the critical path. The proposed method is designed to guarantee the optimal deployment for critical paths and shorten the latency of the critical path as much as possible. Thus, we divide our heuristic solution into two steps: critical path deployment and residual path deployment. All of the critical paths are deployed at the first step so that the critical paths can select preferable deployment nodes to make the SFC latency as low as possible.

6.1 p-SFC Deployment

6.1.1 Path Identification

We firstly design a recursive algorithm to identify the set of service paths within each p-SFC. Algorithm 2 presents the detail procedures of path identification. Algorithm 2 takes the service graph of a p-SFC as input, i.e., g_s, and outputs the set of service paths identified in the p-SFC, P_s.

The algorithm needs some auxiliary variables as follows.

- p: a link structure (a type of data structure) that is used to track the service path,
- v: a variable to record the current node being processed by the algorithm,
- w: a variable to denote a successor node of current processing node v,
- W: the set of successor nodes of current node v.

In the initialization procedure, Algorithm 2 sets the first member of the service path p as the ingress of the service graph g_s, sets the value of current processing node u as the ingress of service graph g_s and assigns the set of successor node of u as empty (line 7 to 9 in Algorithm 2).

The main part of Algorithm 2 is a recursive function as defined in line 10 of Algorithm 2. The recursive function, named PathFinder, takes three parameters as input, i.e., the directed acyclic graph of the p-SFC being processed (g_s), the identified path (p) and the node (u) being processed currently. The PathFinder firstly finds out all successor nodes of current node (u) in g_s, and puts them in the set W. For each successor node w in the set W, the PathFinder appends w to the tail of path p. Further, if w is not the egress of g_s, the PathFinder sets the value of node u as a successor node w and calls itself to proceed with recursion, as shown in line 14 to 17 of Algorithm 2. Otherwise, w is the egress of g_s, which implies the current path p approaches its sink node and a complete path is identified. Thus, the PathFinder copies the path p to the path set of g_s, i.e., P_s and returns to the preceding node to find other paths.

Afterwards, we select the path with largest vNF processing time within the set of the paths belonging to the service graph g_s as the potential critical path of the p-SFC. In practice, we never know which is the critical path until all paths have been deployed, however, the path with largest vNF processing time among the path set of service graph g_s is probably the critical path of the p-SFC. Thus, we sum

Algorithm 2: Path identification

Input: The service graph of a p-SFC: g_s
Output: The set of paths in the service graph g_s: P_s

Variable:
  p: a link to track any path,
  v: current node being processed.
  w: denotes a successor node of node v,
  W: the set of successor nodes of node v.
Initialization:
  $p \leftarrow$ the ingress of g_s;
  $v \leftarrow$ the ingress of g_s;
  $W \leftarrow \phi$;
Function PathFinder(g_s, p, v) **begin**
    $W \leftarrow$ get all successor nodes of v in g_s;
    foreach w *in* W **do**
      Append w to p;
      if w *is not the egress of* g_s **then**
        $v \leftarrow w$; // Set w as current node
        PathFinder(g_s, p, v); // Call the recursive function
      end
      else if w *is the egress of* g_s **then**
        Copy current path p to P_s;
        Return;
      end
    end
end

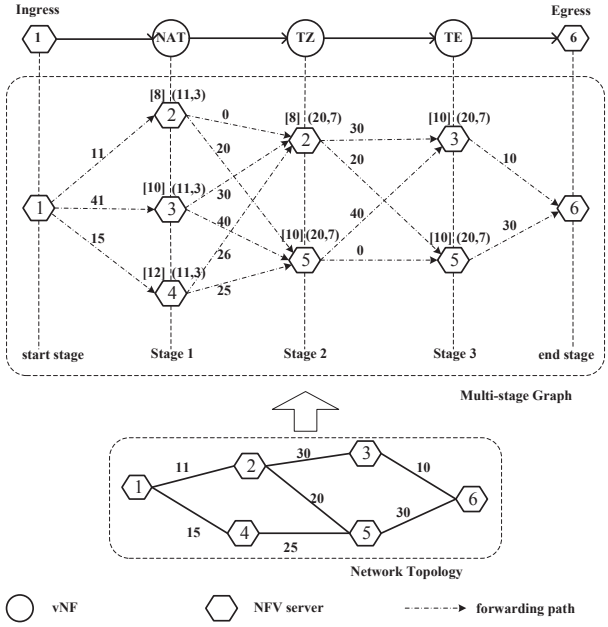


Fig. 2: Illustration of the multi-stage graph. Values over edges represent the transporting latency; values over hexagons represent the resource capacity (as in "[ ]"), the processing latency (i.e., the first value in the tuple "(x, y)"), and the resource requirement (i.e., the second value in the tuple "(x, y)").

up the vNF processing time of all vNFs for each path of a particular p-SFC, and sort the paths according to the amount of processing time. Then the path with maximal amount of processing time is the potential critical path.

6.1.2 Candidate Critical Path Deploying

We model the problem of deploying a service path in a p-SFC as a multi-stage task that deploys one vNF in each stage, and leverage the dynamic programming algorithm to deploy such vNFs iteratively. When deploying multiple service paths of a p-SFC, we first deploy the candidate critical path and then deploy all the remaining paths, so that the candidate critical path has higher priority in selecting preferable resources.

First, we construct a directional acyclic multi-stage graph [48], shown in Fig. 2, to model the multi-stage task. We denote a multi-stage graph as $\mathcal{H}$, and present the procedure to construct such a graph as follows. The ingress and the egress of the service path p are in the "start" and the "end" stage of the multi-stage graph, respectively. The nodes in any of the remaining stages of the graph $\mathcal{H}$, say, the i-th stage, are the NFV servers that have capability and capacity to deploy the i-th vNF of service path p. Specifically, we denote the "start" and the "end" stage as the 0-th and the (I_p+1)-th stage, respectively. Any node in one stage is connected to every node in the next stage via directed edges, and each edge is associated with a latency cost. For each edge, if the two endpoints of the edge are directly connected by a link, e.g., l, in the network topology, the latency cost of the edge is the transporting latency of link l; otherwise, the latency cost is the transporting latency of the shortest routing path between the two endpoints in the network topology. Note that, there is no connection between any nodes within the same stage.

In Fig. 2, we construct the multi-stage graph from the network topology. Given a service path "1 $\rightarrow$ NAT $\rightarrow$ TZ $\rightarrow$ TE $\rightarrow$ 6", where the nodes 1 and 6 are the ingress and the egress, respectively, the corresponding multi-stage graph is constructed as follows. First, the nodes 1 and 6 are designated as the "start" stage and the "end" stage, respectively. Next, suppose only the nodes 2, 3, and 4 are able to host "NAT", then these three nodes are placed in "Stage 1" of the multi-stage graph. Suppose only the nodes 2 and 5 can host "TZ", then these two nodes are placed in "Stage 2". Finally, suppose only the nodes 3 and 5 can host "TE", then they are placed in "Stage 3". Now, note that each node in one stage of the multi-stage graph is connected to every node in the next stage, and each edge is labeled by the transporting latency of the shortest path between its two endpoints, which may consist of one link or multiple links in the network topology. For instance, the edge from 3 to 2 is labeled by 30, since 3 and 2 are directly connected by a link whose transporting latency is 30 in the network topology; in contrast, the edge from Node 3 in Stage 1 to Node 5 in Stage 2 is labeled by 40, since the transporting latency of the shortest path from 3 to 5 in the network topology, i.e., "3 $\rightarrow$ 6 $\rightarrow$ 5", is 40. The transporting latency of a path is the sum of that of each link in the path.

