Fine-Grained Cloud Resource Provisioning for Virtual Network Function

Hui Yu, *Student Member, IEEE*, Jiahai Yang, *Member, IEEE*, and Carol Fung

仿真平台：KVM

两种方法：
提出了ElasticNFV，有两个功能：
业务资源模型与业务迁移方法

***Abstract*—The deployment of Virtualized Network Functions is expected to be dynamic and swift when using Network Function Virtualization technology. The dynamic nature of workload from users requires the resource allocation of underlying infrastructure to be flexible to cope with the changes. Existing works investigated elastic NFV solutions by dynamically creating and dismantling Virtual Machine (VM) replicas, while maintaining balanced workload among VMs. However, those solutions are coarse-grained which may cause unnecessary resource over-provisioning as different network functions consume different amount of resources. In this paper, we present ElasticNFV, a dynamic and fine-grained cloud resource provisioning solution for VNF. ElasticNFV takes real-time resource demand of multiple service chains and allocates resources through an elastic provision mechanism. When a scaling conflict occurs, ElasticNFV provides a two-phase minimal migration algorithm to optimize the migration time and embedding cost of VNF instances. We implement ElasticNFV on top of the KVM platform to provide elastic VM for each VNF instance and Open vSwitch to form elastic intra-cloud network with virtual links between VNF instances. Our evaluation results show that ElasticNFV can improve VNF performance significantly, and achieve high resource utilization and fast migration time with low cost.**

***Index Terms*—Middlebox, network function virtualization (NFV), resource allocation, service chain, cloud computing, resource scaling.**

I. Introduction

NETWORK appliances or "middleboxes" are ubiquitous in modern networks and perform a variety of important functions such as packet forwarding. Studies [1], [2] have shown that enterprises rely heavily on middleboxes to improve the network performance (e.g., WAN Optimizer, load balancers), and to enhance the security (e.g., firewalling, IDS/IPS) by monitoring traffic (e.g., passive network monitor). However, large-scale deployment of middleboxes faces many challenges, including high expenditure on dedicated hardware and complex management. To remedy these issues, Network Function Virtualization (NFV) [3] has been proposed to decouple Network Functions (NFs) from dedicated hardware by running NFs as software on standard commodity servers. The paradigm has been embraced by both academia and industry rapidly with over 270 individual companies and a number of emerging products by 2015 [4].

As a cost-effective way of testing and implementation, most NFV proof of concepts and early implementations were based on deploying NFs on Virtual Machines (VMs) in the clouds [5], [6]. It is increasingly common for Telecommunications Service Providers (TSPs), enterprises and organizations to outsource network processing such as IDS/IPS to the cloud, so the benefits of cloud computing can be brought to middlebox infrastructure [7]. The NFs executed in VMs are called *software middleboxes* or *Virtualized Network Functions* (VNFs). However, the VNF workload often changes frequently which requires the underlying infrastructure to be dynamic and agile to cope with the changes. At the same time, different VNFs consume different amount of resource [8]. For example, IDS/IPS are often CPU-bound, HTTP caching consumes memory the most, and the bottleneck of flow monitoring is the bandwidth. It is a major challenge to perform flexible and efficient VNF resource allocation on physical machines (PMs).

To address the aforementioned issue, several studies [9], [10] have been performed to provide NFV elastic solutions by creating and destroying VM replicas (VNF instances), while balancing the workload among VMs. This is called *horizontal scaling*. However, those solutions are coarse-grained which can cause unnecessary resource over-provisioning which leads to reduced resource utilization. It also causes Service Level Objective (SLO) violations which leads to reduced VNF performance. Furthermore, interruptions often occur during flow migrations. To avoid these problems, an elastic multi-resource scaling mechanism is needed that can adjust the resource cap dynamically based on VNFs resource demands.

In this paper, we propose *ElasticNFV*, a dynamic solution that achieves fine-grained cloud resource provisioning for VNFs. The goal of ElasticNFV is an automatic cloud-based NFV system that meets the SLO requirements of the VNFs running inside the cloud with minimum resource cost. To design a resource scaling system to meet the above goal, we need to address two problems. First, network traffic usually needs to pass through several VNFs in a particular order. This is called *network service chaining*. To form a proper service chain, it is necessary to dynamically reconfigure the

Manuscript received May 31, 2019; revised November 9, 2019 and March 7, 2020; accepted April 1, 2020. Date of publication April 8, 2020; date of current version September 9, 2020. This work was supported by the National Natural Science Foundation of China under Grant 61432009 and Grant 61462009. An earlier version of this paper was presented at the International Conference on IEEE GLOBECOM 2018 [20]. The associate editor coordinating the review of this article and approving it for publication was S. Latre. *(Corresponding author: Jiahai Yang.)*

Hui Yu and Jiahai Yang are with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China, and also with the Beijing National Research Center for Information Science and Technology, Tsinghua University, Beijing 100084, China (e-mail: yuhui7red@outlook.com; yang@cernet.edu.cn).

Carol Fung is with the Department of Computer Science, Virginia Commonwealth University, Richmond, VA 23284 USA (e-mail: cfung@vcu.edu).

Digital Object Identifier 10.1109/TNSM.2020.2986223

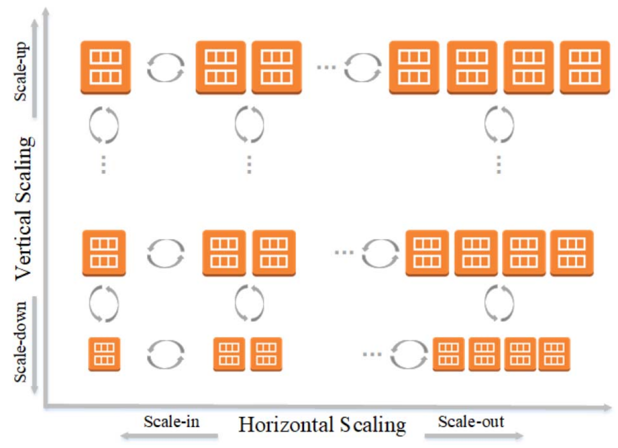


Fig. 1. Horizontal scaling and vertical scaling.

CPU and memory resources within a VNF instance, as well as intra-cloud network resource among VNF instances simultaneously [11]. Note that an intra-cloud management on a shared network infrastructure is an inherently complex and challenging task. Second, co-located VNF instances conflict when the available resources are insufficient to accommodate all scale-ups. Therefore, an efficient VNF migration algorithm with fast migration time and low embedding cost is called upon.

ElasticNFV provides resource elastic provisioning and solves scaling conflicts based on two modules, and both run on top of KVM virtualization platform and Open vSwitch. The *Resource Elastic Provisioning* (REP) module proposes a Dynamic Multi-Resource (DMR) model to represent dynamic resource demand of service chains. It also provides elastic VM for each VNF instance and elastic intra-cloud network for a group of VNF instances connected through virtual links. The elastic VM approach of REP module is based on the KVM hypervisor [12] which employs *CPU hotplug* and *memory ballooning* techniques. Through these two techniques, a guest VM can change the amount of *vCPU* and memory in use at runtime, that we call *vertical scaling* [13], [14]. Our elastic intra-cloud network is based on Open vSwitch [15] which can achieve dynamical bandwidth guarantee for virtual links between VNF instances.

Fig. 1 illustrates the comparison between the horizontal scaling and the vertical scaling. The horizontal scaling provides elasticity by creating new VNF instances (new virtual machines or containers) to meet SLO requirements (Scale-out), or destroying existing VNF instances to improve the utility of resource (Scale-in), while balancing the workload across VNF instances. Instead of adding or removing VNF instances, the vertical scaling increases or decreases their instances' capacity onsite. As the flow volume increases, it can directly provide more resources (CPU, memory or bandwidth) to VNF instances (Scale-up), or reduce the caps of the over-provisioned resources (Scale-down). The vertical scaling provides live resizing capability on VNF instances. Compared to the horizontal scaling, the vertical scaling provides a more efficient resource utilization (finer-grained cloud resource allocation), shorter scaling periods (in milliseconds) and faster response time [16].

When a PM cannot satisfy the scale-up requests from VNF instances, the *scaling conflict handling* (SCH) module migrates some VNF instances out of the overloaded machine. VM migration is often disruptive and migrating VMs upon each request is not a plausible solution. There are some approaches that support a weak form of reconfiguration. For example, Tetris [17] allows users to update vCPU numbers at runtime and ElasticSwitch [18] adjusts rate limiters according to the actual network applications' requirements. However, the efficacy of these solutions is limited due to the resource capacity on the machine. Migration is necessary under some circumstances. There are already some studies, such as [19], focusing on migration algorithms on cloud platform. However, their algorithms are based on the standard Virtual Cluster (VC) model which cannot capture the bandwidth demand of VNF instances. In this paper, the SCH module provides a Two-phase Minimal Migration (TPMM) algorithm to allow a system to dynamically reconfigure CPU, memory and network resources for VNFs and perform migration only if necessary to minimize the cost.

The contributions of this paper can be summarized as follows, building on our prior work [20]. Compared to the preliminary version, we have made important updates for the related work, implementation, evaluation and discussion in this paper.

- We propose ElasticNFV, a hypervisor-based framework to achieve dynamic fine-grained cloud resource provisioning for VNFs.
- We introduce a DMR model to represent the dynamic resource demand of service chains. We use three types of resource elastic provisioning mechanisms to adjust resource cap.
- We design a TPMM algorithm to optimize the migration time and embedding cost when a PM cannot satisfy the scale-up requests from VNFs.
- We implement ElasticNFV on top of the KVM hypervisor and the Open vSwitch bridge to realize elastic VMs and elastic intra-cloud networks.
- We demonstrate that ElasticNFV performs better than existing coarse-grained solutions through extensive evaluations.

The rest of this paper is organized as follows. Section II retrospects related works and compares our work with them. Section III lays out the system overview. Sections IV and V present the REP module and the SCH module, respectively. Section VI demonstrates experimental results. Section VII discusses several possible extensions of ElasticNFV. Finally we conclude the work in Section VIII.

II. Related Work

ElasticNFV enriches the literature on elastic framework for NFV, elastic cloud platform and VM migration. In this section, we briefly overview the works most relevant to ElasticNFV in three main technical fields.

A. Elastic NFV

CoMb [21] utilizes middlebox structure to consolidate heterogeneous middlebox applications onto commodity hardware.

