Deploying Chains of Virtual Network Functions: On the Relation Between Link and Server Usage

Tung-Wei Kuo, Bang-Heng Liou, Kate Ching-Ju Lin, *Senior Member, IEEE*, and Ming-Jer Tsai, *Member, IEEE, ACM*

***Abstract*— Recently, network function virtualization has been proposed to transform from network hardware appliances to software middleboxes. Normally, a demand needs to invoke several virtual network functions (VNFs) following the order determined by the service chain along a routing path. In this paper, we study the joint problem of the VNF placement and path selection to better utilize the network. We discover that the relation between the link and server usage plays a crucial role in the problem. Inspired by stress testing, we first propose a systematic way to elastically tune the link and server usage of each demand based on the network status and properties of demands. In particular, we compute a proper routing path length, and decide, for each VNF in the service chain, whether to use additional server resources or to reuse resources provided by existing servers. We then propose a chain deployment algorithm that follows the guidance of this link and server usage. Via simulations, we show that our design effectively adapts resource allocation to network dynamics and, hence, serves more demands than other heuristics.**

***Index Terms*— Network function virtualization, network function deployment, routing.**

I. Introduction

ENTERPRISE networks nowadays consist of multiple intermediate Network Functions (NFs), such as Network Address Translators (NATs), firewalls, Intrusion Detection Systems (IDSs), Intrusion Prevention Systems (IPSs), and WAN optimizers [2]. These NFs, however, are implemented as expensive special-purpose hardware-based appliances. Moreover, hardware lifecycle is becoming shorter due to acceleration of technology development [3]. To remedy these issues, Network Function Virtualization (NFV) [3] has recently been proposed to transform from hardware to software middleboxes. It is envisioned that 5G networks will be built on NFV, and several telecom operators have started the deployment of SDN and NFV, e.g., AT&T and Verizon [4]. In particular, NFV enables an NF to be executed in virtual machines (VMs) hosted on commodity servers rather than special-purpose hardware [5]. Such virtualization makes service deployment more flexible and scalable. The efficiency of a network, however, is highly affected by the deployment of these software middleboxes, or called Virtual Network Functions (VNFs), in the network, which is the focus of this work.

Network traffic usually needs to pass through several VNFs in a particular order. For instance, a flow might go through IDS, which detects suspicious requests, before the proxy server. This phenomenon is known as *network function chaining* or *service chaining* [6]. Since VNFs might be deployed in virtual machines hosted by different physical servers, a service chain should traverse through those physical servers along a path. Hence, supporting service chains requires not only VMs but also link bandwidth, both are limited resources in a network. The problem of allocating resource and orchestrating NFs, i.e., NFV orchestration [7], has drawn much attention in recent years, and several NFV platforms, including OpenStack [8], Open Source MANO [9], and OPNFV [10], also focus on this problem. In this paper, we aim to improve NFV orchestration by cherry-picking the relation between VM usage and link usage. Intuitively, different service chains that demand the same VNF can share the same VM running that VNF if the computational power of the VM is sufficient to support their demands [11], [12].[1] Though such sharing improves VM utilization, it might consume more link bandwidth because these chains may need to go through a longer path in order to reach the shared VM. In this paper, we argue that the interplay between link usage and VM usage plays an important role in NFV orchestration, and we apply the idea to admission control, which has a huge impact on system capacity. Evidently, increasing the capacity of a system is a fundamental issue for many systems, including 5G networks and NFV service providers. Hence, we consider the following question: *How should we jointly allocate link capacity and available VMs to service chains so as to maximize the served traffic demands?*[2]

Manuscript received March 19, 2017; revised February 2, 2018; accepted May 4, 2018; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor K. Argyraki. Date of publication June 20, 2018; date of current version August 16, 2018. A preliminary version of this paper appeared in the proceedings of IEEE INFOCOM'16, San Francisco, CA, USA, April 2016 [1]. *(Corresponding author: Tung-Wei Kuo.)*

T.-W. Kuo is with the Department of Computer Science, National Chengchi University, Taipei 11605, Taiwan (e-mail: twkuo@cs.nccu.edu.tw).

B.-H. Liou was with the Department of Computer Science, National Tsing Hua University, Hsinchu 30013, Taiwan. He is now with Phison Electronics Corporation, Inc., Jhunan 350, Taiwan.

K. C.-J. Lin is with the Department of Computer Science, National Chiao Tung University, Hsinchu 300, Taiwan (e-mail: katelin@cs.nctu.edu.tw).

M.-J. Tsai is with the Department of Computer Science, National Tsing Hua University, Hsinchu 30013, Taiwan, and also with the Institute of Communications Engineering, National Tsing Hua University, Hsinchu 30013, Taiwan (e-mail: mjtsai@cs.nthu.edu.tw).

Digital Object Identifier 10.1109/TNET.2018.2842798

[1]If, for the sake of security or other reasons, it is inappropriate to let different service chains that demand the same VNF, say A, to share a VM that runs VNF A, we will treat VNF A on these service chains as different VNFs, e.g., A_1, A_2 with $A_1 \neq A_2$, to avoid insecure VM sharing. Throughout this paper, we will assume that the above renaming method has been used to preserve security while enabling VM sharing.

[2]Our solution can be easily generalized to maximize the total utility of served demands.

To see why we should jointly allocate link capacity and available VMs, consider the following examples. When link capacity is very limited and, by contrast, the number of VMs is more than sufficient to support all traffic demands, we should route the demands on the shortest paths and just launch idle VMs along the shortest paths to configure the required VNFs. On the contrary, when links have plentiful capacity but only a few VMs are available, we should, instead, reuse as many VNFs as possible even if a longer path may be required to traverse through those shared VNFs. The above two extreme cases demonstrate that proper link and VM resource allocations are highly correlated with each other and, more importantly, change with the network status. While finding an efficient chain deployment for the aforementioned extreme network status might be simple, this problem is however challenging in general network statuses. This is because the relation between the link resource allocation (in terms of the path length) and the VM resource allocation (in terms of the number of reused VMs), referred to as the *LV relation*, is unsettled in efficient deployments.

Unfortunately, in this paper, we show that, finding a feasible resource allocation is NP-hard (Theorem 1). This hardness result makes it more difficult to find a good LV relation. To bypass this difficulty and to find a proper LV relation, we need a way to assess an LV relation without finding a corresponding resource allocation. To this end, we propose a mathematical program to answer the following questions:

Q1) How to assess an LV relation for a given demand?

Q2) How to *efficiently* find a proper LV relation for a given demand?

Our idea of answering Q1 is inspired by *stress testing*. Stress testing (sometimes called torture testing) is a form of deliberately intense or thorough testing used to determine the stability of a given system or entity. It involves testing beyond normal operational capacity, often to a breaking point, in order to observe the results [13]. At a high level, we assess an LV relation by how much stress it can handle, and the best LV relation should be able to handle the highest stress. Specifically, in this paper, higher stress for a system implies more admitted demands in the system, and a breaking point refers to the point where no more demands can be admitted in the system. In this paper, we will perform stress testing under different system configurations (i.e., under different LV relations) to find the best configuration. By the proposed mathematical program, we can efficiently approximate the result of stress testing, and thus, answer Q2.

After solving the mathematical program to obtain a proper LV relation as guidance, we then design a service chain deployment algorithm by dynamic programming to find a path and VNF placement whose link and VM usage approaches the LV relation obtained by stress testing. The evaluation via simulations demonstrates that our stress-testing-inspired chain deployment outperforms greedy-based and shortest-path-based heuristics. We also test our solution with different LV relations as the guidance for the deployment algorithm. The simulation results show that the deployment following the LV relation obtained by stress testing performs better than those following other LV relations, which reflects the effectiveness of the stress-testing-inspired mathematical program. The following are the contributions of this paper:

- We define the **J**oint **V**NF Placement and **P**ath Selection (JVP) Problem and prove its NP-hardness (Section III).
- We propose a stress-testing-inspired mathematical program to find a proper LV relation. Moreover, the mathematical program can be solved efficiently (Section IV).
- Using dynamic programming, we design a deployment algorithm that approaches a given LV relation (Section V).
- We evaluate our solution in different types of data center network topologies. The result demonstrates the superiority of our solution over other greedy-based and shortest-path-based heuristics (Section VI).

II. Related Work

In this section, we discuss research related to virtual network function deployment. For more information, curious readers may refer to the survey by Herrera and Botero on resource allocation in NFV environments, which covers topics broader than virtual network function deployment [14]. We will also consider research that is not covered in [14] but is closely related to ours. More importantly, we use a taxonomy different from the one in [14] to better understand the difference between our work and others. Basically, all the research on virtual network function deployment considers a set of demands (service chains) as inputs and allocates resource to these demands. Most research relies on Mixed Integer Linear Program (MILP) to solve the problem as it is usually NP-hard to find a feasible solution. Some of the research proposes heuristics to solve the MILP. We categorize the related work into two groups based on the objective. The first is to serve all the input demands (service chains) and to minimize a certain cost function, e.g., Capital Expenditures (CAPEX), link utilization, server utilization, or some combination of the above. By minimizing the cost function, the hope is to allocate resource efficiently. The second is admission control, in which serving all the demands is extremely difficult, if not impossible. Hence, the goal becomes to maximize the admitted demands. Our work falls into the second category.

There is a substantial volume of research in the first group [7], [12], [15]–[33]. As described above, these works try to optimize some objective function subject to the constraints that all input demands are served. However, feasible solutions that can serve all the demands may not exist. Hence, these solutions are best suitable for the case where feasible resource allocation is guaranteed to exist in advance. Moreover, as efficient resource allocation is found by means of optimizing the objective function, it is critical to consider the link utilization and server utilization jointly as a single objective function. However, it is unclear how to capture the interplay between server utilization and link utilization in the objective function. Hence, some works simply consider link utilization alone [12], [15]–[19], [30], [31] or server utilization alone [12], [15], [20]–[22], [32] in the objective function. While some works indeed consider both link utilization and server utilization [7], [23]–[29], [33], the objective function is simply a weighted sum of link utilization and server utilization, or the maximum between link utilization and server utilization (link utilization and server utilization have equal weights). In these works, the weights are either fixed or are leaved for the system operator to decide. As we have stated in Section I, the relation between link and server usage varies with the system status. Hence, the selection of weights should be adapted to the system status. Unlike these works, our work directly resolves the LV relation, which may provide some guidance to the weight selection.

The works in the second group then focus on admission control, where the goal is to maximize the served demands [11], [34]–[42]. The work by Lukovszki and Schmid [34] and the work by Mijumbi *et al.* [35], however, do not consider link bandwidth constraint in resource allocation. Other works then consider both server capacity constraints and link capacity constraints at the same time [11], [36]–[42]. The major difference between the problem studied in our work and these works is that in these works, the resource in a physical server can be partitioned arbitrarily so that each part fits the demand (virtual network function) perfectly. On the other hand, we consider networks in which physical servers are partitioned into VM slots in advance, which is a common setting in data center networks, and much research also adopts such an environment, e.g., [19], [35], [43]. In this kind of environment, we do not directly allocate resources like CPUs and RAMs to a VNF. Instead, we allocate VM slots to a VNF, and one VM slot is only allocated to one VNF. Details of the settings can be found in Section III. Considering VM slots makes the LV relation even more complicated. This is because one more constraint is now associated with server utilization in such an environment. That is, the number of VM slots in a server is limited. This additional constraint explicitly poses an upper bound on the number of VNFs deployed in a server. In this paper, we will consider networks with such predetermined VM slots, which is common in data center networks.