Afterwards, a dynamic programming procedure based on the Viterbi algorithm [19] is applied to search the optimal path, i.e., Viterbi path, that produces the minimal SFC latency in the multi-stage graph. The procedure starts from

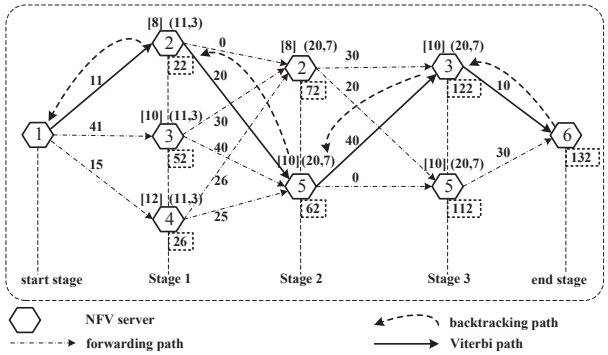


Fig. 3: Computation of the Viterbi path. Values in the dashed boxes represent the intermediate results of the minimal SFC latency from the ingress.

the first stage that follows the "start" stage, and iteratively computes the minimal latency from each node in the preceding stage to every node in the current stage, and also records the intermediate results which help compute the minimal latency in the next stage. This procedure proceeds from one stage to the next, until it eventually reaches the "end" stage. At last, a backtracking procedure is used to find out the Viterbi path that produces the minimal SFC latency.

In Fig. 3, we demonstrate how we compute the Viterbi path in our constructed multi-stage graph. First, Node 2 of Stage 1 computes the latency from the "start" stage by summing up the transporting latency from Node 1 of the "start" stage to Node 2 of Stage 1 (i.e., 11, as shown over the edge) and the processing latency of the vNF corresponding to Stage 1 (i.e., 11, as shown over Node 2). The resultant SFC latency is 22 and is recorded in the dashed box below Node 2 of Stage 1. Nodes 3 and 4 of Stage 1 repeat this process for Node 1 of the "start" stage, respectively, and the corresponding SFC latencies are 42 and 26. Next, Node 2 of Stage 2 computes the SFC latency from each node of previous stage, e.g., Node 4 of Stage 1, by summing up the recorded result in the dashed box at Node 4 (i.e., 26), the transporting latency (i.e., 26), and the processing latency (i.e., 20). The resultant SFC latency is 72. The resultant SFC latencies towards Node 2 of Stage 2 from Nodes 2 and 3 of Stage 1 are 42 and 82, respectively. Here, however, suppose if the vNF "NAT" (as in Stage 1) which requires 3 units of resource is deployed on Node 2, then the vNF "TZ" (as in Stage 2) which requires 7 units of resource cannot be deployed on Node 2 due to insufficient resource (note Node 2 has the capacity of 8); thus, Node 2 of Stage 2 will only consider the transitions from the nodes 3 and 4 of Stage 1, while Node 5 of Stage 2 can still consider the transitions from all the three nodes of Stage 1. Now, Node 2 of Stage 2 selects the minimal SFC latency of 72 from the nodes of Stage 1 and records this result in its dashed box. Similarly, every other node of Stage 2 repeats this process. We move from one stage to the next stage until we reach the "end" stage, where every node of each stage computes and records the shortest latency from nodes of the previous stage. When all nodes in the multi-stage graph finish such computation, a backtracking procedure is applied to identify the Viterbi path by finding the node in each previous stage that incurs the recorded shortest latency. As the backtracking procedure finishes, we have the Viterbi path of "$1 \rightarrow 2 \rightarrow 5 \rightarrow 3 \rightarrow 6$", where each node, except the first and the last nodes, is used to deploy the corresponding vNF(s).

Inspired by the Viterbi computation as shown in Fig. 3, we design a dynamic programming algorithm based on the Viterbi algorithm, as shown in Algorithm 3, to search the path that produces the minimal SFC latency in the multi-stage graph. Algorithm 3 uses two auxiliary variables for caching the intermediate results. One is the forwarding variable $a_i(v)$ that records the minimal latency of any paths from the ingress to a node v in the i-th stage. The other one is the backtracking variable $\delta_i(v)$ that records the preceding node of the node v in the path that produces the minimal latency $a_i(v)$. Specifically, the value of $a_0(v)$ is 0 for any v, and $a_{I_p+1}(e)$ represents the minimal latency of the paths from ingress to egress, where e is the egress of the paths. For $i \in [1, I_p + 1]$ and $j \in [1, |V|]$, the variables $a_i(v)$ and $\delta_i(v)$ are calculated by

$$\begin{cases} a_i(v_{i,j}) = \min\limits_{j'} \Big(a_{i-1}(v_{i-1,j'}) + cost(v_{i-1,j'}, v_{i,j})\Big), \\ \delta_i(v_{i,j}) = \arg\min\limits_{j'} \Big(a_{i-1}(v_{i-1,j'}) + cost(v_{i-1,j'}, v_{i,j})\Big), \end{cases} \tag{11}$$

where $v_{i,j}$ denotes the j-th node in the i-th stage and $cost(v, v')$ is the transporting latency from v to v'.

Algorithm 3 takes a critical path p of a p-SFC and the corresponding multi-stage graph $\mathcal{H}$ as inputs, and outputs the vector of nodes, i.e., d, which are selected to deploy the vNFs of the critical path. Algorithm 3 consists of three parts:

(1) In the **initialization**, it sets the value of the forwarding variable in the "start" stage as zero, and sets the value in all the rest stages as a pre-defined maximum value.

(2) Then, it performs a **forwarding procedure** to compute the minimal latency iteratively and saves the intermediate value in $a_i(v)$ and $\delta_i(v)$. The procedure proceeds from the first stage to the last stage of any multi-stage graph. In any stage i ($1 \leq i \leq I_p$), it computes the minimal latency from the ingress through any node u in stage i-1 to each node of stage i (line 20 to 27). Specifically, the algorithm calculates the transporting latency from each node (e.g., u) in the previous stage (let say stage $i-1$) to a node (e.g., v) in the current stage (say, stage i), obtains the average time for the i-th vNF of the service path p to process one packet, accesses the intermediate value $a_{(i-1)}(u)$ of the minimal latency cached at u, and sums them up according to Eq.(11) to calculate the minimal latency from the ingress to v through u (line 22). The algorithm caches the minimum of the minimal latencies in $a_i(v)$ (line 24) and caches the value of node u in the corresponding backtracking variable $\delta_i(v)$ (line 25).