However, it does not address the issue of scaling, parallelism, or elasticity. The NIDS Cluster [22] is a clustered version of Bro [23] that is capable of performing coordinated analysis of traffic, at large scale. By exposing policy layer state and events as serializable state, individual nodes are able to obtain a global view of the system state. But the NIDS Cluster cannot scale dynamically and statefully, as it lacks the ability to migrate lower layer flow state and their associated network flows across replicas. Probius [24] is an automated analysis system which extracts performance features of VNFs and their service chains on the NFV architecture and finally detects the root causes of performance uncertainties. However, this analysis system does not discuss horizontal scaling and vertical scaling for NFV performance.

Split/Merge [9] proposes a system named FreeFlow, to explore techniques that allow the control of VNF state so that VNFs can split or merge during elastic execution. Using FreeFlow, VNFs identify partitioned states, which can be split among replicas or merged together into a single replica. Hinojosa *et al.* [25] proposed an automated network service scaling in NFV with a NS descriptor (NSD), different scaling procedures and an ETSI-compliant workflow. Stratos [26] considered VNF composition, the issue that triggers scaling, and managed the network interactions of the VNFs during and after scaling. Split/Merge and Stratos are complimentary to each other. E2 [10] provides a single coherent system to manage NFs, while relieving developers from developing per-NF solutions for placement, scaling, fault-tolerance, and other functionality. OFM [27] realizes NFV elasticity control through effective flow migration, which is implemented on top of NFV and SDN environment to address NF scaling out, scaling in, and load balancing for different control goals and challenges. NFV-PEAR [28] is a framework for optimized VNF placement and chaining that (re)arranges previously determined VNF placement and service chaining periodically with the goal of end-to-end flow performance. However, all works mentioned above achieve NFV elasticity by horizontal scaling which are coarse-grained and may lead to unnecessary resource over-provisioning.

Erama *et al.* [29] develops an approach that uses three algorithms to address the resource consolidation problem when placing VNF instances. They balance the tradeoff between the power consumption and QoS degradation to determine whether a migration is appropriate. They adopt VNF vertical scaling but cannot avoid the service interruption. Edmonds *et al.* [30] proposes an open cloud computing interface compliant (OCCI-compliant) and NFV-ready framework for fine-grained resource-aware management in mobile cloud networking, which is developed within a large EU FP7 project. ENVI [31] uses a combination of VNF-level features and infrastructure-level features to construct a machine-learning-based decision engine for detecting resource flexing event. However, the resource flexing engine performs a coarse-grained search over a limited set of VM sizing options, which also leads to resource over-provisioning. Deep neural network models may also suffer from lack of training data and high training overhead.

In our previous work [20], we proposed ElasticNFV, which is the basis of this paper. ElasticNFV provides elastic VMs for single VNF and elastic intra-cloud network for virtual links among VNFs to achieve dynamic fine-grained provisioning, which increases the resource utilization; it leverages live migration technology to avoid the service interruption and minimizes the migration time of the VNF instances. In this paper, we have made important updates for the implementation of elastic VM and elastic intra-cloud networks for virtual link. In addition, we have substantially strengthened the performance evaluation section. Furthermore, we also enriched the literature survey in related work section and have discussed serval possible extensions of the previous system design.

B. Elastic Cloud

Many recent studies have addressed the problem of dynamically adjusting the number of VMs assigned to a cloud application to keep up with load changes. AGILE [32] uses wavelets to provide medium-term performance predictions; it provides an automatically-determined model of how an application's performance relates to the resources it has available. It also implements a way of cloning VMs that minimizes application startup time. Embrane [33] uses a proprietary framework that allows for the flexible scaling of networked services. Amazon's Autoscaling [34] automatically creates or destorys VMs when user-defined thresholds are exceeded. SnowFlock [35] provide sub-second scale-out using a VM fork abstraction. However, like the traditional solutions for elastic NFV, using VM as scaling unit is coarse-grained and results in a waste of resource. These studies are also limited to provider-offered applications and do not work for third-party VNFs.

CloudScale [36] is a fine-grained resource scaling system by combining online resource demand prediction and efficient prediction error handling to support multi-VM concurrent scaling with conflict prediction and predicted migration. Yu *et al.* [37] address bandwidth demand uncertainty of virtual clusters to provide bandwidth guarantee with a stochastic virtual cluster (SVC) and efficient VM allocation algorithms. DejaVu [38] is a framework for learning and reusing optimized VM resource allocation, which can automatically profile, cluster and classify workloads. However, Yu's solution is based on fixed VMs and CloudScale cannot achieve intra-cloud bandwidth scaling among VNFs, which is essential for an elastic service chain. DejaVu assumes that certain workload patterns exist, and identifies and stores resource assignment solutions for future use.

C. VM Migration

Fuerst *et al.* [19] proposed a migration algorithm to find a configurable and optimal tradeoff between embedding and reconfiguration cost. The migration algorithm is based on the standard Virtual Cluster (VC) model which cannot capture bandwidth demand among VNFs compared to the DMR model. Sandpiper [39] uses a greedy algorithm to migrate the VMs to the least loaded PM that can accommodate the VM. PAC [40] can find a suitable PM by matching the resource

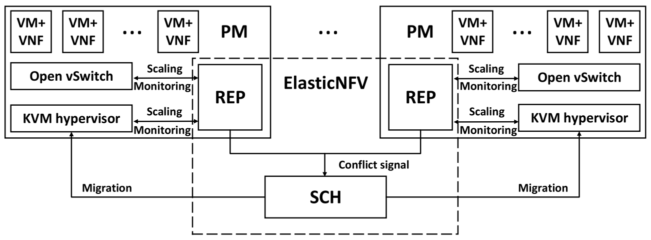


Fig. 2. Overview of ElasticNFV.

demand signature of the VM with the residual resource signature of the PM. Li *et al.* [41] propose two optimized live migration strategies for different application scenarios, one for load balance and fault tolerance, and the other for server consolidation. Zhang *at al.* [42] use a theoretical model to analyze how much bandwidth is required to guarantee the total migration time and downtime of a live VM migration, and then propose a novel transport control mechanism to guarantee the computed bandwidth. BAC [43] chooses suitable compression approach according to the network bandwidth available for the migration process, and employs multi-page compression, which make BAC obtain more migration performance improvements from compression. However, the optimization goal of these studies is not the migration time and the bandwidth between VNFs is also not considered.

III. ElasticNFV Overview

资源动态供应和伸缩冲突管理

ElasticNFV system consists of two modules, namely, the Resource Elastic Provisioning (REP) module and the Scaling Conflict Handling (SCH) module. Both modules are running on top of the KVM virtualization platform and Open vSwitch as show in Fig. 2. The REP module runs within each PM in the cloud system which provides resource elastic provisioning for VNFs. The SCH module runs within a controller node which solves the scaling conflict using migration algorithm.

REP module uses *libvirt virtualization APIs* and *sFlow* to monitor guest VM's resource usage from hypervisor and Open vSwitch. The monitored resource metrics include CPU consumption, memory allocation and intra-cloud network traffic. Measurements are taken every second. REP module uses the DMR model to represent the dynamic resource demand of service chains. REP module currently supports different scaling mechanisms based on the resource usage time series. The CPU resource scaling is realized by adjusting the CPU cap using the KVM's *CPU hotplug* [44] at runtime. The memory resource scaling is achieved using KVM's *memory ballooning* [45]. The intra-cloud network resource scaling is done through Open vSwitch's QoS rate limiting and OpenFlow flow table. Detailed are provided in Section IV.

When a scaling conflict happens, SCH module decides when to trigger a VNF instance migration and which VNF instance to migrate. ElasticNFV uses the *Resource Demand Prediction* (RDP) module developed in CloudScale [36]. Specifically, it uses a hybrid approach that employs a signature-driven prediction algorithm based on a fast Fourier transform (FFT) to identity repeating patterns and state-driven prediction algorithm based on a discrete-time Markov chain to predict the resource demand in the near future. Through the RDP module, ElasticNFV can decide whether to trigger VNF instance migrations or resolve scaling conflicts locally using SLO penalty. SCH module mainly consists of a TPMM algorithm that solves the CPU and memory conflicts in the first step and solves the bandwidth conflicts in the second step with minimal migration time. The TPMM algorithm is based on the DMR model which has an improved resource utilization during the migration process.

IV. Resource Elastic Provisioning (REP)

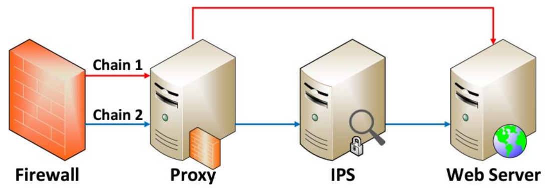
The REP module aims at providing dynamic and fine-grained cloud resource provisioning for VNFs. In this section, we present our DMR model including dynamic resource demands, a multi-resources elastic provisioning mechanism and the implementation of these elastic mechanisms on a cloud platform.

A. The DMR Model

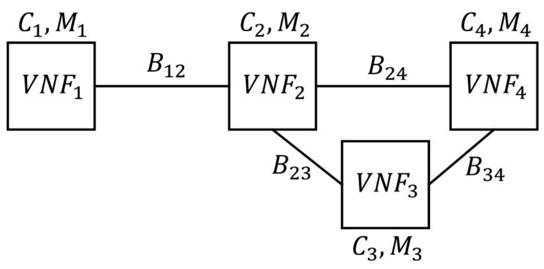
The DMR model is used to express realtime resource demand of one or more service chains formed by a set of VNFs. In ElasticNFV, each VNF has only one instance (one VM) with the vertical scaling. In addition to CPU and memory requirements of each VNF, DMR model also tracks VNF-to-VNF bandwidth requirements. A DMR(n, C, M, B) model has four parameters: n is the number of VNFs in the service chain; C is the vector for the number of vCPUs of all VNFs; M is the memory size vector for all VNFs; and B is the bandwidth matrix for the service chain. DMR model can not only express CPU, memory and bandwidth requirements of a service chain, it can also dynamically update resource caps according to realtime resource demand of the service chain. Fig. 3(a) illustrates a scenario that two service chains pass through multiple VNFs. More specifically, the traffic of chain 1 passes through a firewall, a caching proxy and then a Web server. The traffic of chain 2 passes through a firewall, a proxy, and then an intrusion prevention system before reaching a Web server. The DMR model can dynamically capture the resource demand of the two service chains over time, as shown in Fig. 3(b). In the graph each vertex represents a VNF and each edge represents the bandwidth between two VNFs.

B. CPU Elastic Provisioning

To cope with a higher CPU demand, a naive solution is to immediately raise the resource cap to the maximum possible value (i.e., all residual resources on the PM). However, this will cause excessive resource over-provisioning and scaling conflicts when concurrent scaling for multiple colocated VMs for upper VNFs. In the REP model we divide time into time windows. The length of each time window is one second. The resource pressure P ($0 \leq P \leq 1$) denotes the resource utilization ratio. For CPU resource, the scale-up action is triggered by adding a vCPU to VM (VNF instance) when P exceeds a certain threshold $P_{threshold}$ (e.g., $P_{threshold} = 0.9$). As our future work, we plan to propose a solution to dynamically determine the threshold according to the workload type and the SLO feedback. That way, the scale-down action is triggered



(a) Two service chains.