III. Joint VNF Placement and Path Selection

A. Problem Definition

We model a network as an edge-weighted vertex-weighted directed graph $G = (V, E)$, in which V is the set of physical nodes in the network, e.g., switches and servers. c_e is the link capacity of edge (link) $e \in E$ and c_v is the number of virtual machines (VMs) that can be hosted on physical node $v \in V$. In this paper, we assume that the bandwidth between VMs on the same server is large enough to support the highest possible amount of traffic among VMs on the same server. We collect all the VMs hosted on different servers as a set M. Each VM $m \in M$ has a limited computational capability c_m, which can be specified by system operators or measured in real systems. For example, c_m can be measured by the supportable number of instructions per second (IPS) bounded by memory or CPU resources of VM m. Say the system supports a set of VNFs F. We assume that each VM m can run at most one VNF $f \in F$, while different VMs might support the same VNF. For example, by default, a VM in Google Cloud Platform hosts one application [44]. Moreover, some experiments also indicate that the performance drops when a VM hosts multiple NFs [45], [46]. Finally, several NFV platforms, including OpenStack, Open Source MANO, and OPNFV, assume that each NF is a monolithic VM [47]. Let $v(m)$ and $f(m)$ be the server hosting VM m and the VNF deployed in m, respectively.

Say we are given a set of demands $\mathcal{D} = \{d_1, d_2, \cdots\}$. Let $o_i \in V$ and $t_i \in V$ represent the source and destination vertex of d_i, respectively. Each demand $d_i \in \mathcal{D}$ requests for going through a chain of services $\mathcal{S}_i = (s_{i,1}, s_{i,2}, \cdots)$, and we define $|\mathcal{S}_i|$ as the length of the service chain $\mathcal{S}_i$. Each demand d_i has a traffic rate R_i (bytes/second), and each $s_{i,j}$ (the j^{th} service request of demand d_i) consumes a VM's computational load $L(s_{i,j})$ (e.g., IPS). To obtain the traffic rate and the computational load, we can first admit the demand temporarily, and then obtain the required information by some monitoring services. Most cloud services now include monitoring APIs, and the design of monitoring systems is left outside the scope of this paper. We similarly use the notation $f(s_{i,j})$ to denote the VNF requested by $s_{i,j}$, and assume that a demand can request the same VNF more than once in the chain. That is, $f(s_{i,j})$ could be the same with $f(s_{i,j'})$ for any j and j'. We further assume that each $s \in \mathcal{S}_i$ can only be served by a single VM. However, to better utilize resources, we allow different service requests to share a single VM if its computational capability c_m is sufficient [11], [12].

Given a set of demands, this paper investigates the **J**oint **V**NF Placement and **P**ath Selection (JVP) Problem, which jointly determines the solutions of the following problems:

1) **Path selection and admission control** (p_i): For each accepted demand d_i, we have to find a routing path p_i for it. Otherwise, we set $p_i = \varnothing$ if d_i is rejected.
2) **VNF assignment and placement** ($m(s_{i,j})$, $I(s_{i,j})$, $f(m)$): If a demand d_i is accepted, we have to assign each service request $s_{i,j} \in \mathcal{S}_i$ to a VM hosted on a certain vertex (server) along the selected path p_i. To formally express this assignment, we let $m(s_{i,j})$ denote the VM to which $s_{i,j}$ is assigned. We further define VNF placement as $I(s_{i,j})$, which indicates the *index* of the physical server in path p_i that hosts the VM $m(s_{i,j})$. In particular, if $s_{i,j}$ is assigned to a VM m hosted by the k^{th} vertex along path p_i, then we define $m(s_{i,j}) = m$ and $I(s_{i,j}) = k$. In addition, this implies that we should place the requested VNF $f(s_{i,j})$ on VM m, i.e., $f(m) = f(s_{i,j})$.

JVP's goal is to maximize the sum rate of the admitted demands, i.e., $\max \sum_{d_i : p_i \neq \varnothing} R_i$. In this paper, we also consider the objective function of maximizing the total utility of the admitted demands, i.e., $\max \sum_{d_i : p_i \neq \varnothing} U_i$, where U_i is the utility associated with demand d_i. We have the following constraints.

Path Constraint: If demand d_i is accepted, then p_i is a path connecting from o_i to t_i.

Link Capacity Constraint: The selected routing path p_i should carry a flow of rate R_i, and all flows in the network need to satisfy the capacity constraint of each link. Since $\mathcal{S}_i$ might be deployed on a path with loops (e.g., a path traversing through several VMs hosted on the same server), we should consider the total link bandwidth consumed by the path. That is, $\sum_{d_i \in \mathcal{D}} \delta_e(p_i) R_i \le c_e, \forall e \in E$, where $\delta_e(p_i)$ is the number of times that the path p_i traverses through link e.

VNF Placement Constraint: Each VM can at most support one VNF, but can be shared by multiple demands. This means that, if multiple services are assigned to the same VM, those services must request the same VNF. Namely, if $m(s_{i,j}) = m(s_{i',j'})$, then $f(s_{i,j}) = f(s_{i',j'}), \forall i, i', j, j'$.

VM Capacity Constraint: Each VM m can only support a number of demands, limited by its computational capability. More specifically, $\sum_{s_{i,j}: m(s_{i,j})=m} L(s_{i,j}) \le c_m, \forall m \in M$.

Chaining Constraint: The path p_i of an accepted demand d_i must be able to traverse through VNFs following the order specified in $\mathcal{S}_i$, i.e., $I(s_{i,j}) \le I(s_{i,j'}), \forall s_{i,j}, s_{i,j'} \in \mathcal{S}_i, j < j'$.

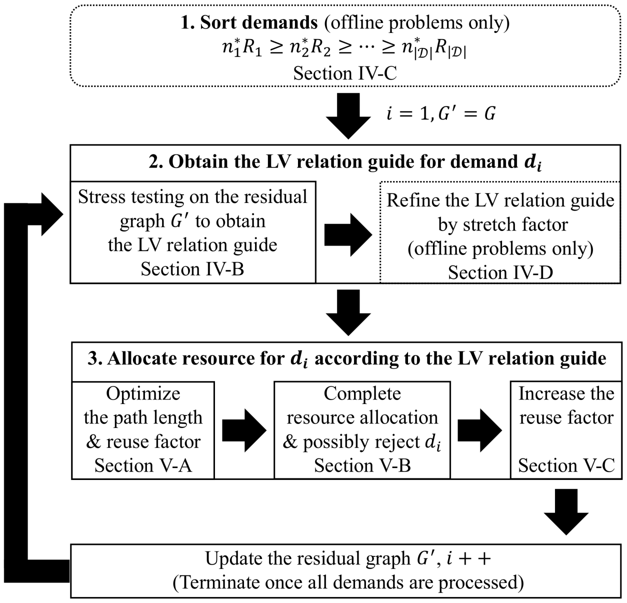
JVP is an extremely difficult problem. In fact, even finding a non-trivial feasible solution of JVP for a single demand is NP-hard. To see this, we first introduce an NP-hard problem, called the edge-disjoint path (EDP) problem [48]. Given a directed graph and two pairs of vertices (o_1, t_1) and (o_2, t_2), the EDP problem asks to connect these two pairs from o_i to t_i via edge-disjoint paths. We then reduce the NP-hard EDP problem to our JVP problem. In particular, given the graph G in the EDP problem, we construct the graph G' in the JVP problem as follows. Initially, $G' = G$. We then add a node r, which hosts the only VM in G', and add two directed edges (t_1, r) and (r, o_2) in G'. All the edges in G' have a link capacity R. The only demand d originates from o_1 and terminates at t_2 with a demanding rate R, and only requests a single VNF service. It is then easy to see that the EDP problem has a feasible solution if, and only if, the JVP problem has a feasible solution with an objective value greater than 0. We then have the following theorem.

Theorem 1: It is NP-hard to find a feasible solution, whose objective value is greater than 0, of the JVP problem. Thus, if $P \neq NP$, then there is no polynomial time approximation algorithm for the JVP problem with bounded ratio.

B. An Overview of Our Solution

Since it is difficult to find the solution for all the demands as a whole, we, instead, consider each demand sequentially, and decide whether and how to serve it. Therefore, as considering a demand d_i, some link/VM resources might have been allocated to some other demands $d_{i'}, i' < i$. A side benefit of such sequential deployment is that this is especially suitable for solving the online version of the JVP problem, where demands may arrive at different times.

Intuitively, deploying a chain on a shorter path consumes less link bandwidth, but might also reduce VM reuse opportunities. On the contrary, reusing more VMs might lead to a longer path, which consumes more link bandwidth. Hence, there exists a dilemma of saving more link bandwidth or reusing more VMs. We refer to the number of services in $\mathcal{S}_i$ that reuse existing VMs as the *reuse factor*. To allocate resource for a demand d_i, we first find a proper path length and a proper reuse factor for d_i by a mathematical program that simulates stress testing. The proper path length and the proper reuse factor then form an *LV relation guide*. We then design a



1. Sort demands (offline problems only)
$n_1^* R_1 \ge n_2^* R_2 \ge \cdots \ge n_{|\mathcal{D}|}^* R_{|\mathcal{D}|}$
Section IV-C
$i = 1, G' = G$
2. Obtain the LV relation guide for demand d_i
Stress testing on the residual graph G' to obtain the LV relation guide Section IV-B
Refine the LV relation guide by stretch factor (offline problems only) Section IV-D
3. Allocate resource for d_i according to the LV relation guide
Optimize the path length & reuse factor Section V-A
Complete resource allocation & possibly reject d_i Section V-B
Increase the reuse factor Section V-C
Update the residual graph G', $i++$
(Terminate once all demands are processed)

Fig. 1. The roadmap of our solution.

usage-guided deployment algorithm to find a routing path and VNF assignment whose resource consumption approaches the LV relation guide.

The Roadmap of Our Solution (Fig. 1):

1) **Sort the demands.** We first apply the mathematical program to each demand on the initial network. We then use the output of the mathematical program to sort the demands (Section IV-C). Note that, for the online version of our problem, we do not sort the demands, and we process each demand immediately once it arrives.
2) **After sorting the demands, we then process each demand sequentially. For each demand d_i:**
   a) **Obtain the LV relation guide for d_i.** This is done by applying the mathematical program to d_i with respect to the current network status (Section IV-B). Unless we are considering the online problem, we further refine the LV relation guide (Section IV-D).
   b) **Allocate resource for d_i according to the LV relation guide.** We first find an incomplete resource allocation that jointly optimizes the path length and reuse factor. This is achieved by integrating a greedy heuristic and a dynamic program (Section V-A). So far, the path may not reach the destination yet, and some trailing services in the service chain may still not be assigned to VMs. We then complete the resource allocation by assigning these trailing services to empty VMs and extend the routing path to the destination. If the path length exceeds that in the LV relation guide, we reject the demand (Section V-B). Finally, we increase the reuse factor (Section V-C).

IV. Obtaining the LV Relation Guide by Stress Testing

In this section, the goal is to find a proper path length and a proper reuse factor for a given demand d_i, i.e., the LV relation guide for d_i. Obviously, the LV relation guide should adapt to the current network status. Since the path length is closely related to the reuse factor x, we will further define a proper path length $l(x)$ as a function of a given reuse factor x. As mentioned in Section I, we find the LV relation guide by answering Q1 and Q2 with the help of stress testing. In this section, we first answer Q1 formally by stress testing. Next, we answer Q2 by a mathematical program that can approximate the result of stress testing efficiently. We then discuss the design of function $l(x)$. Finally, we show how to sort the demands and refine the LV relation guide in Sections IV-C and IV-D, respectively.

A. Solving Q1 by Stress Testing

Because the path length is modeled as a function of reuse factor, we only need to focus on assessing a reuse factor. Intuitively, if x is an efficient reuse factor for demand d, then we should be able to solve the JVP problem optimally by letting d reuse x VMs. However, assessing a reuse factor by this idea is almost as difficult as the JVP problem. Hence, we approximate the best reuse factor for d. Specifically, we consider a stress testing system whose initial status is identical to the current network status. The stress is generated by introducing an excessive number of demands. These introduced demands are all identical to d and all use the same reuse factor. In other words, we treat the reuse factor as a system-wide configuration to simplify the problem and amplify the impact of the choice for the reuse factor. We are interested in the result of the above stress test, i.e., the maximum number of demands that can be admitted in the above system. The desired reuse factor is the one that yields the best result. To get the result of the above stress test, we ask the following question: if there exist an infinite number of identical demands d, each assigned the same reuse factor x, then what is the maximum number of demands that can be served in the system?