(3) Further, the **backtracking procedure** infers the best deployment path with the minimal latency. This procedure starts from the last stage, i.e., the "end" stage (line 32), and searches the node u in stage I_p that yields the minimal latency from the ingress of service path p to stage "end" through node u in stage I_p (line 35 to 36). This operation proceeds from one stage to another, until it reaches the second stage. The procedure leverages a temporary variable v^* to cache the selected best node in each stage, and searches the best "preceding node" of the best node selected in

Algorithm 3: Path Deployment Algorithm

```
1  PathDeploy(p, H) begin
       Input: A service path of any p-SFC: p,
              a multi-stage graph built from p: H.
       Output: The vector of selected nodes to deploy
               the vnFs of the service path p: d
2      Variables:
3        f_i^p: the i-th vNF of service path p;
4        I_p: the amount of vNFs in service path p;
5        v, u: represents a node in graph H;
6        τ^L_(u,v): transporting delay of the link from u to
         v;
7        τ^F_{f_i^p}: average time for f_i^p to process one packet;
8        a_i(v): used to cache computing result;
9        δ_i(v): used to record backtracking information;
10       a: a variable to cache any computing result;
11       v*: a variable to cache any selected node;
12     Initialization:
13       ∀v ∈ V : a_0(v) ← 0;
14       ∀i ∈ [1, I_p + 1], v ∈ V : a_i(v) ← +inf;
15     Forwarding procedure:
16     for i ← 1 to I_p + 1 do
17       τ^F_{f_i^p} ← Get f_i^p's average process time;
18       foreach v in stage i of H do
19         if v is able to deploy f_i^p then
20           foreach u in stage i-1 of H do
21             τ^L_(u,v) ← delay from u to v;
22             a ← (τ^L_(u,v) + τ^F_{f_i^p} + a_{i-1}(u)); /* To
                 caches the latency from ingress to v */
23             if a_i(v) > a then
24               a_i(v) ← a;        /* a_i(v) caches the
                   minimum latency from ingress to v
                   */
25               δ_i(v) ← u; /* Record the value of u
                   */
26             end
27           end
28         end
29       end
30     end
31     Backtracking procedure:
32     Assign the value of v* as the egress;
33     Set v* as the first element of d;
34     for i ← I_p to 1 do
35       v* ← δ_i(v*);      /* Backtrack to search for nodes in
           the path that produces the minimum latency */
36       Append v* to d;
37     end
38     Reverse the orders of elements of d;
39 end
```

current stage (line 35). Once any node is selected as the best one, the algorithm appends it to the vector of the selected nodes, i.e., d (line 36). The algorithm reverses the order of the elements in d so that they have the same order as the vNFs of the critical path.

6.1.3 Deploying Residual Paths

We also leverage Algorithm 3 to deploy the residual paths. We construct the multi-stage graph for each of the residual paths, and apply Algorithm 3 to deploy all of such paths.

The main difference between the critical path deployment and the residual path deployment is how to construct the multi-stage graph of the paths. When constructing the multi-stage graph for one of the residual paths, in some stage (say, i) of the graph, if the i-vNF has been deployed in preceding deployment procedures, either the critical path deployment or the residual path deployment, only the node that has been selected to deploy the vNF will appear in the stage. The order to deploy the residual paths mainly depends on which p-SFC the paths belong to. For each p-SFC, Algorithm 2 has been applied to detect all the paths of the p-SFC as well as the critical path. The identified paths of the p-SFC, except the candidate critical path, is deployed in the order they appear in the detection procedure. A p-SFC will be rejected if one of its paths (critical path or residual paths) fails to be deployed. In the end, we calculate the SFC latency for all paths, both the critical path and the residual paths, and identify the path with maximal SFC latency as the real critical path, whose SFC latency is considered as the SFC latency of the p-SFC.

6.2 Deploying Multiple Concurrent p-SFCs

As we deploy multiple concurrent p-SFCs, we have to coordinate the resource competition so as to optimize the overall performance for all of the concurrent p-SFCs. To this end, we pre-deploy each p-SFC regardless of other p-SFCs to locate the required resource and identify the bottleneck of resources in the network. Then, we sort the p-SFCs according to their dependency on the bottleneck resource and process the p-SFCs according to the sorted order. If the resource is not enough to provision all of p-SFCs, we will drop some abusive p-SFCs that depends too much on the bottleneck resource.

6.2.1 Pre-Deploying p-SFCs

In the phase of pre-deployment, we aim to 1) make statistics about the resource demand on each node, and 2) re-elect the candidate critical path.

We apply the procedures introduced in Section 6.1 to deploy each p-SFC onto the network. For any p-SFC, we apply Algorithm 2 to identify the paths and candidate critical path in the p-SFC, then we leverage Algorithm 3 to deploy the candidate critical path in the network, and finally we use Algorithm 3 to deploy all of the remaining paths of the p-SFC. Note that, we mainly focus on statistics when the p-SFC is deployed, but we do not really allocate resources for any node.

Once a p-SFC has been deployed successfully, we collect the information on resource scheduling. That is, we record the nodes that are selected to deploy the vNFs of the p-SFC, and calculate the resources demanded by the p-SFC and by each node. The metrics to be calculated are as follows.

The total resource requested by p-SFC g_s at node v:

$$x_s(v) = \sum_{p \in P_s} \sum_{i=1}^{I_p} \alpha_v^p(i) r(f_i^p),\ \forall v \in V, s \in S. \tag{12}$$

The amount of resources requested at node v:

$$x_v = \sum_{s \in S} x_s(v),\ \forall v \in V. \tag{13}$$

The total amount of resource requested by all p-SFCs:

$$x = \sum_{v \in V} x_v. \tag{14}$$

Meanwhile, we re-elect the candidate of the critical path for each p-SFC. When we finish deploying a p-SFC, we calculate the latency of each path, including vNF processing time in each vNF and transporting latency in the passing links, and elect the path with maximal SFC latency as the new candidate critical path.

6.2.2 Dropping Abusive p-SFCs

We need to drop some SFCs if resource is limited. Based on the statistical information collected in the pre-deployment phase, we should check whether there are enough resources to deploy all of the SFCs. If the resources are insufficient, we should drop some of SFCs until all of the remaining SFCs can be deployed successfully.

We define the following concepts for the description of the SFC dropping procedure.

resource bottleneck: For each node, we compute the amount of resources required by all of the p-SFCs in the node and call the node a resource bottleneck if the amount of the required resources exceeds that of the residual resource in the node.

bottleneck degree: We also define the difference between the amount of resources required by all p-SFCs in any node and the residual resource of the node as the bottleneck degree of the node. Specially, when any node is not a resource bottleneck, the bottleneck degree is zero.

resource abusiveness: When any p-SFC selects some nodes to instantiate its vNFs, we define the summation of the bottleneck degrees over all of the selected nodes as the resource abusiveness of the p-SFC.

When the resource is insufficient to provision all SFCs, we drop the SFCs with the highest resource abusiveness. The resource abusiveness indicates the contribution of a p-SFC causing the bottleneck. Generally, the higher the resource abusiveness of the p-SFC, the more vNFs of the p-SFC are deployed in the resource bottleneck. In other words, the p-SFC depends on the resource bottlenecks more than others, and is referred to as more abusive. As a result, we select the p-SFCs whose resource abusiveness are larger and drop them preceding others so as to accept as many p-SFCs as possible. We call this dropping policy as **most abusive p-SFC first drop policy**.

6.2.3 Deploying Concurrent p-SFCs

We are prone to assign resources in nodes which are not resource bottlenecks or whose bottleneck degree is small if the performance of p-SFC would not degrade. We call this resource assignment policy as **least competitive node first allocate policy**.

In order to follow the deployment policy and resource assignment policy, we adjust the processing order of p-SFCs according to the resource abusiveness of p-SFCs. Specifically, we sort the p-SFCs in ascending order according to the resource abusiveness and deploy the p-SFCs according to the sorted order. Thus, since the less abusive p-SFCs have smaller probability to cause resource bottleneck, we deploy them preceding the abusive p-SFCs.