(b) DMR model implementation.

Fig. 3. Example DMR model implementation with two service chains. (a) Two service chains. (b) DMR model implementation.

by removing extra e vCPUs from VM (VNF instance) when P is less than $\frac{n-e}{n} \cdot P_{threshold}$, where n denotes the current vCPU number.

KVM supports CPU elastic provisioning which is also called *CPU hotplug* [44]. CPU hotplug is a technique that can be used to dynamically increase or decrease the number of vCPUs available in the KVM host (PM). In ElasticNFV, CPU hotplug is used to vertically scaling up or scaling down the vCPUs at runtime based on the VNF instances' requirements. When using CPU hotplug, one has to specify the maximum number of vCPUs available for the KVM guest (VM). ElasticNFV configures the maximum number of vCPUs for a KVM guest to be the vCPUs available in the KVM host, plus one more vCPU using *virsh setcpus* command on the fly. There is no direct method to remove the vCPUs from the KVM guest. However, we can bring back the CPU core power to the KVM hosts by disabling the CPU core to achieve scaling down.

C. *Memory Elastic Provisioning*

Unlike the discrete CPU elastic provisioning, memory elastic provisioning is continuous. The REP module raises the memory resource cap by multiplying the resource cap by a ratio $\alpha_{up} \geq 1$ when scale-up is triggered. Instead, The REP module divides the resource cap by $\alpha_{down} \geq 1$ when a scale-down action is triggered. The REP module dynamically decides the scale-up ratio α_{up} according to the severity of resource pressure P. We define α_{min} and α_{max} to be the scale-up ratios we want to use in the system, which can be tuned by a cloud provider. α_{min} is the minimum scale-up granularity we intend to achieve during the SLO violation. α_{max} is tuned so that we can scale up to the maximum resource cap in two or three time windows. We calculate α_{up} as follows:

$$\alpha_{up} = \frac{P - P_{threshold}}{1 - P_{threshold}} \cdot (\alpha_{max} - \alpha_{min}) + \alpha_{min} \qquad (1)$$

The REP module also dynamically determines the scale-down ratio α_{down} according to the resource usage. Scale-down is triggered when P is less than $min(\frac{P_{threshold}}{\alpha_{min}}, \frac{1}{\alpha_{max}})$. We set $\alpha_{down} = \frac{P_{threshold}}{P}$.

KVM provides a method to change the amount of memory in use by guests at runtime. The method is called *memory ballooning* [45], which is similar to the CPU hotplug. The KVM host uses balloon drivers running on the KVM guests (VMs) to determine how much memory it can take back from an underutilized VM. Balloon driver must be installed on any VM that uses the memory ballooning technique. Balloon driver gets the target balloon size from the hypervisor and then inflates by allocating the proper number of guest physical pages within the VM. This process is known as inflating the balloon; the process of releasing the available pages is known as deflating the balloon. However, since the system can not exceed the maximum memory limit at runtime, ElasticNFV can only scale up the memory of a KVM guest to the memory available in the KVM host. The *virsh setmem* command can be used to resize memory on the KVM guest on the fly.

D. *Bandwidth Elastic Provisioning*

Bandwidth elastic provisioning is also continuous. The elastic mechanism of bandwidth is similar to memory elastic provisioning. The REP module creates a VM-to-VM path between two VNFs in a service chain as a basic elastic unit. For scale-up action, we use a larger α_{min} and α_{max} for bandwidth scaling. This is because the VNFs reacts to the bandwidth scale-up faster than the memory scale-up. For example, we set $\alpha_{min} = 1.2$ and $\alpha_{max} = 2$ for bandwidth scaling, and $\alpha_{min} = 1.1$ and $\alpha_{max} = 1.5$ for memory scaling. Similarly, for scale-down actions, the REP module could reserve a headroom which is the gap between the physical link capacity and the maximum physical link usage to avoid congestion. Our implementation chose the gap value to be 10%.

We implement bandwidth elastic provisioning as a user-level process on the top of Open vSwitch. Most of the control logic of the Open vSwitch is realized by direct system command with real time arguments (depending on P and α) which will cost about 0.3% to 0.6% of the CPU resource. Though *QoS* commands one can create a VM-to-VM path between two VNFs in a service chain and dynamically set a max rate (bandwidth resource cap) for this virtual path.

V. Scaling Conflict Handling (SCH)

In this section, we describe how ElasticNFV handles concurrent resource scaling for multiple co-located VNF instances. The key issue is to deal with scale-up conflicts when the available resources are insufficient to accommodate all scale-up requests on a PM. We first describe how to achieve VNF instances' migration prediction. Then we introduce the migration algorithm. Some important notations and definitions used in the model are illustrated in Table I.

A. *Migration Prediction*

There are two methods to solve the scale-up conflicts: rejecting scale-up requests or moving out some VNF instances

TABLE I
SUMMARY OF KEY NOTATIONS AND DEFINITIONS

Notations	Definitions
Q_{RP}	the total resource under-provisioning penalty when using the local conflict handling scheme
Q_{MP}	the total migration penalty when using the migration-based conflict handling scheme
$P = (\bar{N}, \bar{L})$	the data center, where $\bar{N}$ is set of PMs and $\bar{L}$ represents physical links
$V = (N, L)$	the request from DMR model, where N is set of VMs and L represents virtual links
R	different types of resources of a PM (CPU and memory)
$v^r_{\bar{n}}$	capacity of PM $\bar{n} \in \bar{N}$ for resource type $r \in R$
v^r_n	capacity of VM $n \in N$ for resource type $r \in R$
$b_{\bar{l}}$	bandwidth capacity of physical link $\bar{l} \in \bar{L}$
b_l	bandwidth capacity of virtual link $l \in L$
$x_{n\bar{n}} \in \{0, 1\}$	a boolean variable that indicates whether VM n is embedded in PM $\bar{n}$
$y_{l\bar{l}} \in \{0, 1\}$	a boolean variable that indicates whether virtual link l is embedded in physical link $\bar{l}$
$y_{l\bar{n}} \in \{0, 1\}$	a boolean variable that indicates whether virtual link l is embedded in PM $\bar{n}$
$s_{nl} \in \{0, 1\}$	a boolean variable that indicates whether VM n is the source of virtual link l
$d_{nl} \in \{0, 1\}$	a boolean variable that indicates whether VM n is the destination of virtual link l
$t^n_{\bar{m}\bar{n}}$	the migration time of VM n from PM $\bar{m}$ to PM $\bar{n}$
$c^n_{\bar{m}\bar{n}}$	the embedding cost of VM n from PM $\bar{m}$ to PM $\bar{n}$
T	a 3-level tree
r	a scaling request
W_v	the unsettled VMs in node v
S_v	the settled VMs in node v
b_i[VM]	the sum of bandwidth of all links toward the specific VM
b_o[VM]	the sum of the bandwidth of all links outside a specific VM

through VM migration. Both approaches may cause SLO violations, but ElasticNFV aims at minimizing the cost through CloudScale [36] technology by answering two questions: 1) what is VNFs' resource demand? and 2) when do VNF instances trigger the migration?

CloudScale can predict the approaching total resource demand on a host using RDP module. The module can predict when a conflict will happen, how serious the conflict will be, and how long the conflict will last. If the conflict duration is short and the conflict degree is small, CloudScale resolves the conflict locally without triggering migration operations. CloudScale defines the *SLO penalty* as the financial loss for cloud providers when applications experience a SLO violation. CloudScale determines whether to trigger a migration or not by comparing Q_{RP} with Q_{MP}. If $Q_{RP} \leq Q_{MP}$, CloudScale does not migrate any VM and resolves the scale-up conflict temporarily using the local conflict handling scheme. Otherwise, CloudScale migrates some VMs out until sufficient resources are released to resolve the conflict. With RDP module and SLO penalty, we can acquire VNFs' resource demand and determine when do VNF instances trigger the migration.

B. Migration Algorithm

1) Setting: Different service chains using the DMR model need to be embedded in a given substrate: a physical network connecting a set of PMs. In this subsection, we present the service chains using a multi-rooted tree (or fat-tree) which is the predominant topology in today's data centers. A fat-tree is hierarchical and recursively made of subtrees at each level. A fat-tree data center is 3-level tree topology which consists of a set of pods which are interconnected by core switches. Pods are composed of a set of racks which are interconnected by a set of the aggregation switches. Racks consist of a set of PMs which are interconnected by a Top-of-Rack (ToR) switch. Each PM can host a fixed number of vCPUs and memory, and each physical link can have a fixed capacity of bandwidth. Fat-tree topology offers multiple paths between PMs. Multiple physical links can be seen as a single aggregated link if traffic is distributed evenly due to the amount of multiplexing and multi-path routing protocol such as ECMP [46], [47]. Some data center operators use the concept of over-subscription with an over-subscription factors $\gamma \geq 1$ to track the utilization of bandwidth resources. The embedding map of a service chain using the DMR model describes the embedding relationship between VNF instances and PMs in the substrate network.

2) Problem Model: The DMR model supports service chains of which the resources can be adjusted dynamically over time. Specifically, we want to be able to upgrade or downgrade a service chain DMR(n, C, M, B), where the service chain consists of n VNFs, and C CPU resources, M memory and B bandwidth resources. The service chain has a factor vector α ($\alpha_i \geq 0, i = 1, \ldots, n$) for CPU, a factor vector β ($\beta_i \geq 0, i = 1, \ldots, n$) for memory and a factor matrix γ ($\gamma_{ij} \geq 0, i = 1, \ldots, n, j = 1, \ldots, n$) for bandwidth, i.e., to DMR(n, $C \times \alpha$, $M \times \beta$, $B \times \gamma$).