B. Approximating the Result of Stress Testing by a Mathematical Program

In fact, the above problem is still NP-hard (since finding a feasible resource allocation is NP-hard).[3] Therefore, we design a method to solve this problem without really deploying demands. Intuitively, the total link capacity imposes an upper bound of the solution. Similarly, the number of VMs in the system also imposes an upper bound. The high-level idea of the following mathematical problem is then to model these two upper bounds and to find the smaller one between these two bounds.

$$
(ST)\qquad
\begin{aligned}
&\text{maximize } n\\
&\text{subject to } n \cdot R_i \cdot l(x) \le C_l &&\text{(1a)}\\
&\qquad n \cdot (|\mathcal{S}_i| - x) \le C_{em} &&\text{(1b)}\\
&\qquad x \in \{0, 1, 2, \cdots, \bar{x}\} &&\text{(1c)}
\end{aligned}
$$

[3]In hindsight, we approximate the problem of finding the best reuse factor twice. The first time occurs when we simplify the problem (to get the problem formulation in Section IV-A). The second time occurs when we solve this simplified problem by ST.

In ST, given a reuse factor x for each served demand, we try to find the approximate maximum number of demands n that can be served in the system. Specifically, the system is modeled as a *residual graph* G', which is obtained from G by excluding links with remaining capacity less than R_i and then deleting all the vertices that o_i cannot reach. This approximate result n is obtained by considering two resource constraints. Eq. (1a) restricts that the total bandwidth resources used by the served demands cannot exceed C_l, the total remaining link capacity of G'. Then, Eq. (1b) ensures that the number of empty VMs assigned to the served demands is no larger than C_{em}, the number of empty VMs in G'. Eq. (1c) gives the search range of x. Intuitively, the upper bound of the reuse factor can be set to $\bar{x} = |\mathcal{S}_i|$. This is, however, an overestimate because some service requests might never find a reusable VM. Hence, we define $\mathcal{S}'_i$ as the set of services that have reuse opportunities. Specifically, we set $\mathcal{S}'_i = \mathcal{S}_i$ initially, and remove any $s_{i,j}$ from $\mathcal{S}'_i$ if there exists no reusable VM that runs VNF $f(s_{i,j})$ and is reachable from o_i. The upper bound can then be tightly set to $\bar{x} = |\mathcal{S}'_i|$. Since there are only $(\bar{x} + 1)$ possible values of x, we can solve n with respect to each possible x, and find the optimal x in ST, denoted as x^*. Hence, x^* is the best reuse factor obtained by approximating the result of stress testing. x^* and $l(x^*)$ then define the LV relation guide and will be used as guidance for the deployment algorithm in Section V.

Relating x With $l(x)$: While ST is easy to solve, a difficult problem remains unsolved: how to estimate the relation between $l(x)$ and x? We start by giving a high-level intuition of how we estimate $l(x)$ for any given x, and then describe the detailed derivation. If a service chain is deployed on a path reusing x existing VMs, the path can be partitioned into $(x + 1)$ sub-paths. To save link bandwidth, ideally, we should pick the shortest path in G' as the sub-path to connect any two consecutive reused VMs (or o_i and t_i). However, since the rest of services that are not served by any reusable VM can only be deployed in empty VMs, the servers along the shortest sub-paths might not have sufficient empty VMs to support those services. Hence, we might extend the shortest sub-paths to locate more empty VMs, and should further consider additional link resources required for this detour. The above intuition can be expressed as follows:

$$
l(x)=\begin{cases} |\pi_{o_i,t_i}| + l_\Delta[|\mathcal{S}_i| - n_{e,o_i,t_i}]^+, & x = 0\\ (x+1)l_r + l_\Delta[|\mathcal{S}_i| - x - (x+1)n_e]^+, & x > 0, \end{cases} \tag{2}
$$

where $|\pi_{o_i,t_i}|$ is the length of the shortest path π_{o_i,t_i} from o_i to t_i on G', l_Δ is the average distance between any two empty/reusable VMs, where one must be empty, n_{e,o_i,t_i} is the number of empty VMs on π_{o_i,t_i}, l_r is the average distance between any two reusable VMs (here o_i and t_i are also deemed as reusable VMs), n_e is the average number of empty VMs between any two reusable VMs, and $[\cdot]^+ \triangleq \max(\cdot, 0)$. We discuss our derivation in two cases: $x = 0$ and $x > 0$. In both cases, the first term in Eq. (2) is the total (estimated) length of the $x+1$ shortest sub-paths, and the second term is the total number of extra links used to extend the sub-paths for reaching empty VMs. When no VM is reused ($x = 0$), it is easy to estimate $l(x)$ since we can find the shortest path π_{o_i,t_i} and count the number of available empty VMs n_{e,o_i,t_i}. However, when $x > 0$, we do not explicitly know which VMs are reused before deployment, and can only estimate the average shortest distance, l_r, and the average number of empty VMs, n_e, between two reusable VMs.

To finalize our derivation, it is then sufficient to estimate l_r and n_e. To do so, we generate $|\mathcal{S}'_i|$ vertex sets, U_j, $1 \leq j \leq |\mathcal{S}'_i|$, each of which collects all the vertices (servers) that host reusable VMs serving VNF $f(s_{i,j})$. That is, $U_j = \{v(m) : m \in M, f(m) = f(s_{i,j}), L(s_{i,j}) \leq c_m\}$. Recall that $\mathcal{S}'_i$ is obtained from excluding the services that cannot find any reusable VM from $\mathcal{S}_i$, so U_j must be non-empty. We further introduce $U_0 = \{o_i\}$ and $U_{|\mathcal{S}'_i|+1} = \{t_i\}$. To meet the chaining constraint, a feasible path for d_i needs to go through any vertex in U_j in an ascending order of j. For ease of description, we define, for each j, a set $\mathcal{P}_j = \{(v, v') : v \in U_j, v' \in U_{j+1}, \pi_{v,v'} \neq \varnothing\}$ to include any pairs of reachable vertices, one in U_j and the other in U_{j+1}. By reachable, we mean that there exist a path from v to v' on G'. Then, the average distance between any two reusable VMs, l_r, can be estimated by the average length of the shortest paths $\pi_{v,v'}$ between $(v, v') \in \mathcal{P}_j, \forall 0 \leq j \leq |\mathcal{S}'_i|$.

$$l_r = \frac{1}{|\mathcal{S}'_i| + 1} \sum_{0 \leq j \leq |\mathcal{S}'_i|} \frac{\sum_{(v,v') \in \mathcal{P}_j} |\pi_{v,v'}|}{|\mathcal{P}_j|}. \tag{3}$$

Likewise, we have

$$n_e = \frac{1}{|\mathcal{S}'_i| + 1} \sum_{0 \leq j \leq |\mathcal{S}'_i|} \frac{\sum_{(v,v') \in \mathcal{P}_j} n_{e,v,v'}}{|\mathcal{P}_j|}, \tag{4}$$

where $n_{e,v,v'}$ is the number of empty VMs on $\pi_{v,v'}$.

C. Sorting Demands According to the Result of Stress Testing

Recall that, to solve the JVP problem, we sequentially process each demand d_i, $i = 1, \cdots, |\mathcal{D}|$. The performance of such sequential deployment closely depends on the order of the demands that are processed. Intuitively, to increase the total size of the admitted demands, we would like to process beneficial demands first. Hence, we sort the demands in the descending order of $n_i^* R_i$, where n_i^* is the optimal n outputted by ST for demand d_i on the initial network, and then consider each demand in order. The rationale behind this sorting is that a larger n_i^* implies that the demand d_i requires fewer resources and a higher R_i results in a higher performance improvement. Similarly, if the goal is to maximize the total utility of admitted demands, then we sort the demands in the descending order of $n_i^* U_i$, where U_i is the utility associated with demand d_i.

D. Refining the LV Relation Guide

ST attempts to find an LV relation guide that enables us to accept as many demands as possible. However, in reality, we may not have so many demands in total, and, as a result, the resources could eventually be underutilized. As we will see in the next section, we reject a demand when we cannot find a resource allocation that follows the LV relation guide. Thus, conservative LV relation guides may lead to unnecessary rejections. To avoid conservative LV relation guides, our intuition is that we should not waste the system resource. In other words, the total LV relation guide of all demands should fully utilize the total resource. Therefore, we scale up the path length guide of every demand (by multiplying the path length guide by a factor, SF) so that the resulting total link consumption guide of all demands (i.e., $\sum_{d_i \in \mathcal{D}} R_i(SF \cdot l(x_i^*))$, where $l(x_i^*)$ is the path length guide of demand d_i outputted by ST) equals the total link capacity of the system (i.e., $\sum_{e \in G} c_e$). Moreover, we do not want to reduce the path length guide after scaling. Thus, SF cannot be less than one. SF is called the stretch factor, and can be determined by the following equation.[4]

$$SF = \max\left(\frac{\sum_{e \in G} c_e}{\sum_{d_i \in \mathcal{D}} R_i \cdot l(x_i^*)}, 1\right) \tag{5}$$

Let $l_i^* = SF \cdot l(x_i^*)$ be the new path length guide for demand d_i. For the online problem, because we cannot compute the total link consumption guide of all demands, we set $l_i^* = l(x_i^*)$.

V. The Usage-Guided Deployment Algorithm

Given a demand d_i and its LV relation guide (l_i^*, x_i^*), the goal of our *usage-guided* deployment algorithm is to find a path and VNF assignment such that the selected path can reuse approximately x_i^* VMs and has a length close to l_i^*. If no such path exists, we reject d_i. A naive way to find such a solution is to examine all the possible paths and assignments. This is however extremely computationally expensive, and can hardly be realized in practice. Hence, we, instead, incrementally find sub-paths that most efficiently reuse existing VMs under the given path length constraint, l_i^*, and concatenate those sub-paths as the routing path (see Section V-A). If the resulting assignments cannot serve all the requested services in $\mathcal{S}_i$ or the path has not reached the destination yet, we next find the last sub-path to complete the whole path and assignments (see Section V-B). Finally, if the resulting deployment still underutilizes the specified reuse factor x_i^*, we adjust the path and assignments so as to improve the reuse factor (see Section V-C). Note that, since we cannot use links with capacity less than R_i, the following algorithms operate on the residual graph G', which is a subgraph of G. Specifically, whenever the routing path is extended or VNF assignments are updated, the remaining resources on G' are updated accordingly and links with insufficient capacity are removed from G'.

A. Finding Effective Sub-Paths Using Dynamic Programming

To deploy a demand d_i, we find a *shortest sub-path* connecting from source o_i to an intermediate vertex v, and iteratively find the next *shortest sub-path* from v to extend the path. Algorithm 1 shows the pseudo-code. Let $\hat{p}_{i,k}$ denote the sub-path found in the k^{th} iteration, and let $p_{i,k}$ denote the merged path until the k^{th} iteration, namely

[4]SF can be updated periodically (e.g., whenever ten more demands are processed by the deployment algorithm) by updating the total remaining link capacity in the numerator and the total link consumption guide of all demands that have not been processed by the deployment algorithm in the denominator.