Meanwhile, we modify the forwarding procedure in Algorithm 3, so that we can try best to first allocate resources in the least competitive node. The modified forwarding procedure is shown in Algorithm 4. When we deploy any SFC, there is usually more than one candidate node which can provide the SFC with the best performance. However, the bottleneck degrees of the candidate nodes might be different. As we want to follow the resource assignment policy and allocate the least competitive node first, the forwarding procedure in Algorithm 3 is modified to select the node that is less competitive, i.e., select the node with small bottleneck degree, as shown in line 16 to 20 of Algrithm 4. The modified algorithm takes one more parameter, i.e., $\chi(v)$, the bottleneck degree of every node $v \in V$, as input, and defines one more variable w to cache any candidate node in the forwarding procedure (as shown in line 2, 14 and 19). When we identify a node that provides with the same performance as the candidate node, we check whether the node is less competitive than the candidate node (line 16). Afterward, we replace the candidate node with the node (line 17 to 19).

According to the deployment policy and resoruce assignment policy, we design procedures as shown in Algorithm 5 to deploy multiple concurrent p-SFCs to achieve minimal overall SFC latency. We first pre-deploy every p-SFC and collect statistical information about the resource assignment, such as the resource requirement in each node and the amount of resources required by the set of p-SFCs (Step 1). Based on the statistical information, we identify the resource bottlenecks whose resource is over-allocated and calculate the abusiveness degree for each p-SFC (Step 2). In Step 3, we sort the p-SFCs in ascending order according to the abusiveness degree of the p-SFCs. In the Step 4, we deploy the p-SFCs one after another according to the sorted order in Step 3, which implies that the p-SFC with the smallest abusiveness degree will be deployed first. Finally, when the resources of the network are insufficient, we stop deploying more p-SFCs. In other words, the residual p-SFCs which are most abusive are dropped.

6.3 Computational Complexity

Let N be the total number of NFV server nodes, N^f the number of NFV server nodes hosting resource for deploying vNF f, and L is the maximum length of SFC. Apparently, we have $N \geq N^f$. Since we aim to gain an upper bound

Algorithm 4: Modified Forwarding Procedure

Input: p: a service path of any p-SFC,
$\mathcal{H}$: a directed multi-stage graph built from p,
$\chi(v)$: resource abusiveness of node $v \in V$.

Variables:
  w: a variable to cache any node in forward procedure;
  Other variables are the same as the Algorithm 3.
for $i \leftarrow 1$ *to* $I_p + 1$ **do**
  $\tau^F_{f^p_i} \leftarrow$ Get f^p_i's average process time;
  foreach v *in stage* i *of* $\mathcal{H}$ **do**
    if v *is able to deploy* f^p_i **then**
      foreach u *in stage* i-1 *of* $\mathcal{H}$ **do**
        $\tau^L_{(u,v)} \leftarrow$ delay from u to v;
        $a \leftarrow \left(\tau^L_{(u,v)} + \tau^F_{f^p_i} + a_{i-1}(u)\right)$; /* To cache the latency from ingress, through u, to v */
        if $a_i(v) > a$ **then**
          $a_i(v) \leftarrow a$; /* $a_i(v)$ caches the minimum latency from ingress, through u, to v */
          $\delta_i(v) \leftarrow u$; /* Record the value of u */
          $w \leftarrow u$;
        else
          /* If u is a less competitive node. */
          if $a_i(v) == a$ *and* $\chi(u) < \chi(w)$ **then**
            $a_i(v) \leftarrow a$;
            $\delta_i(v) \leftarrow u$;
            $w \leftarrow u$;
          end
        end
      end
    end
  end
end

Algorithm 5: Concurrent p-SFCs Deployment Procedures

Step 1	Pre-deploy the set of p-SFCs and collect statistic information about the resource assignment;
Step 2	Identify the resource bottleneck and estimate the abusiveness of each p-SFC;
Step 3	Sort the p-SFCs in ascending order according to the abusiveness of p-SFCs;
Step 4	Deploy the set of p-SFCs one by one according to the sorted order in Step 3.
Step 5	If the residual resource in the network is insufficient to provision more p-SFCs, we drop the residual p-SFCs (i.e., the most abusive p-SFCs);

of our algorithm, it still makes sense to use N instead of N^f in computational complexity analysis according to the property of inequality and the definition of upper bound.

In Algorithm 1, the complexity of traversing each vNF in any SFC in line 4 is $O(L)$. The procedure of checking any monitor q in Q in line 14 also yields a complexity of $O(L)$. As a result, the total computational complexity for Algorithm 1 to process all SFCs is $O(NL^2)$.

Since Algorithm 2 traverses all paths within any p-SFC, the computational complexity is equal to the amount of paths of the p-SFC. Suppose the maximum length of branch chains is L that often is a small number, and the total paths of the p-SFC is usually not large. Thus, the computational complexity of Algorithm 2 is regarded as $O(1)$.

The forwarding procedure in Algorithm 3 performs $O(I_p|\mathbf{S}|N^2L)$ operations to process all SFCs. Meanwhile, the procedure of backtracking takes I_p iterations to identify the best Viterbi path and a few operations in each iteration. Since the amount of paths within each p-SFC is small, we treat I_p as a constant. Therefore, the algorithm performs $O\left(C|\mathbf{S}|N^2L\right)$ operations to deploy all SFC requests.

7 Evaluation

In this section, we present numerical results to evaluate the effectiveness of SFC parallelism and the performance of SFC deployment algorithm. We implement the proposed p-SFC deployment algorithm, i.e., ParaSFC, and the comparison algorithms, including: (1) a greedy method denoted as "Greedy", which calculates the shortest path from the ingress to the egress of each p-SFC and selects nodes as close to the shortest path as possible for deploying the p-SFC; (2) a state-of-the-art coordinated method denoted as "CoordVNF" [15], [16], which uses a backtracking method to recursively deploy vNFs upon valid nodes; and (3) the ILP-based optimization approach denoted as "Optimal", which solves the deployment problem to its optimum by invoking standard ILP solvers. We implemented all of such algorithms using C/C++. We made all our data and source codes publicly available at our Github webpage (https://github.com/SmileRob/paraSFC).

7.1 Simulation Setup

7.1.1 Network Topology and Flow Traces

We consider three real world network topologies from SNDlib [51], including Abilene [52], an Indian network (India35) and a German National Research and Education Network (Germany50). The amounts of nodes and links contained in the topologies are shown in TABLE 3. We randomly select a set of nodes as vNF server for each network topology as shown in the fourth column in TABLE 3.

During the simulation, we generate a set of traffic requests according to the ranges as shown in the fifth column of TABLE 3 and designate random source and destination nodes for each flow. The range of traffic requests is approximate ten to thirty times of the node number in network.

7.1.2 vNFs and SFCs

We consider 10 types of commonly-used vNFs as shown in TABLE 2. We set the values of resource requirements and

TABLE 3: Parameter setting for network and traffic.

Network	nodes	links	# vNF servers
Internet2	12	15	7
India35	35	79	18
Germany50	50	88	20

TABLE 4: Service Function Chains and Requirements.

Service	Service Function Chain	Data Rate
Web Service	NAT-DS-TL-TV	100kbit/s
VoIP	NAT-TE-PHI-TL-TD-NAT	64kbit/s
Video Streaming	TL-TV-TZ-TU-PHI-DPI-NAT	4Mbit/s
Online Gaming	NAT-PHI-DS-NAT	50kbit/s

processing time for each vNF following existing work [6], [18], [33], [53], [54]. According to the complexity of vNFs, the required resources of each vNF range from 4 to 10 units [10], [18], [33]. For a particular vNF, the required resources and processing time are set as fixed values [9], [10], [55]. Since the required resources and processing time of each vNF can vary across servers in practice, we can easily change our algorithm to adapt to any parameter settings. We consider four types of services, i.e., web browsing, VoIP, video streaming, and online gaming, as in TABLE 4, and generate an SFC for each service. The data rate in TABLE 4 represents the required bandwidth for an SFC. More detail information about the parameter settings can be founded in our implementation project in Github (https://github.com/SmileRob/paraSFC).