The challenge is how to support such configuration. In particular, we would like to minimize the migration cost when processing a scale-up request. We propose a model as follows to formulate the performance of the migration for scale-up request such as VDC Planner [48]. The constraints of physical resources can be written as:

PM资源限制

$$\sum_{n \in N} x_{n\bar{n}} \cdot v_n^r \leq v_{\bar{n}}^r \quad \forall \bar{n} \in \bar{N} \tag{2}$$

$$\sum_{l \in L} y_{l\bar{l}} \cdot b_l \leq b_{\bar{l}} \quad \forall \bar{l} \in \bar{L} \tag{3}$$

链路资源限制

Equation (2) and (3) ensure no violation of the capacities of PMs and physical links. Our next step is to consider VM (VNF instance) placement constrains. To ensure the embedding of each VM n and each virtual link l, we must have:

$$\sum_{\bar{n} \in \bar{N}} x_{n\bar{n}} = 1 \quad \forall n \in N \tag{4}$$

$$\sum_{\bar{l} \in \bar{L}} y_{l\bar{l}} + \sum_{\bar{n} \in \bar{N}} y_{l\bar{n}} = 1 \quad \forall l \in L \tag{5}$$

In our model, we also require that the virtual link embedding satisfies the flow constraint between the source and destination node pair in each service chain topology, formally written as:

$$\begin{aligned} \sum_{\bar{l} \in \bar{L}} s_{\bar{n}\bar{l}} \cdot y_{l\bar{l}} - \sum_{\bar{l} \in \bar{L}} d_{\bar{n}\bar{l}} \cdot y_{l\bar{l}} &= \sum_{n \in N} s_{nl} \cdot x_{n\bar{n}} - \sum_{n \in N} d_{nl} \cdot x_{n\bar{n}} \\ &\forall l \in L, \quad \bar{n} \in \bar{N} \end{aligned} \tag{6}$$

Equation (6) states that the total outgoing flow of a PM $\bar{n}$ for a virtual link l should be zero unless $\bar{n}$ hosts either the source or destination node of virtual link l. Finally, we also consider the migration time and embedding cost. The migration time $t_{\bar{m}\bar{n}}^n$ is given by:

$$t_{\bar{m}\bar{n}}^n = \begin{cases} migTime(n, \bar{m}, \bar{n}) & if \quad \bar{n} \neq \bar{m} \\ 0 & if \quad \bar{n} = \bar{m} \end{cases} \tag{7}$$

where *migTime*(n, $\bar{m}, \bar{n}$) denotes the migration time of VM n from PM $\bar{m}$ to PM $\bar{n}$. This time is equal to zero when n is already embedded in the PM $\bar{n}$ (i.e., $\bar{m} = \bar{n}$). The embedding cost $c_{\bar{m}\bar{n}}^n$ is given by:

$$c_{\bar{m}\bar{n}}^n = \begin{cases} embCost(n, \bar{m}, \bar{n}) & if \quad \bar{n} \neq \bar{m} \\ same & if \quad \bar{n} = \bar{m} \end{cases} \tag{8}$$

where *embCost*(n, $\bar{m}, \bar{n}$) denotes the embedding cost of VM n from PM $\bar{m}$ to PM $\bar{n}$. This cost is kept unchanged when n is already embedded in the PM $\bar{n}$ (i.e., $\bar{m} = \bar{n}$).

The objective function of the migration can be defined by minimizing the overall cost, written as:

$$min \sum_{n \in N} \sum_{\bar{n} \in \bar{N}} x_{n\bar{n}} \cdot (p(t_{\bar{m}\bar{n}}^n) + c_{\bar{m}\bar{n}}^n), \tag{9}$$

subject to constraints equations (5) - (9). Here, $p(t_{\bar{m}\bar{n}}^n)$ is a penalty function of $t_{\bar{m}\bar{n}}^n$ which balances the tradeoff between the migration time and the embedding cost. For example, if $t_{\bar{m}\bar{n}}^n \leq \alpha$ (where α is a constant), then $p(t_{\bar{m}\bar{n}}^n) = 0$. Otherwise, $p(t_{\bar{m}\bar{n}}^n)$ grows rapidly when $t_{\bar{m}\bar{n}}^n$ increases. The problem can be generalized into an *NP*-hard multi-dimensional knapsack problem. Therefore, we propose a heuristic algorithm called TPMM to find a solution for this problem.

3) The Two-Phase Minimal Migration (TPMM): We design the TPMM algorithm to fulfill any upgrade request whenever there are sufficient resources available in the substrate.

Algorithm 1 Two-Phase Minimal Migration

```
Require: Topology tree $T$
Input: Request $r$ : $< n, C \times \alpha, M \times \beta, B \times \gamma >$
 1: a = GetResourceAvailable($T$)          ▷ calculate available resources on $T$
 2: d = GetResourceDemand($r$)              ▷ get resource demand of request $r$
 3: if $d \leq a$ then
 4:     update on $T$ according to request $r$
 5:     return true
 6: if $Q_{RP} \leq Q_{MP}$ then    ▷ migrate some VNFs to handing the conflict
 7:     $l$ = 0
 8:     while $l \leq 3$ do
 9:         for every $v$ at level $l$ whose sub-tree places some VNFs of $r$ do
10:             if $l == 0$ then                                  ▷ the PM level
11:                 $W_v$ = all the VNFs of $r$ in $v$
12:                 $S_v = \emptyset$
13:                 $(W_v, S_v)$ = SettleInServer($v$, $W_v$, $S_v$)
14:             else// the switch level
15:                 $(W_v, S_v)$ = SettleInSwitch($v$, $W_v$, $S_v$, $l$)
16:             if size($S_v$) == the number of VNFs of $r$ then
17:                 return true
18:             else if $l == 3$ then                      ▷ reach the root of $T$
19:                 roll back to the state before executing
20:                 return false
21:             else                   ▷ upload the wait set and the settled set
22:                 $v'$ = parent node of $v$
23:                 $W'_v += W_v$
24:                 $S'_v += S_v$
25:         $l = l + 1$
26: return true
```

如果资源需求小于可用资源，则根据请求r更新T上服务链的资源上限(第1-5行)。

否则，当QRP≥QMP(第6行)时，算法触发迁移操作；QRP表示总资源供应不足损失，QMP表示总迁移损失

对于交换机级别上的每个冲突节点，我们调用函数settinswitch来处理此级别上的冲突(第14-15行)。

从左到右遍历每个层级从下到上的冲突节点。对于PM级别的每个冲突节点，我们调用函数settinserver来处理此级别的冲突(第10-13行)。

如果到达根节点前Sv =□ 表示迁移算法执行成功(第16-17行)。

当到达T的根时，迁移算法失败并回滚到执行前的状态(第18-20行)。

否则，我们将等待集Wv和结算集Sv上传到父节点vΔ并继续循环(第21-24行)。

迁移决策：只选择最近的机器迁移，也就是只考虑是否迁移

Furthermore, the TPMM algorithm aims at achieving optimal embedding cost in equation (9). To optimize the migration time of VNFs, we always move the VNFs which need to be migrated to the closest machines based on the first-fit strategy to reduce the bandwidth cost. We choose the VNFs which bring the minimal migration cost.

Note that there is a trade-off between the two metrics: at the price of higher migration time, a lower embedding cost can be achieved often. We design our algorithm according to the following priorities: (1) the top priority is to satisfy a migration request; (2) the second priority is to minimize the migration time; and (3) the third priority is to minimize the embedding cost.

Algorithm 1: shows the pseudo-code of the TPMM algorithm which realizes an migration operation from DMR(n, C, M, B) to DMR(n, C $\times \alpha$, M $\times \beta$, B $\times$ γ). The algorithm first compares new resource demand and available resources on T. If the resource demand is less than available resources, we update the resource cap of service chain on T according request r (line 1-5). Otherwise, the algorithm triggers a migration operation when $Q_{RP} \geq Q_{MP}$ (line 6), where Q_{RP} denotes the total resource under-provisioning penalty and Q_{MP} denotes the total migration penalty. We traverse every conflicting node from left to right at each level from bottom to top. For each conflicting node at PM level, we call function *SettleInServer* to handling the conflict at this level (line 10-13). For each conflicting node at the switch level, we call function *SettleInSwitch* to handling the conflict at this level (line 14-15). When reaching the root of T, the migration algorithm fails and rolls back to the state before executing (line 18-20). If $S_v = \emptyset$ before reaching the root, the migration algorithm is executed successfully (line 16-17). Otherwise, we upload wait set W_v and settled set S_v to the parent node v' and continue to loop (line 21-24).

Algorithm 2 Function SettleInServer(v, W_v, S_v)

```
1: part 1: knapsack problem, cope with cpu and memory resources
2: $(W_v, S_v)$ = Knapsack($v$, $W_v$, $S_v$)   ▷ CPU is volume and memory is value
3: part 2: heuristic search, cope with bandwidth resource
4: $b$ = GetUpstreamBandwidth($S_v$)
5: while $b$ exceeds the current limit and size($S_v$) >0 do
6:     select the exact VM that makes $b_o$[VM] - $b_i$[VM] greatest
7:     $S_v$ -= VM
8:     $W_v$ += VM
9:     $b$ = GetUpstreamBandwidth($S_v$)
10: return $(W_v, S_v)$
```

Algorithm 3 Function SettleInSwitch(v, W_v, S_v, l)

```
1: if $l$ - 1 == 0 then
2:     for every child $v'$ of $v$ in the substrate do
3:         $(W_v, S_v)$ = SettleInServer($v'$, $W_v$, $S_v$)
4:         if size($W_v$) == 0 then
5:             break
6: else
7:     for every child $v'$ of $v$ in the substrate do
8:         $(W_v, S_v)$ = SettleInSwitch($v'$, $W_v$, $S_v$, $l$ - 1)
9:         if size($W_v$) == 0 then
10:            break
11: $b$ = GetUpstreamBandwidth($S_v$)
12: if $b$ exceeds the current limit then
13:     for every VM in $S_v$ do                ▷ roll back
14:         $S_v$ -= VM
15:         $W_v$ += VM
16: return $(W_v, S_v)$
```

Algorithm 2: shows the pseudo-code of the function *SettleInServer* that decides which VNFs can be settled in a specific PM when scale-up conflict happened using the two-phase selection. The function *Knapsack* firstly selects the settled VNFs which meet CPU and memory capacity of this PM and minimizes the migration cost (memory size of the VNFs needed to be migrated) using the knapsack algorithm. vCPU number is the volume and memory size is the item value in the knapsack problem. That is to say, the vCPU number of the settled VNFs should be equal to or less than the CPU capacity of this PM and memory size of the settled VNFs should be less than and close to the memory capacity of this PM (line 1-2). The second phase selection makes the settled VNFs' outside bandwidth smaller than the bandwidth of the PM's upper link. We choose a VM that makes b_o[VM] - b_i[VM] the largest to migrate each time until the bandwidth constraints are met (line 3-9).

Algorithm 3: shows the pseudo-code of the function *SettleInSwitch* that searches which VNFs shall be settled in under a specific switch when a scale-up conflict occurs. For each child node v', if v' is a PM, then the function *SettleInSwitch* could be applied to v' (line 1-5). If v' is also a switch, the *SettleInSwitch* can be called recursively on v' (line 6-10). Finally, we testify whether or not all the bandwidth demands on T are within the bandwidth capacity of physical links (line 11-15).

