Optimizing NFV Chain Deployment in Software-Defined Cellular Core

Jiaqi Zheng, *Member, IEEE*, Chen Tian, Haipeng Dai, *Member, IEEE*, Qiufang Ma, Wei Zhang, *Member, IEEE*, Guihai Chen, *Senior Member, IEEE*, and Gong Zhang, *Member, IEEE*

***Abstract*— Today's cellular core relies on a few expensive and dedicated hardware racks to connect the radio access network and the egress point to the Internet, which are geographically placed at fixed locations and use the specific routing policies. This inelastic architecture fundamentally leads to increased capital and operating expenses, poor application performance and slow evolution. The emerging paradigm of Network Function Virtualization (NFV) and Software Defined Networking (SDN) bring new opportunities for cellular networks, which makes it possible to flexibly deploy service chains on commodity servers and fine-grained control the routing policies in a centralized way. We present a two-stage optimization framework *Plutus*. The network-level optimization aims to minimize the service chain deployment cost, while the server-level optimization requires to determine which Virtualized Network Function (VNF) should be deployed onto which CPU core to balance the CPU processing capability. We formulate these two problems as two optimization programs and prove their hardness. Based on parallel multi-block ADMM, we propose a $(\delta, 2)$-bicriteria approximation algorithm and a learning-based algorithm to address two cases whether the flow information and the resource consumption can be known as a priori, respectively. Large-scale simulations and DPDK-based OpenNetVM platform show that Plutus can reduce the capital cost by 84% and increase the throughput by 36% on average.**

***Index Terms*— Software Defined Networking (SDN), Network Function Virtualization (NFV), approximation algorithm, ADMM.**

I. Introduction

A. Motivation

CELLULAR core is a critical piece of the infrastructure and provides fundamental cellular-specific functions such as user authentication, mobility management and session management, etc. It also requires to support various middlebox services to implement per-user accounting and charging rules of voice calls [2]. However, today's cellular infrastructure are experiencing explosive growth in mobile connected devices. A Report from Cisco suggested that there would be 3 billion IoT devices and around 11.6 billion mobile connected devices by 2020 [3]. In the meantime, the growth rate of signal traffic is more than 50% faster than that of data traffic [4], which together creates huge stresses on the cellular core.

On one hand, in order to response the rapid growth of cellular traffic, the providers have to purchase and deploy more expensive and dedicated hardware racks, which inevitably leads to unfavorable capital and operating expenses. On the other hand, current architecture heavily relies on these dedicated middleboxes [5] to connect the radio access network and the egress point to the Internet, which are geographically placed at fixed locations and use the specific routing configurations. They cannot perfectly react to the changing traffic volume and dynamic policies. The renewal cycles of service innovation have to be prolonged and hindered by vendor support. Furthermore, with more and more hardware racks are deployed, the cellular network protocols become complex and intractable, leading to high management overhead.

Existing works attempt to address these issues above from different angles. CleanG [6] and LTE-Xtend [7] design a simplified control protocol in SDN-based cellular architecture to support emerging mobile devices and services. Usually a set of cellular-specific functions can be virtualized as a service chain, which consists of a sequence of VNFs. The work [8]–[11] consider service chain embedding problem, i.e., determining which VNF can be deployed onto which commodity server such that the packets can be sequentially processed by these VNFs and comply with the service chain constraints. However, the routing selection and VNF deployment are optimized separately, which leads to suboptimal deployment cost in nature. In addition, the traffic has to route from an upstream VNF located on one server to the downstream VNF located in another server to perform a specific function defined by a service chain, which takes up expensive bandwidth resource due to the traffic transmission between two servers. To save the network bandwidth consumption, NFVnice [12] and NFP [13] integrate the whole service chain into a server with multiple physical CPU cores, where one VNF can be fine-grained

Manuscript received June 23, 2019; revised October 17, 2019; accepted November 6, 2019. Date of publication December 31, 2019; date of current version February 19, 2020. This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFB1004700, in part by the National Natural Science Foundation of China under Grant 61802172, Grant 61602194, Grant 61772265, Grant 61872178, and Grant 61832005, in part by the Natural Science Foundation of Jiangsu Province under Grant BK20181251, in part by the Collaborative Innovation Center of Novel Software Technology and Industrialization, and in part by the Jiangsu Innovation and Entrepreneurship (Shuangchuang) Program. This article was presented in part at the IEEE IWQoS'19 [1]. *(Corresponding author: Guihai Chen.)*

J. Zheng, C. Tian, H. Dai, Q. Ma, and G. Chen are with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China (e-mail: jzheng@nju.edu.cn; tianchen@nju.edu.cn; haipengdai@nju.edu.cn; mg1633053@smail.nju.edu.cn; gchen@nju.edu.cn).