7.2 Results

7.2.1 *Benefit of SFC parallelization*

To verify the effectiveness of SFC parallelization, we conducted two groups of simulations: sequential SFC deployment and parallelized SFC deployment. The number of the sequential SFC requests for each network topology was $0.5 \times |V|^2$, with $|V|$ as the node number of the network, as shown in TABLE 3. In the first group, we deployed sequential SFCs without parallelization. In the second group, we converted the sequential SFCs considered in the first group into parallelized SFCs using Algorithm 1, and then deployed the converted p-SFCs.

We performed 100 independent simulations for each group of simulations and obtained the average results. Fig.4 shows the benefit of decreasing the SFC latency in each network topology. The average SFC latency of the parallelized SFCs is often smaller than that of the sequential SFCs. Due to parallelization, the average latency of SFCs deployed by ILP-based approach can be reduced by 12% to 15% and the average latency of SFCs deployed by ParaSFC can be reduced by 15%. Apparently, the SFC parallelism can benefit users from decreasing the SFC latency.

The benefit of SFC parallelism varies in different network scales as shown in Fig. 5. In Internet2 that is a small scale network, about 50% of the SFCs can reduce at least 15% of the latency. In India35, about 40% of the SFCs can reduce at least 15% of the latency. While in Germany50 that is much larger in network scale, only 27% of the SFCs can reduce at least 15% of the latency. The reason of this observation is

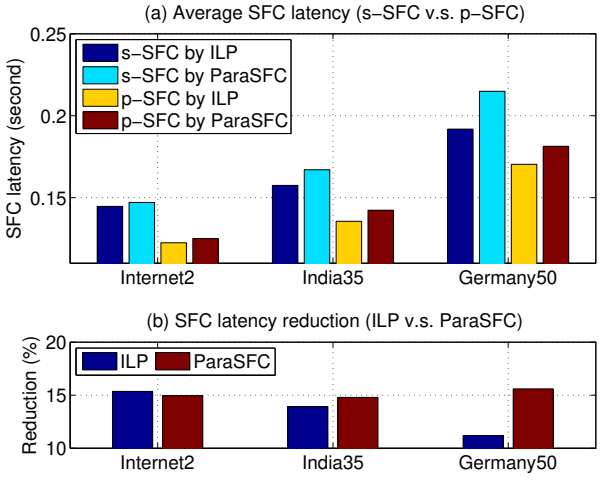


Fig. 4: average SFC latency

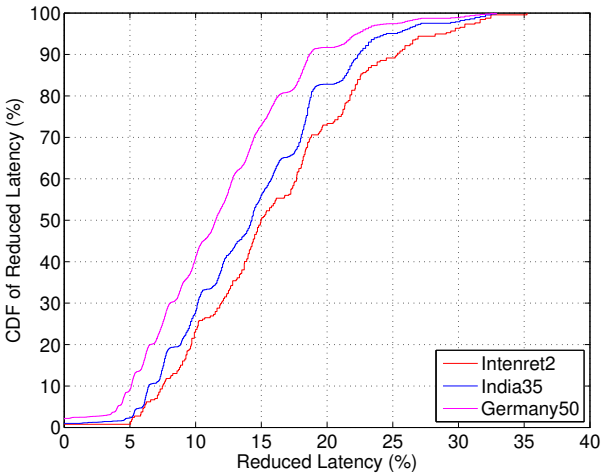


Fig. 5: Reduced SFC latency CDF

that the SFC parallelism mainly reduces the vNF processing time. In a large scale of network, the average transporting latency between any two nodes is larger than that in a small scale of network. Thus, the fraction of the reduction of vNF processing time to the total latency of an SFC in a large scale network is smaller than that in a small scale network. As a result, the benefit of SFC parallelism is more significant in the small network than that in the large network.

To explicitly consider the increased traffic (or the extra bandwidth consumption) caused by parallelization, we added a new constraint regarding the "link capacity" to our ILP problem formulation in Section 5.2.4. When deploying the p-SFCs to minimize the latency, we ensured that the total bandwidth consumption respects the link capacity. We presented the average SFC latency reduction of p-SFCs deployed by ParaSFC and the relation between the average SFC latency reduction versus link capacity increment in Fig. 6. As can be seen, the reduction of average service latency of all the deployed p-SFCs is 14.5%∼15.7% through parallelization. As the link capacity (which is represented in terms of the increment compared to our original link capacity) increases, more p-SFCs can be hosted and accordingly the average latency reduction grows.

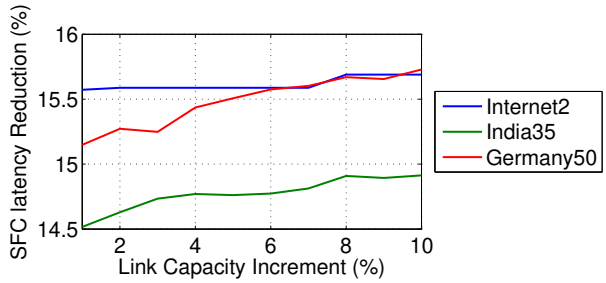


Fig. 6: SFC latency reduction v.s. link capacity increment

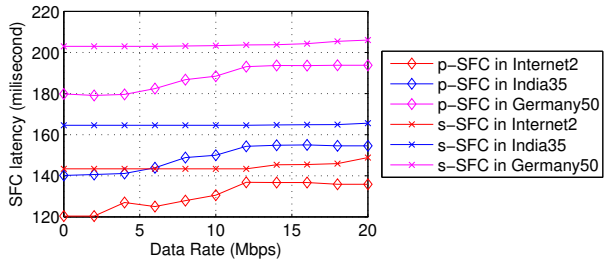


Fig. 7: SFC latency v.s. data rates

Fig. 7 shows how the data rates can affect the SFC latency. If the link capacity of the underlying network is largely sufficient, then data rates will not have any obvious effect on the SFC average latency. In contrast, if the link capacity is not large, i.e., just enough to host all s-SFCs, (as in Fig. 7), then, as the data rates increase, the SFCs that need to be deployed later may not be able to be deployed on shorter paths, because such links have already been occupied by SFCs that are deployed earlier and thus have no residual capacity. In this case, the SFC average latency will increase, since some SFCs have to choose paths of longer delay.

7.2.2 *Comparison with Related Algorithms*

In order to evaluate the performance of our algorithm, we compare our results with benchmark and the comparison algorithms, i.e., CoordVNF and Greedy.