The computation complexity of TMPP algorithm is bounded by $O(S \cdot (N \cdot C \cdot M + N^2))$ in the worst-case scenario, where S is the number of PM, N is the number of VM (VNF instance), C and M are the count of CPU and memory resource of a single PM, respectively. To be more specific, the computation complexity of the first phase selection and second phase selection are $O(S \cdot N \cdot C \cdot M)$ and $O(S \cdot N^2)$, respectively. Hence, the upper bound of TPMM's time cost is determined by the larger one of the two.

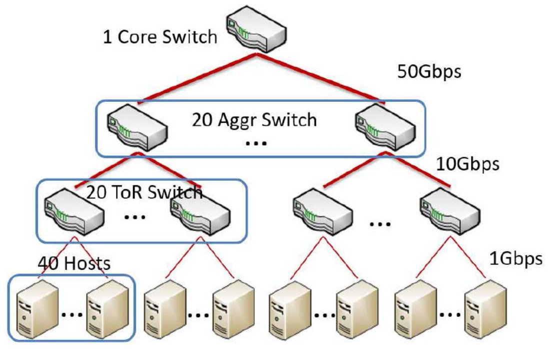


Fig. 4. 3-level tree topology.

VI. Evaluation

In this section we evaluate the performance of ElasticNFV on two ways. First, we use large-scale simulations to quantify the benefits of the TPMM algorithm. Second, we benchmark our implementation on a small testbed. We show that the ElasticNFV can improve VNF performance, and achieve dynamical fine-grained cloud resource provisioning for VNFs according to the workload change.

A. Simulation Setup

We developed a simulator that coarsely models a multi-tenant cloud datacenter. For simplicity, the simulator uses a 3-level tree topology with no path diversity. The cloud datacenter has 1 core switch, 20 aggregation switches, 400 ToR switches and 16000 PMs, as shown in Fig. 4. We used homogenous PM type with identical CPU (12 vCPU), memory (32GB) and bandwidth capacity (1Gbps). The upper-link bandwidth of PM, ToR switches and aggregation switches are 1Gbps, 10Gbps and 50Gbps, respectively. The over-subscription rate of physical network is 4. The default size of CPU, memory and bandwidth for a VM are chosen from an exponential distribution with the mean CPU, memory and bandwidth of 4vCPU, 8GB and 100Mbps, respectively.

We let new DMR requests arrive according to a Poisson process with $\lambda = 0.36$. The lifetime of each service chain is chosen according to an exponential distribution with average 3600s (one hour). To add diversity to the DMR requests, we use six additional Poisson processes which continuously pick service chains for scaling up and/or scaling down in a multiplicative manner. For example, process (1) scales up and process (2) scales down vCPU by a factor f_c (f_c corresponds to α in the former sections); process (3) and (4) scaling up or down the memory size by a factor f_m (f_m corresponds to β in our formal sections), (5)+(6) scaling up or down the bandwidth by a factor f_b (f_b corresponds to γ in the former sections). By default, we assume that $f_c = f_m = f_b = 1.5$. With regards to reporting the results, we focus on the scaling up as these are the ones which trigger migration.

B. Simulation Results

1) Baseline Comparison: ElasticNFV allows to scale up an existing service chain by increasing the bandwidth between VNFs at their current locations, as well as by extending the vCPU numbers and memory size of VNFs. If a local extension is not sufficient to satisfy a request and triggers the migration, ElasticNFV also supports the re-embedding process using the TPMM algorithm.

In order to understand the contribution of the TPMM algorithm, we compare TPMM with Sandpiper [39] which uses a greedy algorithm to migrate the VNFs to the least loaded PM that can accommodate the VNF. Sandpiper can only fix the CPU and Memory scaling up requests, so we extended Sandpiper using the TPMM's second phase selection to meet the new bandwidth demand. For a simple baseline comparison, we also re-implemented Oktopus [46]; we extend Oktopus so that requests can be satisfied by migration with first-fit method.

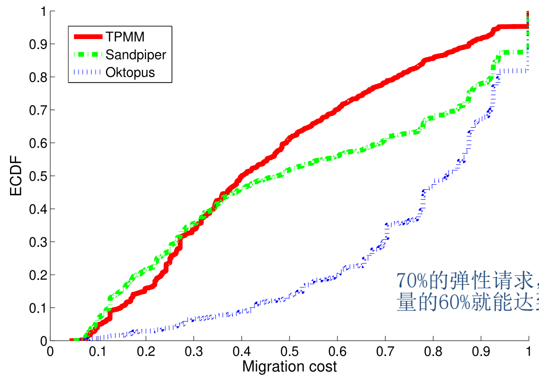
To give a basic understanding of the memory size of migrations required to support elastic service chains, Fig. 5 plots the empirical cumulative distribution function (ECDF) of the migration cost for the three algorithms: TPMM, Sandpiper and Oktopus, and three operations: add more vCPUs or increase memory size, upgrade bandwidth. The three operations can also be requested simultaneously to form a joint upgrade.

The corresponding results for adding vCPUs and memory are shown in Fig. 5(a). We can see that the superiority of TPMM is more significant. When migrating 60% of the memory size, TPMM can satisfy 70% of the requests, while Sandpiper and Oktopus can satisfy 56% and 20% of the requests, respectively. TPMM can satisfy 86% of the requests for migrating 80% of the memory size, while Sandpiper and Oktopus can only satisfy 68% and 47% of the requests, respectively.

We secondly discuss a scenario where only the bandwidth is upgraded, as shown in Fig. 5(b). In general, the performance of TPMM is close to but better than Sandpiper, and far better than Oktopus. TPMM and Sandpiper can satisfy 35% of the requests for migrating less than 20% of the memory size, while Oktopus can only satisfy 8% of the requests for migrating less than 20% of the memory size. TPMM can avoid migrating more than 50% of the memory size for nearly 60% of the requests, while Sandpiper and Oktopus can achieve that for nearly 45% and 30% of the requests, respectively. When migrating 80% of the memory size, TPMM can satisfy 83% of the requests, while Sandpiper and Oktopus can achieve that for 76% and 73% of the requests, respectively.

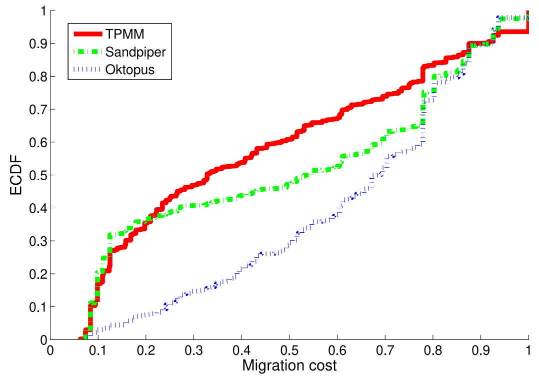
Fig. 5(c) shows the results of joint upgrades (vCPU number, memory size and bandwidth). Here, the overall performance of TPMM, Sandpiper and Oktopus becomes a mixture of the previous cases. TPMM can avoid migrating more than 60% of the memory size for nearly 65% of the requests, while Sandpiper and Oktopus can achieve that for nearly 50% and 20% of the requests, respectively. TPMM can satisfy 82% of the requests for migrating 80% of the memory size, while Sandpiper and Oktopus can achieve that for 66% and 51% of the requests, respectively.

In TPMM, we use the solution of the knapsack problem to choose which VM to migrate. The goal of the knapsack problem is to minimize the memory size of the VNFs needed to be migrated, instead of the number of VNFs needed to be migrated. In contrast, Sandpiper selects VM to be remove by a descending order of load. Hence, when the migrate cost is lower than 30%, we suspect that TPMM prefers the two smallest VMs while Sandpiper prefers the biggest one. In this case, CPU and memory resources are usually not the bottle neck while bandwidth is likely to be. More VMs need to be migrated to other PMs means more bandwidth cost. As a result, Sandpiper can perform better than TPMM when migration cost is low. But if we consider the overall tendency, TPMM surely outperforms Sandpiper.

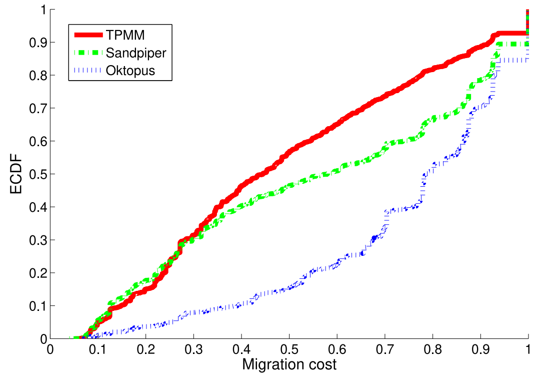


70%的弹性请求，只要迁移业务数据量的60%就能达到

(a) Add vCPU number or memory size.

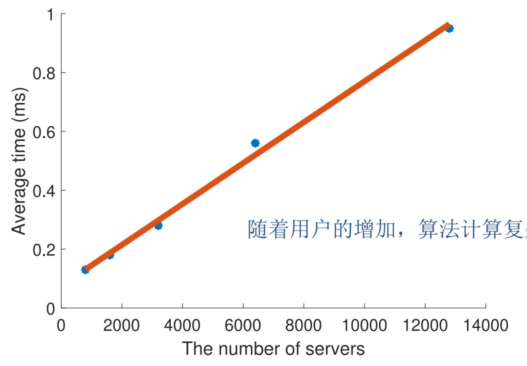


(b) Upgrade bandwidth.



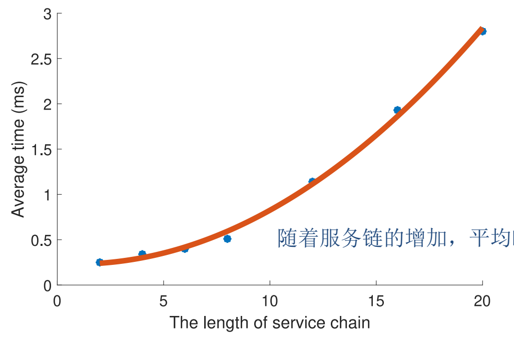
(c) Joint upgrade.

Fig. 5. Migration cost: TMPP vs. Sandpiper vs. Oktopus. (a) Add vCPU number or memory size. (b) Upgrade bandwidth. (c) Joint upgrade.



随着用户的增加，算法计算复杂度呈线性

Fig. 6. TMPP average time with varying the number of servers.