W. Zhang is with Microsoft Azure Networking, Redmond, WA 98052 USA (e-mail: wei.zhang.gb@gmail.com).

G. Zhang is with the Theory Lab, 2012 Labs, Huawei Technologies Company, Ltd., Hong Kong (e-mail: nicholas.zhang@huawei.com).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JSAC.2019.2959180

deployed onto one physical CPU core and different VNFs can share the same one [14]. However, the optimization framework of such service chain deployment with multiple CPU cores has not been explored in the existing literature.

In this paper we initiate the study of orchestrating the service chain deployment with multiple CPU cores, aiming to minimize the provisioning cost, which has the potential to overcome the drawbacks above. The novelty of our work lies in a comprehensive exploration and design based on the multi-core CPU framework, which to our knowledge has not been done before. The cellular traffic in our framework can be dynamically managed by a logically centralized controller in a fine-grained manner. By virtualization techniques, the operator can accelerate the innovation by shortening the renewal cycles of service chain deployment, reducing the capital cost and improving the scalability. In the first stage, we focus on a joint optimization of traffic routing and the service chain deployment. We can address two cases whether the flow information and the resource consumption can be known as a priori or not. Furthermore, for each VNF in a service chain, we seek to determine which VNF should be deployed onto which CPU core in the second stage, in order to balance the processing capacity and improve the throughput.

B. Our Contributions

Firstly, we propose a two-stage optimization framework *Plutus* for Minimum Provisioning Cost Problem (MPCP) and Multi-Core Deployment Problem (MCDP). The optimization program in the first stage aims to minimize the total provisioning cost of service chains, where the required CPU resource and cost for each service chain are given, such that each link's load cannot beyond its capacity and each server's resource cannot be overbooked. The service chain consists of a set of VNFs. The program in the second stage needs to determine which VNF should be placed onto which CPU core to balance the CPU processing capacity.

Our second contribution is a set of algorithms to solve MPCP and MCDP. We prove that MPCP and MCDP are both NP-hard, and thus focus on designing provably algorithms. Based on the multi-block ADMM, we first propose a $(\delta, 2)$-bicriteria approximation algorithm to solve MPCP and prove that it yields a constant approximation ratio of δ, while overbooking the CPU resource capacity at each server by at most a factor of 2, where δ is the number of pre-defined paths between source and destination. Further, we design a learning-based algorithm to address the case that the flow information and the resource consumption cannot be known as a priori. Finally, we propose a local search algorithm to solve MCDP with a constant approximation ratio of 2, which improves upon the results of randomized rounding by greedily moving each VNF at each CPU core.

Our third contribution is a comprehensive performance evaluation of our algorithms. Large-scale simulations using synthetic cellular network topologies show that our algorithms can reduce the total provisioning cost by 84%. Meanwhile, our algorithms run faster compared to state of the art and can provide the near optimal solution. We also develop a prototype on the DPDK-based OpenNetVM platform [15]. Experimental results show that our solution can increase the throughput by 36% on average.

II. Related Work

A. SDN-Based Cellular Core

SoftCell [16] and SoftMoW [17] both present a SDN-based cellular core architecture, where the signal and data traffic are explicitly managed by a logically centralized controller. The main difference between them is that the former aims to minimize the number of forwarding rules in the core switches, while the latter focuses on improving the performance for latency-sensitive applications. Another line of this work advocate to improve the design of control plane protocols. For example, CleanG [6] develops a novel protocol customized for emerging IoT services. LTE-Xtend [7] extends the existing protocols to support M2M communication. ProCel [18] increases the EPC capacity by optimizing the interaction between eNBs and EPC.

B. EPC Network Function Virtualization

The work in [19]–[22] virtualize the cellular-specific functions and guarantee that they can provide backward compatible function. SCALE [23] re-organizes the MME functionality into a front-end load balancer and back-end virtualized processing cluster to improve scalability. KLEIN [10] routes the traffic to the available EPC instances located in geographically distributed data centers and manages virtualized EPC resources. PEPC [11] decomposes the traditional EPC functions into different components and reduces frequent communication by eliminating the duplicated device states.

C. VNF Deployment and Scheduling

VNF-P [24] propose a hybrid VNF deployment model to allocate physical resources, i.e., network services can be provided by a mixture of traditional dedicated hardware and VNFs. As for the fully virtualized environment, Ghaznavi *et al.* [25] present a model to minimize the operational cost and provide elastic services. Furthermore, Cohen *et al.* [8] develop an approximation algorithm to minimize the distance cost between the clients and VNFs such that the capacity constraint of single resource should be satisfied. To reduce the CPU overhead, E2 [26] integrates network functions into a shared address space and executes VNFs in a run to completion manner by one thread. NFP [13] accelerates the packet processing by orchestrating VNFs in a parallel manner. NFVnice [12] proposes a network functions scheduling framework on the shared CPU cores to achieve rate-cost fairness.