Algorithm 1 Path-Extension

```
1  $v_0 \leftarrow o_i$
2  $p_{i,0} \leftarrow \varnothing$, $m(s_{i,j}) \leftarrow \varnothing, \forall s_{i,j} \in \mathcal{S}_i$ // initialize assignments
3  /* iteratively extend the path */ for $k \leftarrow 1$ to $\infty$ do
4      $g^* \leftarrow 0$ // initialize the reuse gain
5      /* test each intermediate vertex */ for $v' \in G'$ do
6          $p \leftarrow p_{i,k-1} + \pi_{v_{k-1},v'}$
7          if $|p_{i,k-1}| + |\pi_{v_{k-1},v',t_i}| > l_i^*$ then continue
8          Obtain assignments $m^*(p, \mathcal{S}^*)$ and $r(p)$ by Eqs. (6,7)
9          if $r(p) \le r(p_{i,k-1})$ then continue
10         // $v'$ satisfies the Properties A and B
11         if $r(p)/|p| > g^*$ then
12             $v_k \leftarrow v'$, $m() \leftarrow m^*(p, \mathcal{S}^*)$, $g^* \leftarrow r(p)/|p|$
13             $\hat{p}_{i,k} \leftarrow \pi_{v_{k-1},v_k}$, $p_{i,k} \leftarrow p_{i,k-1} + \hat{p}_{i,k}$
14     if $g^* = 0$ then break
15     Update $c_e$s based on $p_{i,k}$ and reconstruct $G'$
16 Update remaining VM resources $c_m$ based on $m()$
17 return $p_{i,k}$, $m(s_{i,j}), \forall s_{i,j} \in \mathcal{S}^*$
```

$p_{i,k} = \sum_{k'=1}^{k} \hat{p}_{i,k'}$.[5] Also, v_k represents the intermediate vertex reached by sub-path $\hat{p}_{i,k}$, and $v_0 = o_i$. Thus, $\hat{p}_{i,k} = \pi_{v_{k-1},v_k}$, where π_{v_{k-1},v_k} is the shortest path from v_{k-1} to v_k on G'. The high-level idea of our sub-path selection is to incrementally maximize the reuse factor under the path length constraint. Therefore, a preferred sub-path should possess the following two properties. First (Property A), since the *final* path length cannot exceed l_i^*, we need to ensure $|p_{i,k-1}| + |\pi_{v_{k-1},v_k,t_i}| \le l_i^*$ for all k, where π_{v_{k-1},v_k,t_i} is the concatenation of π_{v_{k-1},v_k} and the shortest path π'_{v_k,t_i} that can support a flow of rate R_i from v_k to t_i after subtracting the link resources occupied by π_{v_{k-1},v_k} (Line 7). The shortest path π'_{v_k,t_i} is a conservative estimate to ensure that some path can connect v_k to the destination. Second (Property B), a newly-added sub-path $\hat{p}_{i,k}$ should improve the reuse factor, i.e., $r(p_{i,k}) > r(p_{i,k-1})$, where $r(p)$ is the maximum number of services in $\mathcal{S}_i$ that can be assigned to reusable VMs on path p (Line 9).

Following the above two properties, we iteratively merge the sub-paths, and terminate until no such sub-path can be found (Line 14). To find the most effective sub-path in iteration k, we try to optimize the reuse factor and the path length simultaneously. Hence, among all the possible intermediate vertices satisfying the two properties (Line 10), we extend the path to the one that maximizes $\frac{r(p_{i,k})}{|p_{i,k}|}$ (Line 13).

The remaining task is to calculate $r(p_{i,k})$. Note that the merged path $p_{i,k}$ in iteration k is only an incomplete path. Hence, we do not need to assign all the service requests in $\mathcal{S}_i$ on path $p_{i,k}$, but just need to assign a service sub-chain on $p_{i,k}$. There are only $|\mathcal{S}_i| + 1$ possible service sub-chains, i.e., $\varnothing$, $(s_{i,1})$, $(s_{i,1}, s_{i,2})$, $\cdots$, $\mathcal{S}_i$. To find $r(p_{i,k})$, we can simply try to deploy each possible sub-chain on $p_{i,k}$, and check which sub-chain gives the maximum reuse factor on $p_{i,k}$. This maximal reuse factor is then $r(p_{i,k})$. In particular, we should solve the following **S**ingle Demand Fixed Path **V**irtual Network Function **P**lacement (SVP) problem: Given a demand d with source o, destination t, a service sub-chain $\mathcal{S}$ and a routing path p connecting o and t, find an assignment $m(s_j)$ for each $s_j \in \mathcal{S}$ on the given path p such that the reuse factor is maximized. We let SVP$(p, \mathcal{S})$ represent an SVP problem instance of assigning $\mathcal{S}$ to VMs on path p. We further denote $m^*(p, \mathcal{S}) = \{m(s) : s \in \mathcal{S}\}$ as the solution of all the assignments, and set it to $m^*(p, \mathcal{S}) = \varnothing$ if there exists no feasible assignment.

We use dynamic programming to solve the SVP problem when the path has no loop. Intuitively, each instance SVP$(p, \mathcal{S})$ can be partitioned into two sub-instances SVP$(p-(v_{|p|}), \mathcal{S}'_h)$ and SVP$((v_{|p|}), \mathcal{S}''_h)$, where $v_{|p|}$ is the last vertex on path p, and $\mathcal{S}'_h = (s_1, \cdots, s_h)$ and $\mathcal{S}''_h = (s_{h+1}, \cdots, s_{|\mathcal{S}|})$ are the first and second parts of $\mathcal{S}$, respectively, broken at service s_h. Therefore, the basic idea of our dynamic programming is that, given any instance SVP$(p, \mathcal{S})$, we recursively solve the two sub-instances SVP$(p-(v_{|p|}), \mathcal{S}'_h)$ and SVP$((v_{|p|}), \mathcal{S}''_h)$, and combine these two solutions as the solution of SVP$(p, \mathcal{S})$. We use $\oplus$ to denote this combining operation. That is, $m(p, \mathcal{S}) = m(p-(v_{|p|}), \mathcal{S}'_h) \oplus m((v_{|p|}), \mathcal{S}''_h)$, and $m(p, \mathcal{S}) = \varnothing$ if any of $m(p-(v_{|p|}), \mathcal{S}'_h)$ and $m((v_{|p|}), \mathcal{S}''_h)$ is infeasible. We can then use dynamic programming to solve the SVP problem for all the possible chain breaking indexes h, and output the solution that achieves the maximal reuse factor. Specifically, the dynamic program can be written as:

$$m^*(p, \mathcal{S}) = \underset{m_h: 0 \le h \le |\mathcal{S}|}{\arg\max}\ r(m_h), \quad \text{where}$$
$$m_h = \begin{cases} m^*(p-(v_{|p|}), \mathcal{S}), & h = |\mathcal{S}| \\ m^*((v_{|p|}), \mathcal{S}), & h = 0 \\ m^*(p-(v_{|p|}), \mathcal{S}'_h) \oplus m^*((v_{|p|}), \mathcal{S}''_h), & 0 < h < |\mathcal{S}| \end{cases} \tag{6}$$

and $r(m_h)$ is the reuse factor of the solution m_h for SVP$(p, \mathcal{S})$ and is set to -1 if m_h is infeasible. Then, $r(p_{i,k})$ can be found by solving

$$\mathcal{S}^* = \underset{\mathcal{S} \in \{\varnothing, (s_{i,1}), (s_{i,1}, s_{i,2}), \cdots, \mathcal{S}_i\}}{\arg\max} r(m^*(p_{i,k}, \mathcal{S})), \tag{7}$$

and we get $r(p_{i,k}) = r(m^*(p_{i,k}, \mathcal{S}^*))$.

Claim 1: Eq. (6) is correct.

Proof: Please refer to the appendix. □

Two things are worth noting. First, any SVP problem on a *single-vertex* path is a base case of dynamic programming, which is easy to solve. In particular, to solve SVP$((v), \mathcal{S})$, we can assign as many services in $\mathcal{S}$ as possible to the reusable VMs in v, and then assign the rest of services to as few empty VMs in v as possible. If all the services in $\mathcal{S}$ are assigned to VMs in v, we return the above assignments as $m^*((v), \mathcal{S})$. Otherwise, the instance is infeasible, and we return $m^*((v), \mathcal{S}) = \varnothing$. The solutions of those base cases can then be recursively combined to find the solution of any sub-instance in Eq. (6). Second, a path with loops is allowed. Therefore, for a path p, if the last vertex $v_{|p|}$ is the

[5]For simplicity, we use $p_1 + p_2$ to denote the concatenation of p_1 and p_2, and use $p_1 - p_2$ to denote the operation of truncating p_2 from p_1.

same with some vertices in $p-(v_{|p|})$, the two sub-instances SVP($p-(v_{|p|}), \mathcal{S}'_h$) and SVP($(v_{|p|}), \mathcal{S}''_h$) cannot be solved independently. Otherwise, the capacity of a VM might be over-provisioned to different services in the two sub-instances. To avoid this issue, we first solve SVP($p-(v_{|p|}), \mathcal{S}'_h$), update the remaining VM resources based on its assignments, and then solve SVP($(v_{|p|}), \mathcal{S}''_h$) on the residual graph.

B. Making the Deployment Feasible

After executing Algorithm 1, we may still not be able to serve all the services in $\mathcal{S}_i$ and reach the destination. In fact, only the services in $\mathcal{S}^*$ solved in Eq. (7) are assigned to VMs. Hence, we further propose an algorithm to locate empty VMs for those unserved services in $\mathcal{S}_i$ (i.e., $s_{i,|\mathcal{S}^*|+1}, s_{i,|\mathcal{S}^*|+2}, \cdots, s_{i,|\mathcal{S}_i|}$), and find a *bridging path* p_b between v_k and t_i to link these empty VMs, where v_k is the last vertex of the path obtained by Algorithm 1, and t_i is destination of demand d_i. Algorithm 2 shows the pseudo-code. Our goal is to minimize the length of p_b. Initially, p_b only contains v_k (Line 1). We then deploy the first unserved service $s \in \mathcal{S}_i$ on an empty VM m such that $d_1(m) = |\pi_{v_L,v(m),t_i}|$ is minimized, where v_L is the last vertex on p_b, and $v(m)$ is the server hosting m (Line 6). If several such VMs can be found, we choose the one that minimizes $d_2(m) = |\pi_{v_L,v(m)}|$ (Line 6). Let m^* and v^* be the chosen VM and the server hosting the chosen VM, respectively (Line 7). p_b is thus extended to v^* by $|\pi_{v_L,v^*}|$ (Line 12). We then set v_L to v^*, update the remaining resources, and remove links with insufficient capacity from G' (Lines 13-14). The above procedure is repeated until the service chain is fully served or no empty VM can be reached, which leads to rejection of d_i (Line 10). If the service chain is fully served, p_b is then extended to t_i by π_{v_L,t_i} (Line 15), and the path for d_i, p_i, will be the concatenation of $p_{i,k}$ (obtained by Algorithm 1) and p_b (Line 16). If $|p_i|$ exceeds the specified length l_i^*, we reject d_i (Line 18).

C. Increasing the Reuse Factor

So far, the reuse factor might be less than x_i^*. To better reuse the VMs, for a service that has a reusable VM but is assigned to an empty VM, we can reassign it to a reusable VM and adjust the path accordingly. However, we should ensure that, after path adjustment, the path length is still no longer than l_i^*. Algorithm 3 shows the pseudo-code. The path adjustment is straightforward. Assume that we want to reassign a service $s_{i,j}$ to a reusable VM m. Then, the sub-path of p_i from $v(m(s_{i,j-1}))$ to $v(m(s_{i,j+1}))$, denoted by $\hat{p}_{old}(s_{i,j})$, is replaced with the subpath $\pi_{v(m(s_{i,j-1})),v(m),v(m(s_{i,j+1}))}$, denoted by $\hat{p}_{new}(s_{i,j},m)$. In addition, we want to keep the path as short as possible. Hence, we choose the service s and the reusable VM m for s such that $\Delta(s,m) = |\hat{p}_{new}(s,m)| - |\hat{p}_{old}(s)|$ is minimized. Let s^* and m^* be the chosen service and the chosen reusable VM, respectively (Line 7). We then reassign s^* to the reusable VM m^* by the above method (Lines 10-11). This process is performed repeatedly (Line 1) until all possible reassignments result in overlong paths (Line 9) or the reuse factor equals the specified usage (Line 13).