Firstly, we present the average SFC latency for different network topologies in Fig.8. The average SFC latency produced by ParaSFC is often closest to the optimal solution among the three algorithms, which implies that the proposed solution outperforms the other two comparison algorithms in term of average SFC latency. As can be seen, ParaSFC is close to Greedy for small-scale evaluations; yet, for large-scale evaluations, ParaSFC differs from Greedy more apparently. While the Greedy algorithm splits the problem of deploying an SFC into multiple subproblems and selects the best solution for each subproblem, ParaSFC searches for the optimal solution for deploying SFCs via the Viterbi dynamic programming algorithm, estimates each SFC's occupation of bottleneck resources, and then adjusts the processing order of the SFCs in order to approximate the optimal solution. For small-scale evaluations, it is thus true that the results of both approaches could be close to optimum, because the solution space is small. However, for large-scale evaluations, the advantage of ParaSFC becomes more apparent. In this new figure, we run 100 simulations using inputs of larger sizes, i.e., the number of SFCs in each simulation is now $0.5 * |V|^2$ versus a random selected size ranging from 0 to $30 * |V|$ previously, where $|V|$ is the number of nodes of the underlying network. The results show that the average SFC latency of ParaSFC is much better than that of Greedy.

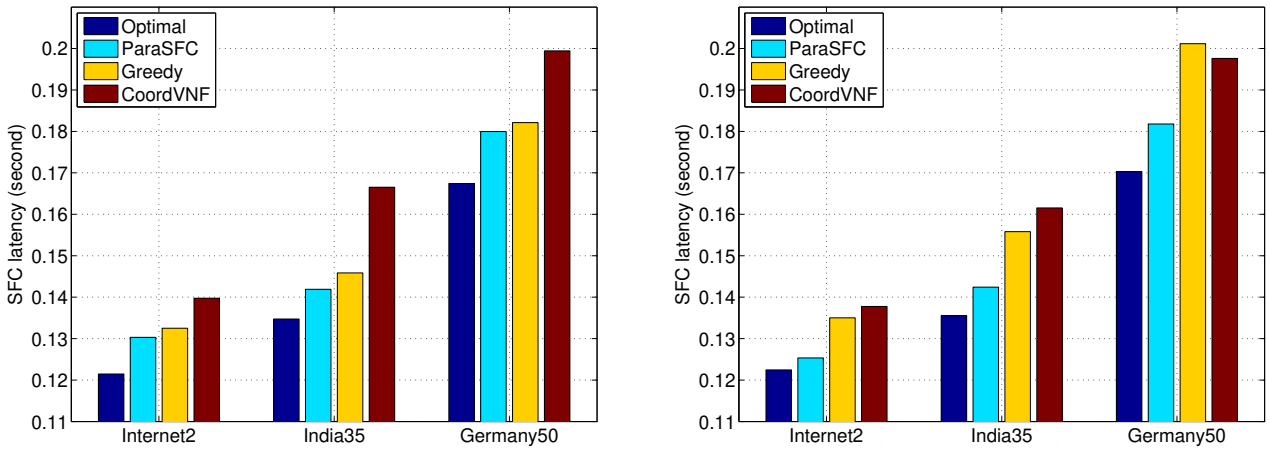
Secondly, we present Fig.9 to show the closeness between the results produced by each method and the results of the optimal solution. In Fig.9, the X-axis is the closeness that is defined as the difference between the SFC latency of each deployed p-SFC and that of the corresponding optimal result, and the Y-axis is the cumulative probability of each SFC latency difference, e.g., the cumulative distribution function (CDF) of closeness. The closeness of most p-SFCs are zero, which means that most of the p-SFCs are deployed in the optimal shortest path between the ingress and egress of each SFC. As shown in Fig.9, by using the proposed method, almost 90% of p-SFCs in Internet2 are deployed in optimal path, more than 70% of p-SFCs in both India35 and Germany50 are also deployed in their optimal paths. The results also show that the closeness of ParaSFC is often better than other methods. It implies that the proposed ParaSFC method is more probable to deploy the p-SFCs in optimal path than the comparison methods.

Next, we present the acceptance rate of p-SFCs when the virtualization resource is limited, as shown in Fig.10. We set the amount of CPU cores within each network topology as 2000 cores, and increase the amount of SFC requests from 1 to 200. We found that each algorithm produced 100% acceptance of p-SFCs at the beginning since the resources in the network is enough to deploy all requests. But the acceptance drops gradually as the amount of SFC requests increases. However, the acceptance rate produced by ParaSFC is never less than the other two algorithms. The reason is that ParaSFC adjusts the processing order of SFCs and selects a better processing order. Meanwhile, the ParaSFC also drops some SFCs that causes resource bottlenecks in the network.

At last, we provide comparison results about the algorithm execution time in Fig. 11. Although ILP outputs the optimal solution, its execution time is much longer than those of the other algorithms. Note the exponential growth in the vertical axis of the figure. Thus, ILP is not suitable for practical use due to bad scalability. The algorithm execution time of ParaSFC, Greedy, and CoordVNF are close. Despite ParaSFC is a little slower, it vastly outperforms Greedy and CoordVNF in terms of the SFC latency, much close to the optimal solution of ILP, with higher acceptance rates.

8 Conclusion

SFC is a key technique to the success of the NFV paradigm. Composed of a sequence of vNF, an SFC, however, is subject to a high latency in processing traffic flows. In this paper, we studied how to reduce the latency through parallelism, and further introduced a near-optimal heuristic approach to deploying many concurrent parallelized SFCs over networks, which has a polynomial computational complexity. Our simulation shows that the SFC parallelization can reduce the average SFC latency by 12∼15%, and compared to greed-based algorithm and CoordVNF approach, our SFC



(a) Smaller number of SFCs

(b) Larger number of SFCs

Fig. 8: Average SFC latency.

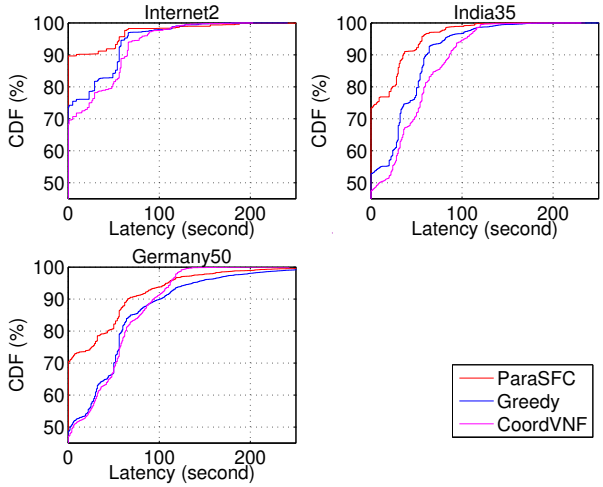


Fig. 9: Closeness to the optimal solution

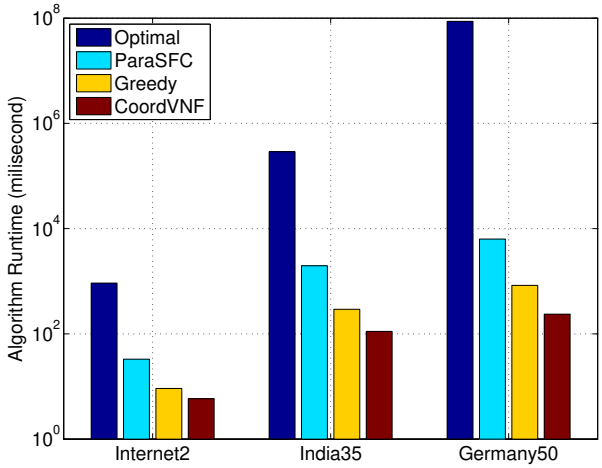


Fig. 11: Algorithm Runtime (milisecond)