D. Reinforcement Learning-Based VNF and Service Chain Deployment

Recent solutions to VNF and service chain deployment have integrated with reinforcement learning technologies.

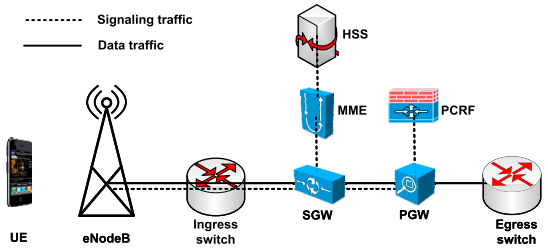


Fig. 1. Today's cellular network architecture.

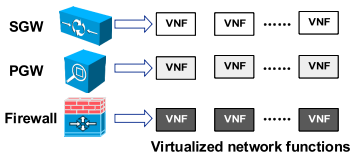


Fig. 2. Legacy hardware devices can be virtualized into multiple software instances (virtualized network functions).

Kim and Kim [27] propose a dynamic service function chaining algorithm through reinforcement learning. They can predict the physical resources usage in each node and virtual resource usage consumed by each service function. However, they only take resource usage into consideration. To maximize the benefit of the service provider, Sun *et al.* [28] investigate service function chain deployment in a dynamic network, which uses Q-learning to learn the network topology and resource usage and produces the output for the alternative paths. Combining deep learning with reinforcement learning, Khezri *et al.* [29] aim to minimize the placement cost while maximizing the number of admitted services. They consider the reliability requirement of the service in the formulation and propose a deep Q-Network method for dynamic reliability-aware VNF placement problem.

Today's cellular network architecture [30] is briefly shown in Fig. 1. The dotted lines and solid lines represent signaling traffic and data traffic, respectively. It mainly consists of the Radio Access Network (RAN) and the Evolved Packet Core (EPC). The RAN is a radio interface that connects User Equipment (UE) and eNodeBs (i.e., base stations). Once the traffic from UE arrives at eNodeBs, it will be forwarded to the EPC, where the EPC consists of a set of hardware racks such as Mobility Management Entity (MME), Serving Gateway (SGW), PDN Gateway (PGW), Home Subscriber Database (HSS) and Policy Charging Rules Function (PCRF). The MME handles all the signaling traffic from the UEs and the eNodeBs, and is responsible for user authentication, mobility management and session management. The SGW and PGW process all the data traffic. The SGW forwards the data traffic from the eNodeBs to the PGW, and PGW queries PCRF for setting the charging rules. The PGW then forwards the data traffic to a specific egress switch to the Internet.

This inelastic architecture suffers from poor scalability [31], high management complexity [32] and capital costs [33]. To address these issues, the legacy hardware devices can be virtualized into multiple software instances as shown in Fig. 2. Based on this, we propose a two-stage optimization framework. The network-level optimization in the first stage is

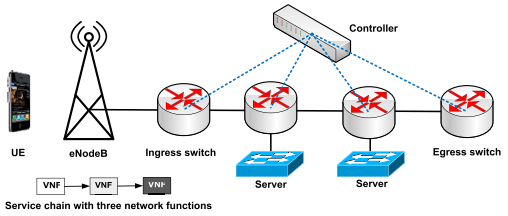


Fig. 3. The network-level optimization in the first stage of Plutus.

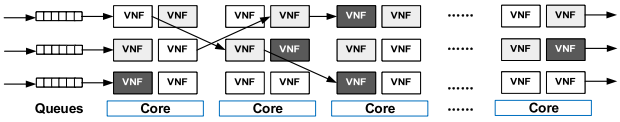


Fig. 4. The server-level optimization in the second stage of Plutus.

based on SDN architecture shown in Fig. 3, where a logically centralized controller has a global view and is responsible for directing the mobile traffic passing through a service chain in the data plane. Given the flow demand and the pre-defined service policies, the controller requires to install the optimal routing and determine the service chain deployment with the objective of minimizing the total cost, such that each link's load in the network cannot beyond its capacity and each server's CPU resource cannot be overbooked. Also the controller can use machine learning techniques to predict the flow information and the possible resource consumption to make this framework practical. Furthermore, the server-level optimization in the second stage needs to determine which VNF should be placed onto which CPU core to balance the CPU processing capacity and improve the throughput.