Algorithm 2 Bridging

```
1  $p_b \leftarrow v_k, v_L \leftarrow v_k$
2  for $h \leftarrow |\mathcal{S}^*|+1$ to $|\mathcal{S}_i|$ do
3      // find an empty VM for $s_{i,h}$
4      $d_1^{min} \leftarrow \infty, d_2^{min} \leftarrow \infty$
5      forall the empty VM $m$ do
6          if $d_1(m) < d_1^{min}$ or ($d_1(m) = d_1^{min}$ and $d_2(m) < d_2^{min}$) then
7              $m^* \leftarrow m, v^* \leftarrow v(m)$
8              $d_1^{min} \leftarrow d_1(m), d_2^{min} \leftarrow d_2(m)$
9      if $d_1^{min} = \infty$ then
10         Reject $d_i$
11         return
12     $m(s_{i,h}) \leftarrow m^*, p_b \leftarrow p_b + \pi_{v_L,v^*}$
13     $v_L \leftarrow v^*$
14     Update $c_{m^*}$ and $c_e$s and reconstruct $G'$
15 $p_b \leftarrow p_b + \pi_{v_L,t_i}$
16 $p_i \leftarrow p_{i,k} + p_b$
17 if $|p_i| > l_i^*$ then
18     Reject $d_i$
19     return
```

Algorithm 3 Increase-Reuse-Factor

```
1  while true do
2      $\Delta^{min} \leftarrow \infty$
3      forall the $s_{i,j} \in \mathcal{S}_i$ that is assigned to an empty VM do
4          forall the reusable VM $m$ for $s_{i,j}$ do
5              if $\Delta(s_{i,j},m) < \Delta^{min}$ then
6                  $\Delta^{min} \leftarrow \Delta(s_{i,j},m)$
7                  $s^* \leftarrow s_{i,j}, m^* \leftarrow m$
8                  $\hat{p}^*_{new} \leftarrow \hat{p}_{new}(s_{i,j},m)$, $\hat{p}^*_{old} \leftarrow \hat{p}_{old}(s_{i,j})$
9      if $|p_i| + \Delta^{min} > l_i^*$ then return $p_i, m(s_{i,j}), \forall s_{i,j} \in \mathcal{S}_i$
10     $m_{old} \leftarrow m(s^*), m(s^*) \leftarrow m^*$
11     Replace $\hat{p}^*_{old}$ with $\hat{p}^*_{new}$ in $p_i$
12     Update $c_e$s, $c_{m(s^*)}, c_{m_{old}}$ and reconstruct $G'$
13     if the reuse factor $= x_i^*$ then return $p_i, m(s_{i,j}), \forall s_{i,j} \in \mathcal{S}_i$
```

Implication of Theorem 1: A different approach to find the desired path and assignment is the following: iterate all the VNF assignments and find the corresponding desired path. Consider a simple example where the service chain only has one VNF, and the desired path length is large enough so that the path can be arbitrarily long. Moreover, assume that only one VM v in the current network hosts the VNF requested by the demand, and that the desired reuse factor is one. The proof of Theorem 1 actually implies that finding a feasible path visiting v for the demand is NP-hard. The intuition is that the path P_1 connecting the source to v and the path P_2

connecting v to the destination may use the same link. Hence, we cannot find P_1 and P_2 separately because the resulting flow may exceed some link capacity. Due to the hardness of the problem, to the best of our knowledge, we still need to iterate all possible paths to realize this approach.

VI. Numerical Results

We conduct simulations to evaluate the performance of our design in three data center networks, Fat-tree [49], VL2 [50], and BCube [51], and one inter-data-center network, the network of Cogent, which is a real tier 1 Internet service provider [52]. For each of the three data center networks, we generate 128 servers, each of which is connected to some edge switches. The number of switches may vary in different topologies of data centers. Each server and switch is represented as a vertex in the graph G, and we set the link capacity c_e to 1 Gbps for each link in G. The vertex capability c_v is set to 4 if v is a server, while is set to 0 if v is a switch. Cogent's network topology contains almost 190 access nodes with about 260 links and 40 Cogent data centers. Each access node and data center is represented by a vertex in the graph G. The capacity of each link is set to 10 Gbps, and $c_v = 100$ if v is a data center and $c_v = 0$, otherwise.

We assume that the capability of each VM is limited by its CPU power. The CPU of each VM is capable of executing 10^{10} instructions per second, i.e., 10^4 MIPS [53]. To generate the service chain of each demand, we first generate a set of VNFs F, which contains 30 distinct VNFs. We define the computational load of running a VNF $f \in F$ with an input traffic rate R as $L(f, R)$ (IPS). For each $f \in F$, we set $L(f, R) = 10R$ and $L(f, R) = 10R\ln(R)$ with probabilities 90% and 10%, respectively. Then, $L(s_{i,j})$ is set to $L(f(s_{i,j}), R_i)$. Each service chain $\mathcal{S}_i$ is constructed by sequentially and uniformly randomly selecting VNFs from F. The length of a service chain is uniformly randomly chosen from 1 to 8. The rate of each demand R_i follows the power law distribution [54], whose probability density function is

$$P[X = x] = \begin{cases} Cx^{-\alpha} & x \geq x_{min} \\ 0 & x < x_{min}, \end{cases}$$

where $C = (\alpha - 1)x_{min}^{\alpha-1}$, and x_{min} is the minimum demand size. Like [54], we set $\alpha = 2.1$. Unless stated otherwise, we use power law to generate 2000 demands with $x_{min} = 10$ Mbps and 4000 demands with $x_{min} = 100$ Mbps for the three data center center networks and the network of Cogent, respectively. The source and destination of each demand are chosen uniformly randomly from the vertices in the network.

We compare our design with the following heuristics.

- **Shortest Path Heuristic (SPH)**: It deploys a demand along the shortest path between the source and the destination π_{o_i,t_i}, and uses our dynamic programming, i.e., Eq. (6), to solve SVP$(\pi_{o_i,t_i}, \mathcal{S}_i)$ and assign services to VMs on π_{o_i,t_i}.
- **Greedy on VM Reuse (GVR)**: It extends the routing path of d_i iteratively on G' and assigns each service $s_{i,j} \in \mathcal{S}$ in order. $s_{i,j}$ is assigned to a reusable VM if possible; otherwise, it is assigned to an empty VM. Moreover, the selected reusable (empty) VM is the one closest to the VM serving its previous service, i.e., $m(s_{i,j-1})$. Then, the final path is $(o_i, v(m(s_{i,1})), v(m(s_{i,2})), \cdots, v(m(s_{i,|\mathcal{S}_i|})), t_i)$.
- **Shortened Greedy Heuristic (SGH)**: The idea of SGH is similar to GVR, but SGH might use a shorter path. It also extends the routing path iteratively from the source to the destination. In the k^{th} iteration, it finds the first unserved service that can be served by a reusable VM, say $s_{i,j}$, and assigns it to the closest reusable VM m. Let v_k denote the intermediate server that hosts m selected in iteration k, i.e., $v_k = v(m)$ and $v_0 = o_i$. Then, we find the shortest sub-path π_{v_{k-1},v_k} on G', and assign each of the unserved services before $s_{i,j}$, in order, to the first empty VM on π_{v_{k-1},v_k}.

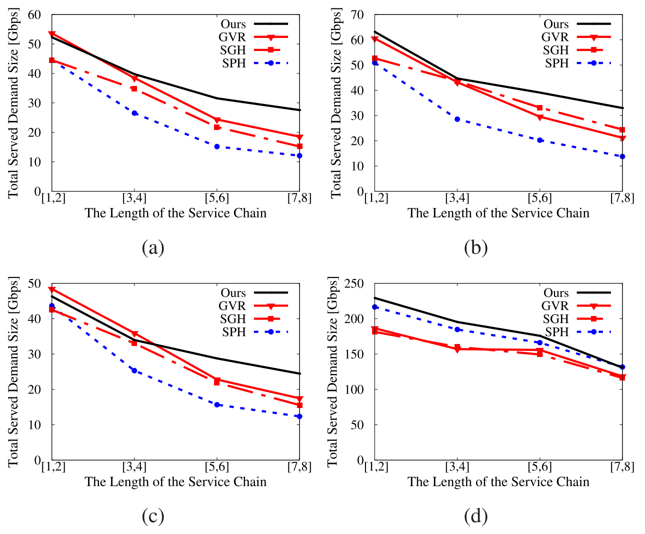


Fig. 2. Impact of the length of service chains on (a) Fat-Tree, (b) BCube, (c) VL2, and (d) the network of Cogent.

For all the comparison schemes, we also deploy demands sequentially, and sort the demands in the descending order of $n_i^* R_i$. When the problem is to maximize the total utility of admitted demands, we sort the demands in the descending order of $n_i^* U_i$, where U_i is the utility associated with demand d_i. A demand is rejected if the scheme cannot output a feasible solution for it. Like our solution, whenever the deployment is updated, these schemes also update the residual graph G' accordingly. For each simulation, we output the average result of 10 different instances.

Impact of the Length of Service Chains: We first check how the comparison schemes adapt to different lengths of service chains. We test four ranges of lengths, $[1, 2]$, $[3, 4]$, $[5, 6]$, and $[7, 8]$. Specifically, we consider four types of demands $\mathcal{D}$, and all the demands d_i in the n^{th} type of $\mathcal{D}$ have $|\mathcal{S}_i| \in [2n - 1, 2n]$. The results are shown in Fig. 2. Since a longer chain consumes more resources, the total size of accepted demands decreases as the length of the service chains increases. SPH performs worst in the three data center networks. This is because, in this simulation, the total rate of demands is relatively small, as compared to the link capacity, and, thus, the VM usage is more critical than the link usage. In the

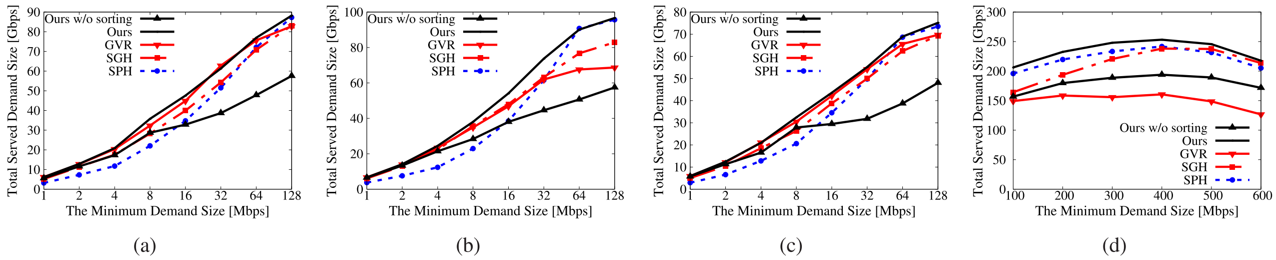


Fig. 3. Impact of the minimum demand size on (a) Fat-Tree, (b) BCube, (c) VL2, and (d) the network of Cogent.

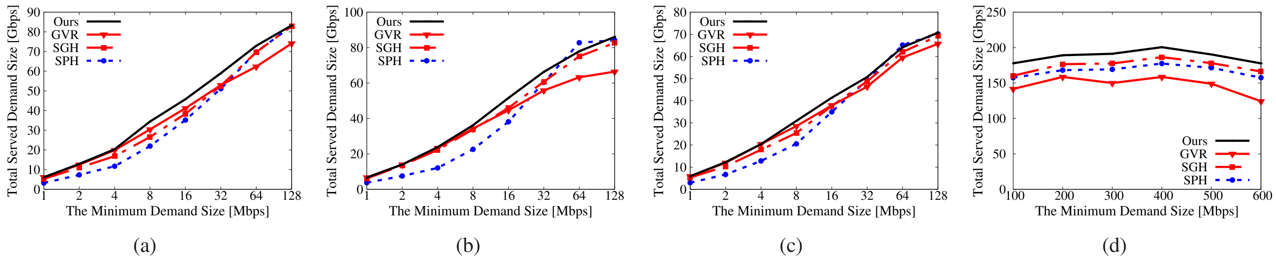


Fig. 4. Impact of the minimum demand size on (a) Fat-Tree, (b) BCube, (c) VL2, and (d) the network of Cogent. All methods sort the demands in the descending order of R_i.

network of Cogent, because all the demands are large, SPH performs well by taking the shortest path to utilize the precious communication resource efficiently. Our solution outperforms all the heuristics, especially for the more challenging cases, i.e., demands with longer service chains. The gain comes from our ability of striking a balance between the reuse factor and the path length derived from ST. Also, we use dynamic programming to maximize the reuse factor of a path, and most efficiently utilize the VMs given the guide of link usage.