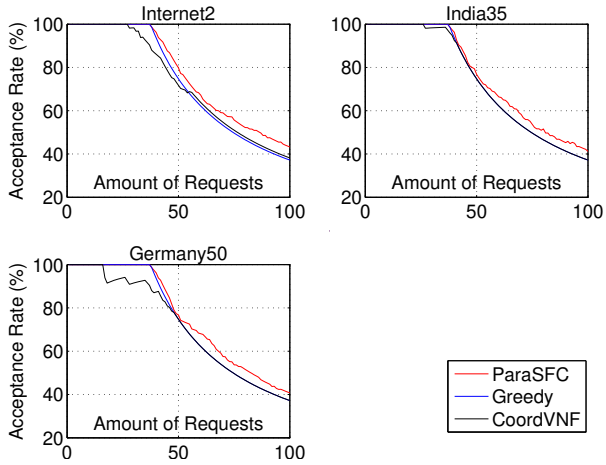


Fig. 10: Acceptance rate. The amount of CPU cores within each network topology is 2000 cores.

deployment approach achieves a higher rate of deployment success and a lower SFC latency.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments. The work of J. Luo and J. Cai was supported in part by the National Key Research and Development Program of China (SQ2019YFB180098), the National Natural Science Foundation of China (No. 61972104, No.61902080, No.61702120), The Key Areas of Guangdong Province (No. 2019B010118001), The science and technology project in Guangzhou (No. 201803010081), The National key R & D plan (No. SQ2019YFB180098, No. 2018YFB1802200), Foshan Science and Technology Innovation Project, China (2018IT100283), Science and Technology Program of Guangzhou, China (202002020035). L. Jiao was supported in part by the Ripple Faculty Fellowship.

References

[1] R. Mijumbi, J. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys and Tutorials*, vol. 18, pp. 236–262, 09 2015.

[2] K. Joshi and T. Benson, "Network function virtualization," *IEEE Internet Computing*, vol. 20, no. 6, pp. 7–9, 11 2016.

[3] P. Quinn and T. D. Nadeau, "Problem statement for service function chaining," Internet Requests for Comments, RFC Editor, RFC 7498, April 2015. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7498.html

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

[5] S. Ayoubi, S. R. Chowdhury, and R. Boutaba, "Breaking service function chains with khaleesi," in *IFIP Networking Conference*. IFIP, 2018.

[6] M. Ghaznavi, N. Shahriar, S. Kamali, R. Ahmed, and R. Boutaba, "Distributed service function chaining," *IEEE Journal on Selected Areas in Communications*, vol. 35, no. 11, pp. 2479–2489, 2017.

[7] Y. Zhang, B. Anwer, V. Gopalakrishnan, B. Han, J. Reich, A. Shaikh, and Z.-L. Zhang, "Parabox: Exploiting parallelism for virtual network functions in service chaining," in *Proceedings of the Symposium on SDN Research*. ACM, 2017, pp. 143–149.

[8] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu, "Nfp: Enabling network function parallelism in NFV," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 43–56.

[9] L. Wang, Z. Lu, X. Wen, R. Knopp, and R. Gupta, "Joint optimization of service function chaining and resource allocation in network function virtualization," *IEEE Access*, vol. 4, pp. 8084–8094, 2016.

[10] J. Liu, W. Lu, F. Zhou, P. Lu, and Z. Zhu, "On dynamic service function chain deployment and readjustment," *IEEE Transactions on Network and Service Management*, vol. PP, no. 3, pp. 543–553, 2017.

[11] A. Tomassilli, F. Giroire, N. Huin, and S. Pérennes, "Provably efficient algorithms for placement of service function chains with ordering constraints," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, 2018, pp. 774–782.

[12] 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 *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2015, pp. 98–106.

[13] B. Addis, D. Belabed, M. Bouet, and S. Secci, "Virtual network functions placement and routing optimization," in *2015 IEEE 4th International Conference on Cloud Networking (CloudNet)*, 2015, pp. 171–177.

[14] R. Riggio, A. Bradai, D. Harutyunyan, T. Rasheed, and T. Ahmed, "Scheduling wireless virtual networks functions," *IEEE Transactions on Network and Service Management*, vol. 13, no. 2, pp. 240–252, 2016.

[15] M. T. Beck and J. F. Botero, "Coordinated allocation of service function chains," in *2015 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2015, pp. 1–6.

[16] ——, "Scalable and coordinated allocation of service function chains," *Computer Communications*, vol. 102, pp. 78–88, 2017.

[17] S. Khebbache, M. Hadji, and D. Zeghlache, "Scalable and cost-efficient algorithms for VNF chaining and placement problem," in *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*, 2017, pp. 92–99.

[18] C. Pham, N. H. Tran, S. Ren, W. Saad, and C. S. Hong, "Traffic-aware and energy-efficient VNF placement for service chaining: Joint sampling and matching approach," *IEEE Transactions on Services Computing*, to be published.

[19] G. D. Forney, "The viterbi algorithm," *Proceedings of the IEEE*, vol. 61, no. 3, pp. 268–278, 1973.

[20] R. Riggio, A. Bradai, T. Rasheed, J. Schulz-Zander, S. Kuklinski, and T. Ahmed, "Virtual network functions orchestration in wireless networks," in *2015 11th International Conference on Network and Service Management (CNSM)*, 2015, pp. 108–116.

[21] B. Martini, F. Paganelli, P. Cappanera, S. Turchi, and P. Castoldi, "Latency-aware composition of virtual functions in 5g," in *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*, 2015, pp. 1–6.

[22] S. Mehraghdam and H. Karl, "Placement of services with flexible structures specified by a yang data model," in *NetSoft Conference and Workshops (NetSoft), 2016 IEEE*. IEEE, 2016, pp. 184–192.

[23] S. Dräxler and H. Karl, "Specification, composition, and placement of network services with flexible structures: Specification, composition, and placement of flexible services," *International Journal of Network Management*, vol. 27, no. 2, p. e1963, 2017.

[24] M. Bjorklund, "Yang - a data modeling language for the network configuration protocol (netconf)," Internet Requests for Comments, RFC Editor, RFC 6020, October 2010. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6020.txt

[25] E. S. Ogasawara, D. D. Oliveira, P. Valduriez, J. Dias, and M. Mattoso, "An algebraic approach for data-centric scientific workflows," *Proceedings of the Vldb Endowment*, vol. 4, no. 12, pp. 1328–1339, 2011.

[26] J. Halpern and C. Pignataro, "Service function chaining (sfc) architecture," Internet Requests for Comments, RFC Editor, RFC 7665, October 2015. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7665.html

[27] S. Mehraghdam, M. Keller, and H. Karl, "Specifying and placing chains of virtual network functions," in *IEEE International Conference on Cloud Networking*, 2014.

[28] T. W. Kuo, B. H. Liou, C. J. Lin, and M. J. Tsai, "Deploying chains of virtual network functions: On the relation between link and server usage," in *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, 2016.

[29] X. Li and C. Qian, "The virtual network function placement problem," in *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2015, pp. 69–70.

[30] Q. Zhang, Y. Xiao, F. Liu, J. C. S. Lui, and T. Wang, "Joint optimization of chain placement and request scheduling for network function virtualization," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017.

[31] R. Cohen, L. Lewin-Eytan, J. S. Naor, and D. Raz, "Near optimal placement of virtual network functions," in *2015 IEEE Conference on Computer Communications (INFOCOM)*, 2015, pp. 1346–1354.