随着服务链的增加，平均时间

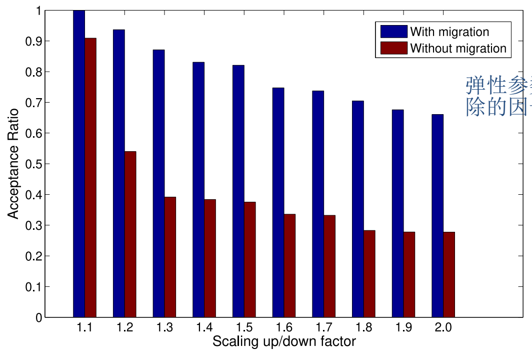
Fig. 7. TMPP average time with varying the length of service chain.

Based on the analysis above, we learned that the upper bound of our algorithm is $O(S \cdot (N \cdot C \cdot M + N^2))$. Since S and N are usually the major independent variables, we have a theoretical upper bound by $O(S \cdot N^2)$. In order to show the runtime effectiveness of our proposed algorithm, we do a series of experiments varying the number of servers and the length of service chain respectively, to observe the average runtime for each request.

As shown in Fig. 6, when the number of servers reaches 12,800, which is more than 10,000 requests, the average time is still less than 1ms. The relationship between the number of servers and average time is linear. When we increase the length of service chain from 2 to 20, we can see in Fig. 7 that the average time is nearly a quadratic function of the service chain length. Specifically, the average time is 0.51ms when the length of service chain is 8. From the results of runtime experiments, we can conclude that the TPMM algorithm is robust and effective in large-scale datacentres.

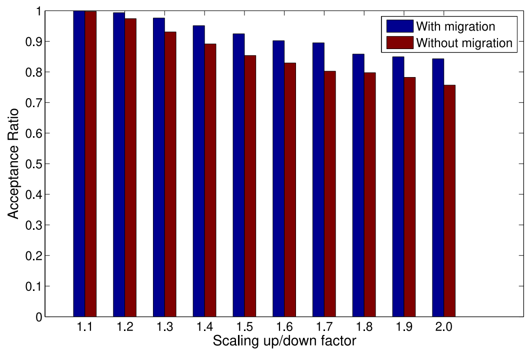
2) Sensitivity Study: We also conducted a sensitivity study of TPMM, in which we performed parameter sweeps for the scaling up and down ratios f_c, f_m and f_b and the substrate load. In particular, we have done a series of experiments varying the life time parameter values of DMR requests (λ and μ). The overall tendency is the same. Therefore, we only show the experimental results with $\lambda = 0.36$. We will first study the effect of the scaling up ratios f_c, f_m and f_b in greater detail, and subsequently, we report on our observations for the platform load.

Fig. 8 shows the acceptance ratio as the service chain scales up. The red area represents the scaling up requests that do not require a migration. The blue area of the bar corresponds to

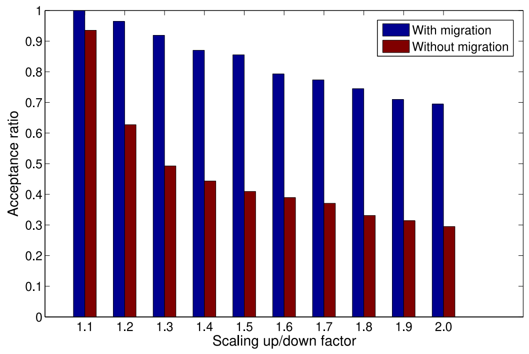


弹性参数：即乘法和除法乘除的因子

(a) Add vCPU number or memory size.



(b) Upgrade bandwidth.



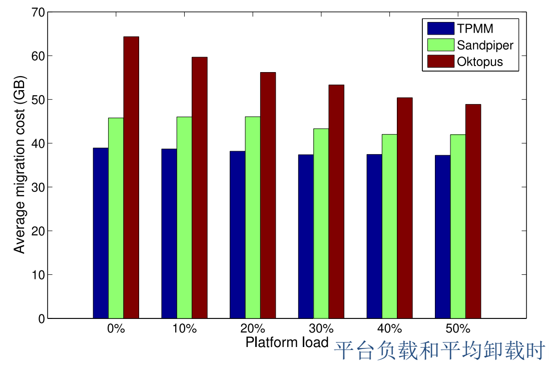
(c) Joint upgrade.

Fig. 8. TMPP acceptance ratio: with migration, without migration. (a) Add vCPU number or memory size. (b) Upgrade bandwidth. (c) Joint upgrade.

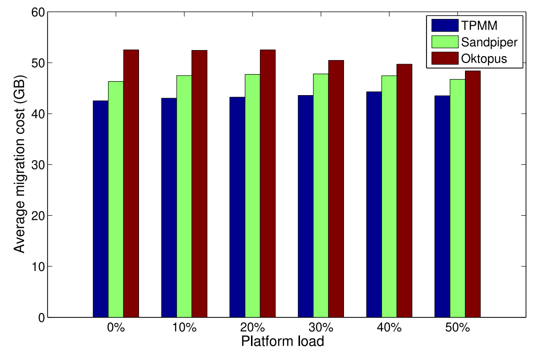
those requests that can be satisfied by ElasticNFV but require a migration. We also have three subplots corresponding to the three operations: add vCPU number or memory size, upgrade bandwidth, and jointly upgrade three kinds of resources.

The impact of the scaling up ratios f is significant, opening a spectrum from accepting almost all requests without migrations (for factors close to one) to no migration for only 30% of VNFs' CPU and memory upgrade requests. Fig. 8(a) demonstrates that the range of the acceptance ratio for TPMM is 66% - 100%. The range of the acceptance ratio without migration is 28% - 91%. When the scaling up ratio is 1.5, the acceptance ratios of migration and without migration are 82% and 38% respectively.

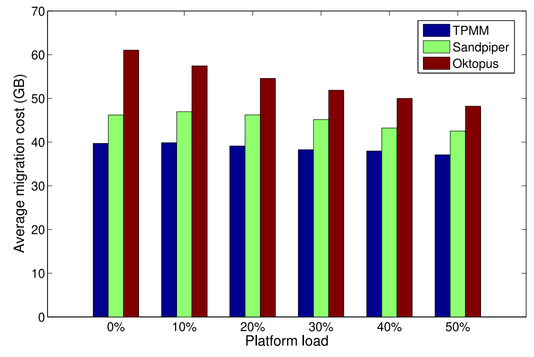
The impact of f on the bandwidth upgrades is less articulated. Indeed, the problem is unfeasible for 70% of the requests



(a) Add vCPU number or memory size.



(b) Upgrade bandwidth.



(c) Joint upgrade.

Fig. 9. Average migration cost with varying platform load: TMPP vs. Sandpiper vs. Oktopus. (a) Add vCPU number or memory size. (b) Upgrade bandwidth. (c) Joint upgrade.

if the upgrade factor is 2 in joint upgrade scenario. As shown in Fig. 8(b), the range of the acceptance ratio with migration is 66% - 100%. Without migration, the range of the acceptance ratio decreases to 28% - 91%. When f_b equals to 1.5, the acceptance ratios of migration and without migration are 93% and 86% respectively. Regarding to the embedding costs we find that VNFs' CPU and memory upgrades are typically more costly. This is fully consistent with the observations above.

Fig. 8(c) shows the results of joint upgrades. When joint upgrades the scaling up and down ratios f_c, f_m and f_b, the acceptance ratio of TPMM and without migration both decrease correspondingly. The range of the acceptance ratio with migration is 70% - 100%. Without migration, the range of the acceptance ratio is 30% - 94%. As the scaling up ratio

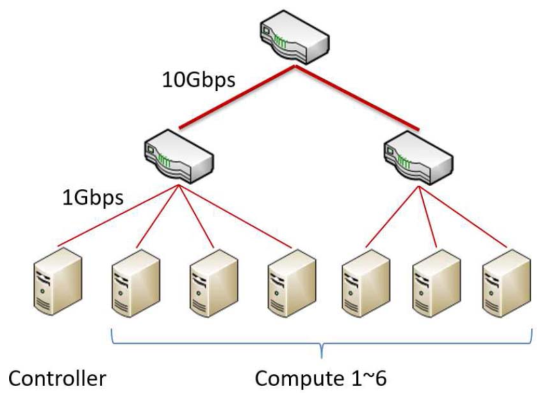


Fig. 10. Testbed topology.

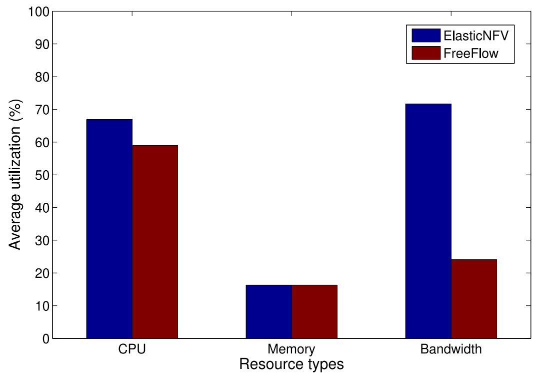
grows to 1.5, the acceptance ratios of migration and without migration are 86% and 41% respectively. The TPMM algorithm shows good robustness when varying the scaling up and down ratios.

We now report our observations on the platform load: Varying the platform load from 0% to 50% and compare the average migration cost of the three algorithm. Fig. 9 also shows three subplots corresponding to the three upgrade operations. In general, as for the average migration costs of TPMM and Sandpiper, the impacts of the platform load are not significant. However, the average migration cost of Oktopus goes down as the platform load goes up. This is because adding platform load using first-fit simplifies the structure of the physical substrate. As a result, Oktopus will have much fewer opportunities to settle virtual clusters across the core switch, which leads to a less bandwidth-consumed migration result and better performance.

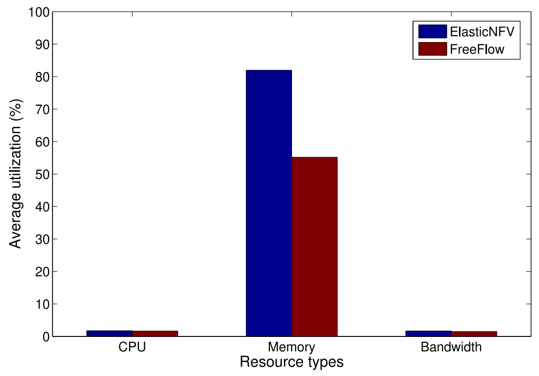
The corresponding results for the CPU and memory upgrades are shown in Fig. 9(a). colorblueThe ranges of the average migration cost are 37GB-39GB for TPMM, 42GB-46GB for Sandpiper, and 49GB-64GB for Oktopus. TPMM migrates about 15% less memory than Sandpiper and 40% less memory than Oktopus in normal case. When the platform load reaches 50%, TPMM migrate about 10% less memory than Sandpiper and 23% less memory than Oktopus.