Impact of the Minimum Demand Size: We further check the performance of the comparison schemes as the minimum demand size, x_{min}, varies from 1 Mbps to 128 Mbps and from 100 Mbps to 600 Mbps for the three data center networks and the network of Cogent, respectively. Fig. 3 plots the total size of the served demands. For the three data center networks, when the minimum demand size is small, SPH has the worst performance due to a reason similar to that in Fig. 2. However, when the minimum demand size is large, SPH outperforms other heuristics. This is because link resources become more important than VM resources in such a setting. This also explains the result on the network of Cogent. These results show that SPH is more suitable to deal with bandwidth shortage, while GVR is more resilient to VM shortage. Note that SGH, which also tries to reuse as many VMs as possible, outperforms GVR when the minimum demand size is large. This is because SGH connects reusable VMs by shortest paths. Our design elastically tunes the LV relation according to network topologies and demand properties, and, hence, outperforms other heuristics in a diverse range of scenarios.

Impact of the Order of Demands: In Fig. 4, all methods sort the demands in the descending order of R_i. In Fig. 3, we also show the performance of our solution without sorting demands. Two things are noteworthy. First, when most demand sizes are small, the order of demands has a very small impact on the performance. This is because under this setting, we can easily serve lots of demands regardless of the order of demands. However, when most demand sizes are large (comparing to the link capacity), the order of demands plays a crucial role in the performance. This is because, among all demands that have larger sizes, we should first serve demands that require less resource (e.g., demands with shorter paths). Our sorting strategy leverages the above idea and thus outperforms the other sorting strategies as the minimum demand size increases.

Performance of the Online Version of JVP: In this simulation, we consider a scenario where demands arrive at different times and we do not know all the demands in advance. In particular, we cannot sort the demands, and we need to process each demand immediately once it arrives. In addition, we cannot adjust the path length guide by the stretch factor. In this setting, because the number of demands that will arrive in the future is unknown, one might want to relax the usage guide and keep accepting demands until we run out of resource. Hence, we also consider a variation of our solution where we set the path length guide to infinity (referred to as "Ours (Unbounded)" in Fig. 5). We set x_{min} to 10 Mbps for data center networks and set x_{min} to 400 Mbps for the network of Cogent. The result is shown in Fig. 5. When the accumulated number of arrived demands is small, the above variation performs slightly better than our solution. However, because the variation consumes the communication resource aggressively, our solution outperforms the variation when the communication resource should be allocated conservatively, e.g., in the network of Cogent.

Effectiveness of Stress Testing: Next, we evaluate the effectiveness of ST, i.e., stress testing, and check whether the resulting LV relation guide can lead to efficient chain deployment. In particular, we apply our usage-guided deployment

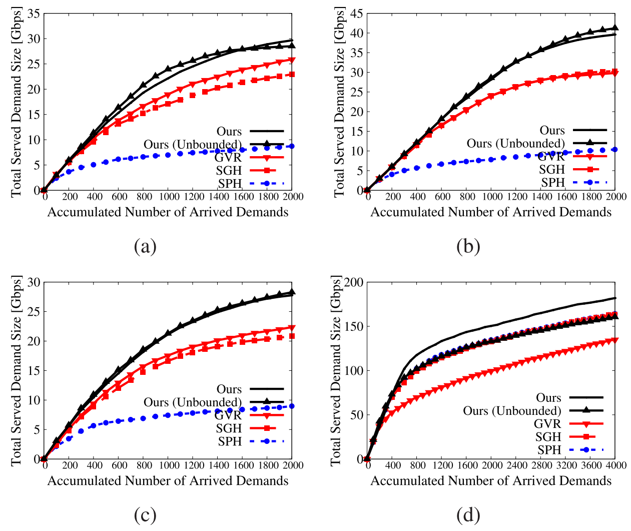


Fig. 5. Performance of the online version of JVP on (a) Fat-Tree, (b) BCube, (c) VL2, and (d) the network of Cogent.

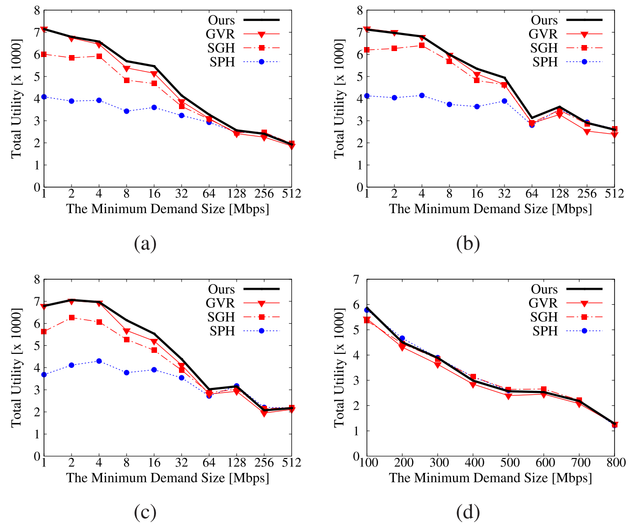


Fig. 7. Total utility on (a) Fat-Tree, (b) BCube, (c) VL2, and (d) the network of Cogent.

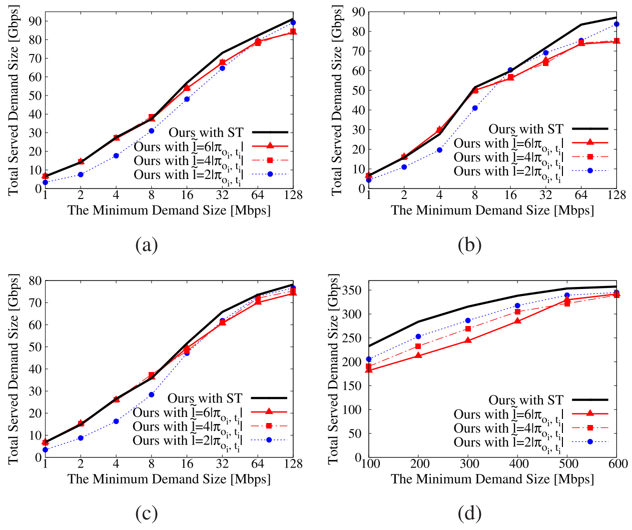


Fig. 6. Effectiveness of stress testing on (a) Fat-Tree, (b) BCube, (c) VL2, and (d) the network of Cogent.

algorithm (the one described in Section V), but compare the results of following the link usage $l(x^*)$ obtained by stress testing with those of following the link usage $\tilde{l} = k|\pi_{o_i,t_i}|$, $k = 2, 4$ and 6. The VM usage is fixed to x^* outputted by ST. We vary the minimum demand size, x_{min}, from 1 Mbps to 128 Mbps and from 100 Mbps to 600 Mbps for the three data center networks and the network of Cogent, respectively.

Fig. 6 shows the results. For the data center networks, when the minimum demand size is small, all the demands combined cannot fully utilize the link capacity of the network. Hence, we should allow longer paths so as to approach the given reuse factor. As a result, following a limited link usage $2|\pi_{o_i,t_i}|$ produces the worst performance. On the other hand, for the network of Cogent, the link capacity of the network is insufficient to serve all the demands, even when the minimum demand size is small. In this case, we should avoid long paths. This explains why the guidance of a long link usage leads to a worse performance in this case. The results also verify that no single link usage is always appropriate for different network conditions. However, following link usage specified by ST produces a higher performance than following other usage in most of the cases. This means that ST can effectively adapt the relation between link and VM usage to network dynamics, and guide the deployment algorithm to better utilize the limited resources to support more demands.

Simulation Results for the Problem of Maximizing the Total Utility of Admitted Demands: Fig. 7 shows the result when the goal is to maximize the total utility of admitted demands. In this simulation, each demand d_i is associated with a utility U_i generated by the aforementioned power law distribution, where $x_{min} = 1$. Similar to the previous result, when the minimum demand size is small, SPH has the worst performance. Moreover, the gap between the performances of SPH and other methods is larger in this simulation. One of the reasons is that, in previous simulations, influenced by the sorting strategy obtained by stress testing, all methods rarely process large demands after the early stage of deployment (e.g., after 20% of the input demands are processed). However, in this simulation, because large demands typically have small n_i^*, all methods process small demands first (unless some large demands have very high utilities). Thus, after the early stage of deployment, unprocessed demands may be too large for SPH to find reusable VMs with sufficient resource (since SPH only searches on the shortest path). However, other solution can extend the path to reach reusable VMs. Note that, when the minimum demand size is large, in order to increase the total utility, demands should take short paths to admit more demands. Thus, SPH performs well when the minimum demand size is large. Again, no matter what the minimum demand size is, our solution matches the best performance of the three comparison schemes.

The Ratio of the Optimum to Our Solution: In this simulation, we explore the gap between our solution and the optimum. The setting is identical to that of the simulation shown in Fig. 3. Due to the hardness of our problem, we

Min. Demand \ Network Type	Fat-Tree	BCube	VL2
1	1.194959	1.145433	1.158267
2	1.282629	1.16368	1.310137
4	1.628702	1.309031	1.65367
8	1.790059	1.675944	2.015414
16	2.449322	2.179217	2.728224
32	2.267371	2.438177	2.578631
64	1.922136	2.399172	2.212856
128	1.807852	2.381795	2.209054

Min. Demand \ Network Type	The Network of Cogent
100	3.570315
200	3.945012
300	4.165924
400	4.420268
500	4.878853
600	5.772593

Fig. 8. The ratio of the optimum to our solution.

compute an upper bound of the optimum. For each problem instance, we construct a corresponding knapsack instance. Given a capacity and a set of objects S, where each object is associated with a weight and a utility, the knapsack problem asks for a subset of S whose total weight is no more than the capacity and the total utility is maximized. Given an instance of our problem, the capacity in the corresponding knapsack instance is the total link capacity of the system, C_l. For each demand d_i, we generate an object J_i in the corresponding knapsack instance. Specifically, the utility of J_i is R_i, and the weight of J_i is $R_i|\pi_{o_i,t_i}|$, where $|\pi_{o_i,t_i}|$ is the shortest path length between the source and the destination of d_i. Obviously, if a set of demands can be admitted in our problem instance, then the set of the corresponding objects can be chosen in the corresponding knapsack instance. Thus, the optimum of the corresponding knapsack instance, OPT_{KP}, is an upper bound of the optimum of our problem instance, OPT_{JVP}. Because the knapsack problem is NP-hard, we calculate the optimum of the corresponding fractional knapsack problem, OPT_{FKP}. In the fractional knapsack problem, an object can be chosen partially (e.g., if a half of an object with (utility, weight) $= (200, 100)$ is chosen, then the total utility and total weight are increased by 100 and 50, respectively). Thus, OPT_{FKP} is an upper bound of OPT_{KP} (and thus an upper bound of OPT_{JVP}), and can be computed efficiently.

A common heuristic for the knapsack problem is to sort the objects in the descending order of the ratio of the utility to the weight (e.g., $\frac{R_i}{R_i|\pi_{o_i,t_i}|} = \frac{1}{|\pi_{o_i,t_i}|}$ in the constructed instance). The heuristic then considers each object sequentially and chooses an object if the remaining capacity is sufficient. Recall that we sort the demands in the descending order of $n_i^* R_i$. If we only consider the constraint imposed by the total link capacity in ST, then n_i^* is equal to $\frac{C_l}{|\pi_{o_i,t_i}|R_i}$. Therefore, for every demand d_i, $n_i^* R_i = C_l/|\pi_{o_i,t_i}|$. As a result, our sorting strategy degenerates to that of the common heuristic.