[32] S. D'Oro, L. Galluccio, S. Palazzo, and G. Schembra, "Exploiting congestion games to achieve distributed service chaining in NFV networks," *IEEE Journal on Selected Areas in Communications*, vol. 35, no. 2, pp. 407–420, 2017.

[33] M. F. Bari, S. R. Chowdhury, R. Ahmed, R. Boutaba, and O. C. M. B. Duarte, "Orchestrating virtualized network functions," *IEEE Transactions on Network and Service Management*, vol. 13, no. 4, pp. 725–739, 2016.

[34] M. T. Beck and J. F. Botero, "Scalable and coordinated allocation of service function chains," *Comput. Commun.*, vol. 102, pp. 78–88, 2017.

[35] S. S. Gill and I. Chana, "A survey on resource scheduling in cloud computing: Issues and challenges," *Journal of Grid Computing*, vol. 14, 02 2016.

[36] L. Jiao, A. M. Tulino, J. Llorca, Y. Jin, and A. Sala, "Smoothed online resource allocation in multi-tier distributed cloud networks," *IEEE/ACM Transactions on Networking*, vol. 25, no. 4, pp. 2556–2570, 2017.

[37] S. Li, Y. Zhou, L. Jiao, X. Yan, X. Wang, and M. R. Lyu, "Delay-aware cost optimization for dynamic resource provisioning in hybrid clouds," in *IEEE International Conference on Web Services*, 2014.

[38] P. Waibel, A. Yeshchenko, S. Schulte, and J. Mendling, "Optimized container-based process execution in the cloud," in *On the Move to Meaningful Internet Systems. OTM 2018 Conferences*, H. Panetto, C. Debruyne, H. A. Proper, C. A. Ardagna, D. Roman, and R. Meersman, Eds. Cham: Springer International Publishing, 2018, pp. 3–21.

[39] P. Waibel, C. Hochreiner, S. Schulte, A. Koschmider, and J. Mendling, "Viepep-c: A container-based elastic process platform," *IEEE Transactions on Cloud Computing*, in press, 2019.

[40] M. B. Gawali and S. K. Shinde, "Task scheduling and resource allocation in cloud computing using a heuristic approach," *Journal of Cloud Computing*, vol. 7, no. 1, p. 4, 2018.

[41] A. Alhubaishy and A. Aljuhani, "The best-worst method for resource allocation and task scheduling in cloud computing," in *2020 3rd International Conference on Computer Applications Information Security (ICCAIS)*, 03 2020, pp. 1–6.

[42] M. Zeng, W. Fang, and Z. Zhu, "Orchestrating tree-type vnf forwarding graphs in inter-dc elastic optical networks," *Journal of Lightwave Technology*, vol. 34, no. 14, pp. 3330–3341, 2016.

[43] A. Bremler-Barr, Y. Harchol, D. Hay, and Y. Koral, "Deep packet inspection as a service," in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 271–282.

[44] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 323–336.

[45] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: Network processing as a cloud service," in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 13???24.

[46] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. Greenberg, and Y.-M. Wang, "WebProphet: Automating performance prediction for web services," in *7th USENIX Symposium on Networked Systems Design and Implementation (NSDI 10)*. San Jose, CA: USENIX Association, Apr. 2010.

[47] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall, "Demystifying page load performance with WProf," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, Apr. 2013, pp. 473–485.

[48] F. Bari, S. R. Chowdhury, R. Ahmed, R. Boutaba, and O. C. M. B. Duarte, "Orchestrating virtualized network functions," *IEEE Transactions on Network and Service Management*, vol. 13, no. 4, pp. 725–739, 2016.

[49] N. Huin, B. Jaumard, and F. Giroire, "Optimal network service chain provisioning," *IEEE/ACM Transactions on Networking*, vol. 26, no. 3, pp. 1320–1333, June 2018.

[50] M. Chen, S. C. Liew, Z. Shao, and C. Kai, "Markov approximation for combinatorial network optimization," *IEEE transactions on information theory*, vol. 59, no. 10, pp. 6301–6327, 2013.

[51] S. Orlowski, R. Wessäly, M. Pióro, and A. Tomaszewski, "SNDlib 1.0–Survivable Network Design Library," *Networks*, vol. 55, no. 3, pp. 276–286, April 2010.

[52] Y. Zhang, "Abilene dataset," 2004. [Online]. Available: http://www.cs.utexas.edu/~yzhang/research/AbileneTM

[53] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *ACM SIGCOMM Computer Communication Review*, vol. 38, pp. 63–74, 10 2008.

[54] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2014, pp. 459–473.

[55] D. Harutyunyan, N. Shahriar, R. Boutaba, and R. Riggio, "Latency-aware service function chain placement in 5g mobile networks," in *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2019, pp. 133–141.

Jianzhen Luo received his B.S. and Ph.D. Degree from Sun Yat-Sen University in 2009 and 2015, respectively. He is currently an Associate Professor with School of Cyber Security, Guangdong Polytechnic Normal University, China. Since 2018, he is a Visiting Scholar at University of Oregon, USA. His research interests include virtual network functions chaining and deployment, future network performance improvement, network behavior modeling and network vulnerability analysis.

Jun Li received the B.S. degree from Peking University in 1992, the M.E. degree from the Chinese Academy of Sciences in 1995 (with a Presidential Scholarship), and the Ph.D. degree (with Outstanding Doctor of Philosophy honor) from UCLA in 2002, all in computer science. He is currently a Professor with the University of Oregon, where he also directs the Network and Security Research Laboratory, Department of Computer and Information Science, and serves as the Founding Director of the Center for Cyber Security and Privacy. He has authored a research book on disseminating security updates over the Internet and over 100 peer-reviewed research papers. Currently, he is researching Internet monitoring and forensics, Internet privacy, software-defined networking, social networking, cloud computing, Internet of things, and various network security topics. His research is focused on computer networks, distributed systems, and network security. He has served on U.S. National Science Foundation research panels and 70 international technical program committees, including chairing several of them.

Lei Jiao received the Ph.D. degree in computer science from University of Göttingen, Germany. He is currently an assistant professor at the Department of Computer and Information Science, University of Oregon, USA. Previously he worked as a member of technical staff at Alcatel-Lucent/Nokia Bell Labs in Dublin, Ireland and also as a researcher at IBM Research in Beijing, China. He is interested in exploring optimization, control, learning, mechanism design, and game theory to manage and orchestrate large-scale distributed computing and communication infrastructures, services, and applications. He has published papers in journals such as JSAC, TON, TMC, and TPDS, and in conferences such as MOBIHOC, INFOCOM, ICNP, ICDCS, SECON, and IPDPS. He served as a guest editor for IEEE JSAC Series on Network Softwarization and Enablers. He was on the program committees of many conferences including MOBIHOC, INFOCOM, ICDCS, and IWQoS, and was the program chair of multiple workshops with INFOCOM and ICDCS. He was also a recipient of the Best Paper Awards of IEEE CNS 2019 and IEEE LANMAN 2013, and the 2016 Alcatel-Lucent Bell Labs UK and Ireland Recognition Award.

Jun Cai received the B.S degree from Hunan Normal university, Changsha, China, the M.S degree from Jinan University, Guangzhou, China, and the Ph.D. degree from Sun Yat-Sen University, China in 2003, 2006 and 2012, respectively. He is currently a professor with the School of Cyber Security, Guangdong Polytechnic Normal University, Guangzhou, China. He is interested in the research of network function virtualization (NFV), software-defined networks (SDN) and complex network.