As a next step we discuss a scenario where only the bandwidth is upgraded as shown in Fig. 9(b). The ranges of the average migration cost is 43GB-44GB for TPMM, 46GB-48GB for Sandpiper, and 48GB-53GB for Oktopus. In normal case whose platform load is 0%, TPMM migrates about 8% less memory than Sandpiper and 19% less memory than Oktopus. When the platform load reaches 50%, TPMM still performs the best, that migrate about 7% less memory than Sandpiper and 10% less memory than Oktopus.

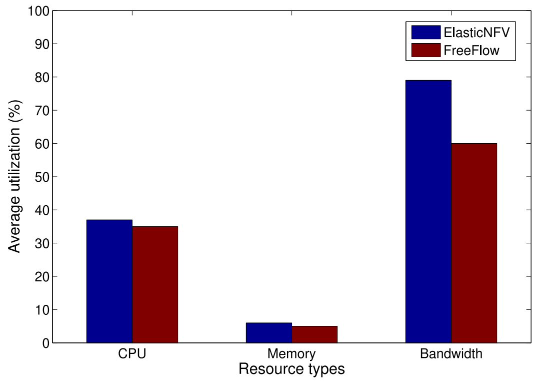
Fig. 9(c) shows the results of joint upgrade. The ranges of the average migration cost is 37GB-40GB for TPMM, 43GB-47GB for Sandpiper, and 48GB-61GB for Oktopus. As the platform load changes from 0% to 50%, TPMM always migrates 12% to 15% less memory than Sandpiper and 23% to 35% less memory than Oktopus. Hence we can see that TPMM outperforms the other two algorithms from the aspect of average migration cost when varying the platform load.



(a) Suricata.



(b) Varnish.



(c) Flow Monitor.

Fig. 11. Average utilization: ElasticNFV vs. FreeFlow. (a) Suricata. (b) Varnish. (c) Flow Monitor.

C. Testbed Experiment

We evaluate ElasticNFV with a small-scale prototype deployment. The goals of this evaluation are to: 1) confirm whether or not ElasticNFV provides more efficient utilization of resource than scaling in/out NFV; 2) confirm whether or not ElasticNFV has faster completion time than scaling in/out NFV.

We built a small scale-datacenter testbed consisting of 7 physical machines and we deployed OpenStack software (release Kilo) to this datacenter as a private cloud which contains 1 controller node and 6 compute nodes, as shown in Fig. 10. KVM is configured as the hypervisor for the compute notes. We implemented ElasticNFV on this private cloud. Each server has two 2GHz Intel Xeon E5-2620 CPUs and 16GB of memory and each VM has one vCPU, 1GB of memory and 500Mbps bandwidth. We verify the above goals of ElasticNFV in one scenarios: VM Z processes the traffic sent by VM X with two different scaling ways. We deploy three NFs (Suricata, Varnish and flow monitoring implemented on Click) on VM Z respectively and the traffic is processed by VNF Z. As the flow changes, ElasticNFV adjusts the vCPU number and memory size of Z and bandwidth between X and Z. We compare ElasticNFV with FreeFlow [9] (a scaling in/out solution as the existing of enterprises and organizes do) with real workloads from CAIDA [49].

Fig. 11(a) shows the utilization of three kinds of resource for two different scaling solutions ElasticNFV and FreeFlow deploying Suricata. A 18GB workload is sent from VM X in this experiment. ElasticNFV can achieve 66.9% CPU utilization, while FreeFlow is 58.9% in the whole process. The result is very close because adding or removing a vCPU number is equal to adding or removing a VM replica. The memory utilization of these two scaling solutions are both 16.3% because the memory usage is irrelevant to the workload. ElasticNFV can achieve 71.7% bandwidth utilization, while FreeFlow is 24.1% in the whole process. The bandwidth elastic provisioning is also continuous, the effect of ElasticNFV is significant.

Fig. 11(b) shows the results of two different scaling solutions deploying Varnish. The Varnish HTTP cache is a memory-bound VNF which only consumes a small amount of CPU and bandwidth resource. Therefore the utilization of CPU and bandwidth for these two different scaling solutions is very close. ElasticNFV can achieve 82.0% memory utilization, while FreeFlow is 55.2% in the whole process. The memory elastic provisioning is also continuous. Therefore, ElasticNFV performs better than FreeFlow.

Fig. 11(c) shows the result of flow monitoring implemented on Click. We can see that ElasticNFV can achieve 36.3% CPU utilization, while FreeFlow is 35.6% in the whole process. The memory utilization of these two scaling solutions are both close to 5.5% because the memory usage is irrelevant to the workload in this case. ElasticNFV can achieve 79.6% bandwidth utilization, while FreeFlow is 60.1% in the whole process. The effect of ElasticNFV is significant.

We calculate the completion time for two different scaling solutions deploying the three different NFs, as shown in Fig. 12. The completion times of ElasticNFV and FreeFlow in Suricata are 569s and 645s respectively. ElasticNFV is 13.3% faster than FreeFlow. When deploying Varnish, the completion time of ElasticNFV is 382s, and FreeFlow is 422s. ElasticNFV is 10.5% faster than FreeFlow. The completion time of ElasticNFV and FreeFlow in Flow Monitor are 577s and 620s, and ElasticNFV is 7.5% faster than FreeFlow. The testbed experiments show that the vertical scaling solution ElasticNFV can achieve better performance than the existing horizontal scaling solution.

VII. Discussion and Future Work

In this section, we briefly discuss several possible extensions of the overall ElasticNFV system design.

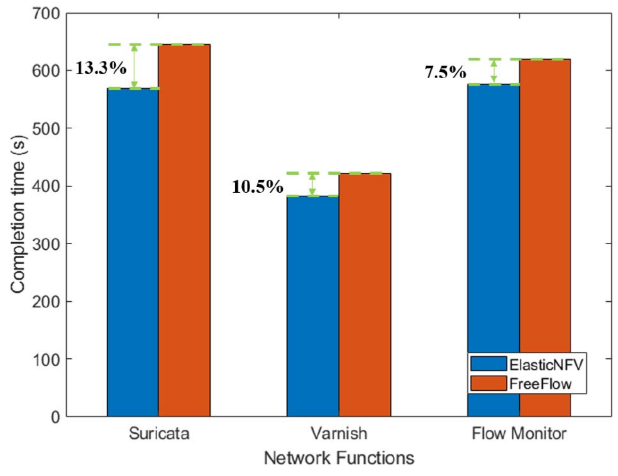


Fig. 12. Completion time: ElasticNFV vs. FreeFlow.

Compared to horizontal scaling method, ElasticNFV provides a fine-grained cloud resource provisioning for VNFs, with an efficient utilization of resources, short scaling period and fast response time. Although the vertical scaling method is superior in terms of performance optimization, it still suffers from two problems. First, Cao *et al.* [50] shows a horizontal scaling method outperforms a vertical scaling method in terms of compatibility for NFs, such as the single-threaded application Snort [51]. Second, horizontal scaling method has priority to vertical scaling method in scalability. Limited to PM capacity, the vertical scaling method suffers from the performance bottleneck of single node. Therefore, the vertical scaling method is more suitable for small and medium businesses. We plan to propose a hybrid scaling method that combines the advantages of both methods to improve the fine-grained cloud resource provisioning model for VNFs in the future.

Although the hybrid elasticity brings much convenience, it will suffer if we do not consider the overall placement of VNF instances [52]. Since the scaling of VNF instances is determined dynamically by temporal workload, the specific placement is unpredictable. A careless placement may leads to a much longer packets traveling distance, which may waste the bandwidth resource as a result [53]. In addition, a unsuccessful placement may cause significant VM operation overhead (e.g., VM launch, termination and migration). The ever-changing workload in cloud platform may lead to frequent VNF instance scaling and migration, which inevitably brings much VM operation overhead. Combined with the hybrid elasticity, we plan to propose an efficient VNF instances placement algorithm for resource saving and overhead reduction as our future work. Moreover, the TPMM algorithm should be applicable to heterogeneous PM type. Therefore, we also plan to do more evaluation for the proposed migration algorithm on heterogeneous environment as our future work.

Virtual machines and containers are the two most widely deployed virtualization mechanisms in the cloud. Currently the resource scaling engine is based on KVM virtualization platform and Open vSwitch. Containers such as LXC [54] and Docker [55] start to be prevalent for tenant and application isolation in cloud ecosystems. Compared to VMs, containers exhibit lower overhead and higher performance. We plan to compares the resource usage efficiency and system performance of VMs with those of containers for achieving fine-grained cloud resource provisioning for VNFs.

VIII. Conclusion

In this paper, we present ElasticNFV: a dynamic solution that provides a dynamic and fine-grained cloud resource provisioning for VNFs. ElasticNFV consists of two key modules: 1) the REP module uses a DMR model to capture realtime resource demand of one or more service chains and incorporates both resource elastic provisioning mechanism and resource provisioning technique. 2) the SCH module can not only predict the time to trigger the migration and the resource demand of VNFs, it also formulates the performance of migration and provides a TPMM algorithm to optimize the migration time and embedding cost. We implemented ElasticNFV on top of the KVM and Open vSwitch. Through simulations and testbed experiments with real traces, we show that ElasticNFV can significantly improve VNF performance, and achieve efficient utilization of resource and fast migration time with low cost.

References

[1] V. Sekar, R. Ratnasamy, M. Reiter, N. Egi, and G. Shi, "The middlebox manifesto: Enabling innovation in middlebox deployment," in *Proc. HotNets*, 2011, pp. 1–6.

[2] J. Sherry, S. Hasan, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxs someone else's problem: Network processing as a cloud service," in *Proc. ACM SIGCOMM*, 2012, pp. 13–24.

[3] *NFV—White Paper*. Accessed: Jun. 10, 2018. [Online]. Available: https://portal.etsi.org/NFV/NFV_White_Paper.pdf

[4] R. Mijumbi, J. Serrat, J. Gorricho, N. Bouten, F. Turck, and R. Boutaba, "Network function virtualization state-of-the-art and research challenges," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 1, pp. 236–262, 1st Quart., 2016.

[5] *CloudNFV: Taking NFV to the Cloud*. Accessed: Jan. 20, 2019. [Online]. Available: https://www.cloudnfv.com/

[6] *Open Platform for NFV (OPNFV)*. Accessed: Jan. 15, 2019. [Online]. Available: https://www.opnfv.org/

[7] C. Lan, J. Sherry, R. Popa, S. Ratnasamy, and Z. Liu, "Embark: Securely outsourcing middleboxes to the cloud," in *Proc. USENIX NSDI*, 2016, pp. 255–273.

[8] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica, "Multi-resource fair queueing for packet processing," in *Proc. ACM SIGCOMM*, 2015, pp. 1–12.