Fig. 8 shows the ratio of the upper bound to our performance. When the minimum demand size is small, our performance is very close to the optimum. The reason is that the demand size follows the power law distribution. Therefore, most demands have small sizes. Thus, we can gradually fill up the remaining capacity. Moreover, our solution sorts the demands in a reasonable way as the common heuristic does. The ratio is bigger in the network of Cogent. One of the reasons is that, OPT_{FKP} is not a very tight upper bound in the network of Cogent. In the three data center networks, the networks are highly connected, and thus every link can be accessed easily from all nodes within a few hops. Therefore, the overall link resource in the system can be captured properly by the total link capacity. However, this is not the case in the network of Cogent since it is a cross-continental inter-data-center network where links are not distributed regularly. In other words, some links in the network of Cogent cannot be accessed easily from some distant nodes. Thus, it is over-optimistic to assume that all demands can access all the links (as it is assumed implicitly in the constructed knapsack instance). We remark that, due to Theorem 1, under the assumption that $P \neq NP$, for every polynomial-time algorithm of our problem, there exist instances such that the algorithm cannot admit any demand but the optimum can admit at least one demand. Hence, the worst-case ratio for every polynomial-time algorithm is infinity.

VII. Discussion

In this work, we assume that the demand size remains the same after being served by a VNF. This might not always be true because some VNFs, e.g., firewall, might drop some packets or reshape the traffic. Our proposed solution can be generalized to deal with this traffic reshaping. The high-level idea of generalization is to consider the actual bandwidth consumption of each demand $B(x)$, instead of just the required path length $l(x)$. Specifically, in ST, $R_i l(x)$ in Eq. (1a) can be replaced with the total bandwidth consumption of each served demand $B(x)$, which can be estimated using a similar way mentioned in Section IV-B. Also, the calculation of the stretch factor should be changed accordingly. Then, we can apply our deployment algorithm to again find a solution whose resource consumption approaches the given bandwidth usage $B(x^*)$. The major modification we need to make is that, when combining $m^*(p-(v_{|p|}), \mathcal{S}'_h)$ and $m^*((v_{|p|}), \mathcal{S}''_h)$ in dynamic programming, we should further check whether the capacity of link $(v_{|p|-1}, v_{|p|})$ is sufficient to support the demand after invoking $s_{i,h}$. If not, the combined solution is infeasible.

So far, we assume that each NF is a VM. In practice, NFs can run inside containers, which are more lightweight than VMs [55], [56]. Note that, to achieve better performance, an NF should have dedicated resource [45], [46]. For example, it is assumed in [33] that NFs do not share CPU cores to avoid context switching overheads. (One exception is [56], since they focus on short-lived lightweight NFs.) In our model, an NF running as a container with dedicated resource is equivalent to an NF running as a VM. Moreover, even if the system is not partitioned into VMs in advance, we can treat each CPU as a VM in our problem model and then apply our solution. This method can also be used to solve the problem where NFs may share a VM provided that they do not share CPU cores.

In practice, a demand can be partitioned into smaller ones, and thus the network can be better utilized. We can model a partition of a demand d as a set of small demands $\mathcal{P} = \{\hat{d}_1, \hat{d}_2, \cdots, \hat{d}_k\}$ such that if all demands in $\mathcal{P}$ are served, then d is served. We call a demand in $\mathcal{P}$ a *slice* of demand d.

To adapt our solution to demand partition, we need more inputs. First, because no universal way can be used to partition all types of traffics and VNFs, each input demand must be associated with its valid partition. If there are multiple ways to partition the demand, only the most fine-grained partition has to be given, i.e., the partition composed of the smallest slices. We can further merge different slices after resource allocation when applicable (e.g., the resources of different slices are from the same set of devices). Second, because we consider admission control, an additional input must be associated with each demand to indicate whether or not the demand can be partially admitted, i.e., only some slices are admitted. Given the above two additional inputs, we can then generalize our solution as follows: If a demand can be partially admitted, then we simply treat each slice as a demand in our original problem. If a demand d cannot be partially admitted, then we apply stress testing on d (not on the slices) to get the LV relation guide. We also sort the set of the original demands (not the set of slices). When we process a demand d, we then apply our DP-based deployment algorithm on each of its slices. Specifically, the LV relation guide for each slice is that of d. Therefore, the total link consumption guide of d is equal to the total link consumption guide of all slices. In addition, if the first slice can reuse x VMs, then it is easy for the next slice to reuse x VMs. If some slice is rejected (due to an overlong path), then all slices are rejected. Finally, note that it would be inefficient if we partition a demand into lots of tiny slices, since the overhead of partitioning and merging demands would be high. Therefore, we should avoid partitions that result in lots of arbitrarily small slices. As a result, the JVP problem is still NP-hard, even if demands can be partitioned (since finding a feasible solution for a slice is NP-hard).

VIII. Conclusion

In this paper, we study the joint VNF placement and path selection problem. We show that this problem is NP-hard. To tackle this problem, we study the relation between the path length and the VM reuse factor, i.e., the LV relation. Specifically, we use the idea of stress testing to find a LV relation guide. Following this guidance, we then propose a chain deployment algorithm to find a solution whose path length and reuse factor approximately meet the guide. Via simulation, we show that our design outperforms other heuristics, which cannot manage the relation between link and server usage well across different types of networks. In contrast, our usage-guided deployment elastically adapts link and VM usage to different networks, and, hence, better utilizes the limited resources to serve a larger size of demands.

Two directions are worth studying in the future. First, the online version of our problem where the number of demands that will arrive in the future is unknown. To better reflect the reality, in the online problem, one can also consider the completion time of a demand or the trade-off between the total size of admitted demands and the elapsed time since the first demand arrives. Second, in the current solution, we do not leverage the relation between demands. In practice, the distributions of some properties of demands may be biased. For example, the distribution of the source and the destination or the distribution of NFs in a service chain may be biased. If some distribution is biased, one may design better solutions by leveraging the bias. For example, one can divide the demands into groups where similar demands belong to the same group (clustering), and then considers the resource allocation of demands in the same group jointly.

Appendix

For ease of presentation, we set $\mathcal{S}'_h = \mathcal{S}$ when $h = |\mathcal{S}|$ and set $\mathcal{S}''_h = \mathcal{S}$ when $h = 0$. Consider $|\mathcal{S}| + 1$ problems, SVP$_h(p, \mathcal{S})$, $0 \le h \le |\mathcal{S}|$, each has one more constraint than that of SVP$(p, \mathcal{S})$. The additional constraint is that the services assigned to VMs on $p-(v_{|p|})$ are fixed to $\mathcal{S}'_h$ (if $1 \le h \le |\mathcal{S}|$) or $\varnothing$ (if h=0). If SVP$_h(p, \mathcal{S})$ has a feasible solution, then both SVP$(p-(v_{|p|}), \mathcal{S}'_h)$ (if $h > 0$) and SVP$((v_{|p|}), \mathcal{S}''_h)$ (if $h < |\mathcal{S}|$) have a feasible solution. Also note that the paths in these two problems do not overlap, and so do their service sub-chains. Thus, if $m^*(p-(v_{|p|}), \mathcal{S}'_h)$ and $m^*((v_{|p|}), \mathcal{S}''_h)$ are feasible, then m_h must be feasible for SVP$(p, \mathcal{S})$ according to Eq. (6). Combining the above arguments, we get that, if SVP$_h(p, \mathcal{S})$ has a feasible solution, then m_h is feasible for SVP$_h(p, \mathcal{S})$ and SVP$(p, \mathcal{S})$. On the other hand, if SVP$_h(p, \mathcal{S})$ has no feasible solution, then at least one of SVP$(p-(v_{|p|}), \mathcal{S}'_h)$ and SVP$((v_{|p|}), \mathcal{S}''_h)$ has no feasible solution and, thereby, m_h is not feasible for both SVP$_h(p, \mathcal{S})$ and SVP$(p, \mathcal{S})$.

We consider two cases of $m^*(p, \mathcal{S})$, feasible or infeasible. If $m^*(p, \mathcal{S})$ is infeasible, then all SVP$_h(p, \mathcal{S})$s, $0 \le h \le |\mathcal{S}|$, have no feasible solution. Hence, by the first part of the proof, m_h, $0 \le h \le |\mathcal{S}|$, is infeasible for SVP$(p, \mathcal{S})$. Thus, Eq. (6) holds for this case. For the case of $m^*(p, \mathcal{S})$ being feasible, since any feasible solution of SVP$(p, \mathcal{S})$ must be feasible for some SVP$_h(p, \mathcal{S})$, $0 \le h \le |\mathcal{S}|$, $m^*(p, \mathcal{S})$ must be an optimal solution for some SVP$_{h^*}(p, \mathcal{S})$. It is then sufficient to show that m_{h^*} is feasible and optimal for SVP$_{h^*}(p, \mathcal{S})$. The feasibility of m_{h^*} clearly holds by the first part of the proof. We prove the optimality of m_{h^*} by contradiction. Assume that a better solution m' exists. Then, compared with m_{h^*}, m' must reuse more VMs in $p-(v_{|p|})$ or in $(v_{|p|})$, which contradicts to the optimality of $m^*(p-(v_{|p|}), \mathcal{S}'_{h^*})$ and $m^*((v_{|p|}), \mathcal{S}''_{h^*})$.

References

[1] T.-W. Kuo, B.-H. Liou, K. C.-J. Lin, and M.-J. Tsai, "Deploying chains of virtual network functions: On the relation between link and server usage," in *Proc. IEEE Int. Conf. Comput. Commun. (INFOCOM)*, Apr. 2016, pp. 1–9.

[2] J. Sherry, S. Ratnasamy, and J. S. At, "A survey of enterprise middlebox deployments," Dept. Elect. Eng. Comput. Sci., Univ. of California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2012-24, 2012.

[3] (2012). *Network Functions Virtualisation—Introductory White Paper*. [Online]. Available: https://portal.etsi.org/nfv/nfv_white_paper.pdf

[4] D. Meyer. (2016). *AT&T and Verizon NFV and SDN Moves Seen as Industry-Leading*. Accessed: Jan. 16, 2018. [Online]. Available: https://www.rcrwireless.com/20160609/network-function-virtualization-nfv/att-verizon-nfv-sdn-moves-seen-industry-leading-tag2

[5] J. Martins *et al.*, "ClickOS and the art of network function virtualization," in *Proc. USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2014, pp. 459–473.

[6] P. Quinn and T. Nadeau, *Problem Statement for Service Function Chaining*, document IETF RFC 7498, Apr. 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc7498.txt, doi: 10.17487/RFC7498.

[7] F. Bari, S. R. Chowdhury, R. Ahmed, R. Boutaba, and O. C. M. B. Duarte, "Orchestrating virtualized network functions," *IEEE Trans. Netw. Service Manage.*, vol. 13, no. 4, pp. 725–739, Dec. 2016.

[8] *OpenStack*. Accessed: Jan. 16, 2018. [Online]. Available: https://www.openstack.org/

[9] *Open Source MANO*. Accessed: Jan. 16, 2018. [Online]. Available: https://osm.etsi.org/

[10] *OPNFV*. Accessed Jan. 16, 2018. [Online]. Available: https://www.opnfv.org/

[11] M. T. Beck and J. F. Botero, "Coordinated allocation of service function chains," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2015, pp. 1–6.

[12] S. Mehraghdam, M. Keller, and H. Karl, "Specifying and placing chains of virtual network functions," in *Proc. IEEE Int. Conf. Cloud Netw. (CloudNet)*, Oct. 2014, pp. 7–13.

[13] (2017). *Stress Testing—Wikipedia, the Free Encyclopedia*. Accessed: Jan. 11, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Stress_testing

[14] J. G. Herrera and J. F. Botero, "Resource allocation in NFV: A comprehensive survey," *IEEE Trans. Netw. Service Manage.*, vol. 13, no. 3, pp. 518–532, Sep. 2016.

[15] B. Addis, D. Belabed, M. Bouet, and S. Secci, "Virtual network functions placement and routing optimization," in *Proc. IEEE Int. Conf. Cloud Netw. (CloudNet)*, Oct. 2015, pp. 171–177.

[16] J. Elias, F. Martignon, S. Paris, and J. Wang, "Efficient orchestration mechanisms for congestion mitigation in nfv: Models and algorithms," *IEEE Trans. Services Comput.*, vol. 10, no. 4, pp. 534–546, Jul. 2015.

[17] A. Gupta, M. F. Habib, P. Chowdhury, M. Tornatore, and B. Mukherjee, "On service chaining using virtual network functions in network-enabled cloud systems," in *Proc. IEEE Int. Conf. Adv. Netw. Telecommun. Syst. (ANTS)*, Dec. 2015, pp. 1–3.

[18] P. Bellavista *et al.*, "Virtual network function embedding in real cloud environments," *Comput. Netw.*, vol. 93, no. 3, pp. 506–517, Dec. 2015.

[19] F. Wang, R. Ling, J. Zhu, and D. Li, "Bandwidth guaranteed virtual network function placement and scaling in datacenter networks," in *Proc. IEEE Int. Perform. Comput. Commun. Conf. (IPCCC)*, 2015.

[20] H. Moens and F. De Turck, "VNF-P: A model for efficient placement of virtualized network functions," in *Proc. Int. Conf. Netw. Service Manage. (CNSM)*, 2014, pp. 418–423.

[21] M. Xia, M. Shirazipour, Y. Zhang, H. Green, and A. Takacs, "Network function placement for NFV chaining in packet/optical datacenters," *J. Lightw. Technol.*, vol. 33, no. 8, pp. 1565–1570, Apr. 15, 2015.

[22] R. Cohen, L. Lewin-Eytan, J. Naor, and D. Raz, "Near optimal placement of virtual network functions," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2015, pp. 1346–1354.

[23] M. Bagaa, T. Taleb, and A. Ksentini, "Service-aware network function placement for efficient traffic handling in carrier cloud," in *Proc. IEEE Wireless Commun. Netw. Conf. (WCNC)*, Apr. 2014, pp. 2402–2407.

[24] A. Baumgartner, V. S. Reddy, and T. Bauschert, "Mobile core network virtualization: A model for combined virtual core network function placement and topology optimization," in *Proc. IEEE Conf. Netw. Softwarization (NetSoft)*, Apr. 2015, pp. 1–9.

[25] R. Mijumbi, J. Serrat, J. L. Gorricho, J. Rubio-Loyola, and S. Davy, "Server placement and assignment in virtualized radio access networks," in *Proc. Int. Conf. Netw. Service Manage. (CNSM)*, 2015, pp. 398–401.

[26] M. Bouet, J. Leguay, and V. Conan, "Cost-based placement of vDPI functions in NFV infrastructures," in *Proc. 1st IEEE Conf. Netw. Softwarization (NetSoft)*, Apr. 2015, pp. 1–9.

[27] A. Baumgartner, V. S. Reddy, and T. Bauschert, "Combined virtual mobile core network function placement and topology optimization with latency bounds," in *Proc. Eur. Workshop Softw. Defined Netw.*, 2015, pp. 97–102.

[28] T. Lin, Z. Zhou, M. Tornatore, and B. Mukherjee, "Demand-aware network function placement," *J. Lightw. Technol.*, vol. 34, no. 11, pp. 2590–2600, Jun. 1, 2016.

[29] I. Jang, S. Choo, M. Kim, S. Pack, and M. K. Shin, "Optimal network resource utilization in service function chaining," in *Proc. IEEE NetSoft Conf. Workshops (NetSoft)*, Jun. 2016, pp. 11–16.

[30] A. Gember *et al.* (May 2013). "Stratos: A network-aware orchestration layer for virtual middleboxes in clouds." [Online]. Available: https://arxiv.org/abs/1305.0209

[31] S. Palkar *et al.*, "E2: A framework for NFV applications," in *Proc. 25th Symp. Operat. Syst. Principles (SOSP)*, New York, NY, USA: ACM, 2015, pp. 121–136. [Online]. Available: http://doi.acm.org/10.1145/2815400.2815423

[32] Z. A. Qazi *et al.*, "SIMPLE-fying middlebox policy enforcement using SDN," in *Proc. ACM SIGCOMM Conf. (SIGCOMM)*, New York, NY, USA: ACM, 2013, pp. 27–38. [Online]. Available: http://doi.acm.org/10.1145/2486001.2486022

[33] W. Zhang *et al.*, "SDNFV: Flexible and dynamic software defined control of an application-and flow-aware data plane," in *Proc. 17th Int. Middleware Conf.*, New York, NY, USA: ACM, 2016, pp. 2-1–2-12. [Online]. Available: http://doi.acm.org/10.1145/2988336.2988338

[34] T. Lukovszki and S. Schmid, "Online admission control and embedding of service chains," in *Proc. Int. Colloquium Struct. Inf. Commun. Complexity (SIROCCO)*, 2015, pp. 104–118.

[35] R. Mijumbi *et al.*, "Design and evaluation of algorithms for mapping and scheduling of irtual network functions," in *Proc. IEEE Conf. Netw. Softwarization (NetSoft)*, Apr. 2015, pp. 1–9.

[36] M. C. Luizelli *et al.*, "Piecing together the nfv provisioning puzzle: Efficient placement and chaining of virtual network functions," in *Proc. IFIP/IEEE Int. Symp. Integr. Netw. Manage. (IM)*, May 2015, pp. 98–106.

[37] R. Riggio *et al.*, "Virtual network functions orchestration in wireless networks," in *Proc. Int. Conf. Netw. Service Manage. (CNSM)*, 2015, pp. 108–116.

[38] M. Ghaznavi *et al.*, "Elastic virtual network function placement," in *Proc. IEEE Int. Conf. Cloud Netw. (CloudNet)*, Oct. 2015, pp. 255–260.

[39] A. Mohammadkhan *et al.*, "Virtual function placement and traffic steering in flexible and dynamic software defined networks," in *Proc. IEEE Int. Workshop Local Metropolitan Area Netw.*, Apr. 2015, pp. 1–6.

[40] S. Sahhaf *et al.*, "Network service chaining with optimized network function embedding supporting service decompositions," *Comput. Netw.*, vol. 93, no. 3, pp. 492–505, 2015.

[41] R. Riggio, A. Bradai, D. Harutyunyan, T. Rasheed, and T. Ahmed, "Scheduling wireless virtual networks functions," *IEEE Trans. Netw. Service Manag.*, vol. 13, no. 2, pp. 240–252, Jun. 2016.

[42] M. Ghaznavi, N. Shahriar, R. Ahmed, and R. Boutaba. (Jan. 2016). "Service function chaining simplified." [Online]. Available: https://arxiv.org/abs/1601.00751

[43] X. Meng, V. Pappas, and L. Zhang, "Improving the scalability of data center networks with traffic-aware virtual machine placement," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Mar. 2010, pp. 1–9.

[44] *Google Cloud Platform*. Accessed: Jan. 11, 2018. [Online]. Available: https://cloud.google.com/

[45] M. Falkner, A. Leivadeas, I. Lambadaris, and G. Kesidis, "Performance analysis of virtualized network functions on virtualized systems architectures," in *Proc. IEEE CAMAD*, Mar. 2016, pp. 1–9.

[46] A. Leivadeas, M. Falkner, I. Lambadaris, and G. Kesidis, "Optimal virtualized network function allocation for an SDN enabled cloud," *Comput. Standard Interfaces*, vol. 54, no. 4, pp. 266–278, 2017.

[47] A. Bremler-Barr, Y. Harchol, and D. Hay, "OpenBox: A software-defined framework for developing, deploying, and managing network functions," in *Proc. ACM SIGCOMM Conf. (SIGCOMM)*, New York, NY, USA: ACM, 2016, pp. 511–524. [Online]. Available: http://doi.acm.org/10.1145/2934872.2934875

[48] M. Andrews *et al.*, "Inapproximability of edge-disjoint paths and low congestion routing on undirected graphs," *Combinatorica*, vol. 30, no. 5, pp. 485–520, 2010.

[49] R. N. Mysore *et al.*, "PortLand: A scalable fault-tolerant layer 2 data center network fabric," *ACM Special Interest Group Data Commun. (SIGCOMM)*, vol. 39, no. 4, pp. 39–50, 2009.

[50] A. Greenberg *et al.*, "VL2: A scalable and flexible data center network," *ACM Special Interest Group Data Commun.*, vol. 39, no. 4, pp. 51–62, 2009.

[51] C. Guo *et al.*, "BCube: A high performance, server-centric network architecture for modular data centers," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 63–74, 2009.

[52] *Cogent's Network Map*. Accessed: Jul. 24, 2015. [Online]. Available: http://cogentco.com/en/network/network-map

[53] D. A. Patterson and J. L. Hennessy, *Computer organization and Design: The Hardware/Software Interface*. San Francisco, CA, USA: Morgan Kauffman, 2013.

[54] X. Li and C. Qian, "Low-complexity multi-resource packet scheduling for network function virtualization," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, 2015, pp. 1400–1408.

[55] W. Zhang *et al.*, "OpenNetVM: A platform for high performance network service chains," in *Proc. Workshop Hot Topics Middleboxes Netw. Function Virtualization (HotMIddlebox)*, New York, NY, USA: ACM, 2016, pp. 26–31. [Online]. Available: http://doi.acm.org/2940147.2940155

[56] W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, and T. Wood, “Flurries: Countless fine-grained NFs for flexible per-flow customization,” in *Proc. 12th Int. Conf. Emerg. Netw. EXperim. Technol. (CoNEXT)*, New York, NY, USA: ACM, 2016, pp. 3–17. [Online]. Available: http://doi.acm.org/10.1145/2999572.2999602

Tung-Wei Kuo received the B.S. degree from National Chiao Tung University, Taiwan, in 2009, and the M.S. and Ph.D. degrees in computer science from National Tsing Hua University, Taiwan, in 2011 and 2015, respectively. He was a Post-Doctoral Fellow with the Research Center for Information Technology Innovation, Academia Sinica, Taiwan, from 2015 to 2016. He is currently an Assistant Professor with the Department of Computer Science, National Chengchi University, Taiwan. His current research interests include network design and blockchain technology.

Bang-Heng Liou received the B.S. degree in computer science from National Chi Nan University in 2013, and the M.S. degree in computer science from National Tsing Hua University in 2015. His research interests include network management and the design of approximation algorithms.

Kate Ching-Ju Lin (SM’16) received the B.S. degree from the Department of Computer Science, National Tsing Hua University, in 2003, and the Ph.D. degree from the Graduate Institute of Networking and Multimedia, National Taiwan University, in 2009. She was a Visiting Scholar with CSAIL, MIT, from 2007 to 2008 and from 2010 to 2011. She is currently an Associate Professor with the Department of Computer Science, National Chiao Tung University, Taiwan. Her current research interests include wireless systems, wireless multimedia networking, and visible light communications. She was a recipient of the K. T. Li Young Researcher Award from the ACM Taipei, Taiwan, in 2014, and the Research Project for Excellent Young Scholars from the Ministry of Science and Technology, Taiwan (2013–2020).

Ming-Jer Tsai (M’04) received the Ph.D. degree in electrical engineering from National Taiwan University in 1997. Since then, he has been with the Computer and Communication Laboratory, Industrial Technology Research Institute. He joined the Department of Computer Science, National Tsing Hua University, in 2003, and the Institute of Communications Engineering, National Tsing Hua University, in 2009, where he is currently a Professor. His research interests include distributed systems and mobile computing. He is a member of the ACM.