[9] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes," in *Proc. USENIX NSDI*, 2013, pp. 227–240.

[10] S. Palkar *et al.*, "E2: A framework for NFV applications," in *Proc. ACM SOSP*, 2015, pp. 121–136.

[11] M. Ghaznavi, N. Shahriar, S. Kamali, R. Ahmed, and R. Boutaba, "Distributed service function chaining," *IEEE J. Sel. Areas Commun.*, vol. 35, no. 11, pp. 2479–2489, Nov. 2017.

[12] *Kernel Virtual Machine*. Accessed: Mar. 9, 2018. [Online]. Available: https://www.linux-kvm.org

[13] K. Hwang, X. Bai, Y. Shi, M. Li, W. Chen, and Y. Wu, "Cloud performance modeling and benchmark evaluation of elastic scaling strategies," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 1, pp. 130–143, Jan. 2016.

[14] X. Fei, F. Liu, H. Xu, and H. Jin, "Adaptive VNF scaling and flow routing with proactive demand prediction," in *Proc. IEEE INFOCOM*, 2018, pp. 486–494.

[15] *Open VSwitch*. Accessed: Sep. 28, 2017. [Online]. Available: https://openvswitch.org

[16] H. Yu, J. Yang, C. Fung, R. Boutaba, and Y. Zhuang, "ENSC: Multi-resource hybrid scaling for elastic network service chain in clouds," in *Proc. IEEE ICPADS*, 2018, pp. 34–41.

[17] X. Lin, Y. Yuan, D. Wang, and J. Yang, "Tetris: Optimizing cloud resource usage unbalance with elastic VM," in *Proc. IEEE IWQoS*, 2016, pp. 1–10.
[18] L. Popa, P. Yalagandula, S. Banerjee, J. Mogul, Y. Turner, and J. Santos, "Elasticswitch: Practical work-conserving bandwidth guarantees for cloud computing," in *Proc. ACM SIGCOMM*, 2013, pp. 351–362.
[19] C. Fuerst, S. Schmid, L. Suresh, and P. Costa, "Kraken: Online and elastic resource reservations for cloud datacenters," *IEEE Trans. Netw.*, vol. 26, no. 1, pp. 422–435, Feb. 2018.
[20] H. Yu, J. Yang, and C. Fung, "Elastic network service chain with fine-grained vertical scaling," in *Proc. IEEE GLOBECOM*, 2018, pp. 1–7.
[21] V. Sekar, N. Egi, S. Ratnasamy, M. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *Proc. USENIX NSDI*, 2012, p. 24.
[22] M. Vallentin, R. Sommer, J. Lee, and C. Leres, "The NIDS cluster: Scalable, stateful network intrusion detection on commodity hardware," in *Proc. ICRAID*, 2007, pp. 107–126.
[23] *The Bro Network Security Monitor*. Accessed: Dec. 11, 2018. [Online]. Available: https://www.bro.org/
[24] J. Nam, J. Seo, and S. Shim, "Probius: Automated approach for VNF and service chain analysis in software-defined NFV," in *Proc. ACM SOSP*, 2018, pp. 1–13.
[25] O. Hinojosa, J. Lucena, P. Ameigeiras, J. Munoz, D. Lopez, and J. Folgueira, "Automated network service scaling in NFV: Concepts, mechanisms and scaling workflow," *IEEE Commun. Mag.*, vol. 56, no. 7, pp. 162–169, Jul. 2018.
[26] A. Gember *et al.*, "Stratos: A network-aware orchestration layer for middleboxes in the cloud," 2013. [Online]. Available: https://arxiv.org/abs/1305.0209.
[27] C. Sun, J. Bi, Z. Meng, T. Yang, X. Zhang, and H. Hu, "Enabling NFV elasticity control with optimized flow migration," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 10. pp. 2288–2303, Oct. 2018.
[28] G. Miotto, M. Luizelli, Y. Shi, W. Cordeiro, and L. Gaspary, "Adaptive placement & chaining of virtual network functions with NFV-pear," *J. Internet Service Appl.*, vol. 10, p. 3, Feb. 2019.
[29] V. Eramo, E. Miucci, M. Ammar, and F. Lavacca, "An approach for service function chain routing and virtual funcation network instance migration in network funcation virtualization architectures," in *IEEE/ACM Trans. Netw.*, vol. 25, no. 4, pp. 1–33, Aug. 2017.
[30] A. Edmonds *et al.*, "An OCCI-compliant framework for fine-grained resource-aware management in mobile cloud networking," in *Proc. IEEE ISCC*, 2016, pp. 1306–1313.
[31] L. Cao, P. Sharma, S. Fahmy, and V. Saxena, "ENVI: Elastic resource flexing for network function virtualization," in *Proc. USENIX HotCloud*, 2017, p. 11.
[32] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes, "AGILE: Elastic distributed resource scaling for infrastructure-as-a-service," in *Proc. USENIX ICAC*, 2013, pp. 69–72.
[33] *Powering Virtual Network Services*. Accessed: Feb. 7, 2019. [Online]. Available: https://www.embrane.com/
[34] *Amazon EC2: Auto Scaling*. Accessed: May 21, 2017. [Online]. Available: http://aws.amazon.com/autoscaling/
[35] T. Knauth and C. Fetzer, "Scaling non-elastic applications using virtual machines," in *Proc. IEEE ICCC*, 2013, pp. 468–475.
[36] Z. Shen, S. Subbiah, X. Gu, and J. Wikes, "Cloudscale: Elastic resource scaling for multi-tenant cloud systems," in *Proc. ACM SOCC*, 2011, pp. 1–14.
[37] L. Yu, H. Shen, Z. Cai, L. Liu, and C. Pu, "Towards bandwidth guarantee for virtual clusters under demand uncertainty in multi-tenant clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 2, pp. 450–465, Feb. 2018.
[38] N. Vasic, D. Novakovic, S. Miucin, D. Kostic, and R. Bianchini, "DejaVu: Accelerating resource allocation in virtualized environment," in *Proc. ACM ASPLOS*, 2013, pp. 423–436.
[39] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Black-box and gray-box strategies for virtual machine migration," in *Proc. USENIX NSDI*, 2007, pp. 1–17.
[40] Z. Gong and X. Gu, "PAC: Pattern-driven application consolidation for efficient cloud computing," in *Proc. MASCOTS*, 2010, pp. 1–10.
[41] Z. Li and G. Wu, "Optimizing vm live migration strategy based on migration time cost modeling," in *Proc. ACM/IEEE ANCS*, 2016, pp. 99–109.
[42] J. Zhang, F. Ren, R. Shu, T. Huang, and Y. Liu, "Guaranteeing delay of live virtual machine migration by determining and provisioning appropriate bandwidth," in *IEEE Trans. Comput.*, vol. 65, no. 9, pp. 2910–2917, Sep. 2016.
[43] C. Li, D. Feng, and Y. Hua, "BAC: Bandwidth-aware compression for efficient live migration of virtual machines," in *Proc. IEEE INFOCOM*, 2017, pp. 177–189.
[44] *KVM CPU Hotplug*. Accessed: Jan. 10, 2018. [Online]. Available: https://www.linux-kvm.org/page/CPUHotPlug
[45] *KVM Memory Ballooning*. Accessed: Jan. 5, 2018. [Online]. Available: https://www.linux-kvm.org/page/Projects/auto-ballooning
[46] H. Ballani, P. Costa, T. Karaginnis, and A. Rowntron, "Towards predictable datacenter networks," in *Proc. ACM SIGCOMM*, 2011, pp. 242–253.
[47] D. Xie, N. Ding, C. Hu, and R. Kompella, "The only constant is change: Incorporating time-varying network reservations in data centers," in *Proc. ACM SIGCOMM*, 2012, pp. 199–210.
[48] M. Zhani, Q. Zhang, G. Simon, and R. Boutaba, "VDC planner: Dynamic migration-aware virtual data center embedding for clouds," in *Proc. IEEE IM*, 2013, pp. 18–25.
[49] *Caida*. Accessed: Oct. 11, 2017. [Online]. Available: https://www.caida.org/
[50] L. Cao, P. Sharma, S. Fahmy, and V. Saxena, "NFV-Vital: A framework for characterizing the performance of virtual network functions," in *Proc. IEEE NFV-SDN*, 2017, pp. 93–99.
[51] *Snort: An Open-Source, Free and Lightweight Network Intrusion Detection System (NIDS) Software*. Accessed: Feb. 1, 2018. [Online]. Available: https://www.snort.org/
[52] F. Wang, R. Ling, J. Zhu, and D. Li, "Bandwidth guaranteed virtual network function placement and scaling in datacenter neworks," in *IEEE IPCCC*, 2015, pp. 1–8.
[53] T. Benson, A. Akella, and D. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. ACM SIGCOMM*, 2010, pp. 267–280.
[54] *Linux Containers: LXC*. Accessed: May 6, 2018. [Online]. Available: https://linuxcontainers.org/
[55] *Docker: An Open Platform for Distributed Applications for Developers and Sysadmins*. Accessed: May 5, 2018. [Online]. Available: https://www.docker.com/

Hui Yu (Student Member, IEEE) received the B.Sc. degree from Chongqing University and the Ph.D. degree from the Department of Computer Science and Technology, Tsinghua University. His major research interests include network function virtualization, cloud computing, and datacenter network.

Jiahai Yang (Member, IEEE) received the B.Sc. degree in computer science from Beijing Technology and Business University and the M.Sc. and Ph.D. degrees in computer science from Tsinghua University, Beijing, China, where he is a Professor with the Institute for Network Sciences and Cyberspace. His research interests include network management, network measurement, Internet routing and applications, cloud computing, and big data applications. He is a senior member of IEEE and a member of ACM.

Carol Fung received the bachelor's and master's degrees in computer science from the University of Manitoba, Canada, and the Ph.D. degree in computer science from the University of Waterloo, Canada. She has been a visiting scholar with POSTECH, South Korea, a Software Engineer Intern with Google, and a Research Intern with BlackBerry. Her research interests include collaborative intrusion detection networks, social networks, security issues in mobile networks and medical systems, location-based services for mobile phones, and machine learning in intrusion detection. She was a recipient of the Young Professional Award in IEEE/IFIP IM 2015, the Alumni Gold Medal of University of Waterloo in 2013, the Best Dissertation Awards in IM 2013, the Best Student Paper Award in CNSM 2011, and the Best Paper Award in IM 2009. She received numerous prestige awards and scholarships, including the Google Anita Borg scholarship, the NSERC Postdoc Fellowship, the David Cheriton Scholarship, the NSERC Postgraduate Scholarship, and the President's Graduate Scholarship.