III. An Optimization Framework

A. Provisioning Model and Problem Formulation

Before formulating the problem, we first present our network model. A network is a directed graph $G = (V \cup L, E)$, where V is the set of switches, L is the set of servers and E the set of links with capacities b_e. Each flow f is associated with a demand d_f, splitted at the ingress switch v_f^+ among the possible path set P_f and routed to the egress switch v_f^-. Before going out to the egress switch, each flow f should pass through a specific service chain i deployed at least a server. In the first stage of Plutus, the flow demand d_f and the service policy $\alpha_{f,i}$ (which flow should pass through which service chain) is known, and which service chain should be placed onto which server needs to be determined. In the second stage of Plutus, we need to determine which network function is required to be placed onto which CPU core so as to balance the processing capability of multiple CPU cores. For convenience, we summarize important notations in Table I. What's new in the cellular core is that the request of application flows may involve both signaling function and data-plane function. For this type of requests, the traffic transmission starts only if the deployment of both the signaling functions and the data-plane functions are ready. Otherwise, it will result in incorrectness.

TABLE I
KEY NOTATIONS IN THIS PAPER

F	The set of flows f
V	The set of switches v
E	The set of links e
L	The set of servers l.
G	The acyclic directed network graph $G = (V \cup L, E)$
SC	The set of service chains i.
NF	The set of network functions j.
K_l	The set of CPU cores at the server l.
$r_{i,l}$	The required CPU resource for the service chain i at the server l
R_l	The total CPU resource at server l
$c_{i,l}$	The provisioning cost for the service chain i at the server l
b_e	The bandwidth capacity of link e
P_f	The set of possible paths for flow f from ingress switch v_f^+ to egress switch v_f^-
d_f	The demand of the flow f
$\alpha_{f,i}$	The indicator variable that equals 1 if flow f should pass through service chain i and 0 otherwise
$\beta_{i,j}$	The indicator variable that equals 1 if network function j belongs to the service chain i and 0 otherwise
σ_j	The consumed CPU cycles per packet for network function j
$x_{i,l}$	The indicator variable that equals 1 if service chain i locates at the server l and 0 otherwise
$y_{f,p}$	The fractional flow demand for flow f on path p
$q_{j,k}$	The indicator variable that equals 1 if network function (or instance) j is placed onto the CPU core k and 0 otherwise

Based on the above model and definition, we first formulate the *Minimum Provisioning Cost Problem (MPCP)* as a program to solve the network-level optimization in the first stage of Plutus. The formulation is shown in (1) and that in the second stage is shown in (2). We will discuss them soon.

$$\text{minimize} \sum_{l\in L}\sum_{i\in SC} c_{i,l}\cdot x_{i,l} \tag{1}$$
$$\text{subject to (1a), (1b), (1c), (1d), (1e), (1f).}$$

The objective of formulation (1) aims to minimize the sum of provisioning cost $c_{i,l}$ in the whole network. Basically, we seek to find an optimal routing and service chain deployment schemes so as to minimize the total provisioning cost, such that each link's load cannot beyond its capacity and the CPU resource at each server cannot be overbooked.

$$\sum_{i\in SC} x_{i,l}\cdot r_{i,l} \le R_l, \quad \forall l \in L, \tag{1a}$$

For each server $l \in L$, constraint (1a) indicates that the sum of provisioning resource for each service chain must be less than or equal to the total resource R_l. The zero-one integer variable $x_{i,l}$ equals one when the service chain i is placed at the server l, and equals zero otherwise.

$$\sum_{f\in F} d_f \sum_{p\in P_f: e\in p} y_{f,p} \le b_e, \quad \forall e \in E, \tag{1b}$$

The LHS of constraint (1b) characterizes the load of total flows at link e, which must be less than or equal to its capacity. This optimization variable $y_{f,p}$ determines that the fractional flow demand for flow f on path p.

$$\sum_{i\in SC}\sum_{l\in p} \alpha_{f,i}\cdot x_{i,l} \ge y_{f,p}, \quad \forall f \in F,\ \forall p \in P_f, \tag{1c}$$

Constraint (1c) ensures that if the flow is routed on the path p, the service chain i corresponding to the flow f should be placed onto at least one of the servers on this path.

$$\sum_{p\in P_f} y_{f,p} = 1, \quad \forall f \in F, \tag{1d}$$

Constraint (1d) is the flow demand conservation constraint. The sum of all fractional flow demand among all the possible paths should equal to d_f.

$$x_{i,l} \in \{0,1\}, \quad \forall i \in SC,\ \forall l \in L, \tag{1e}$$
$$y_{f,p} \ge 0, \quad \forall f \in F,\ \forall p \in P_f, \tag{1f}$$

The zero-one integer variable $x_{i,l}$ indicates if service chain i can be placed onto the server l or not.

In the second stage of Plutus, we focus on server-level optimization. A service chain consists of a set of VNFs. The packets are sequentially processed from upstream VNF to downstream VNF. The maximum throughput of one service chain depends on that of the bottleneck VNF. Which service chain is deployed onto which server has been fixed in the first stage, we need to determine which VNF should be deployed onto which CPU core in the second stage. Given the solution $x_{i,l}$ and the computation cost (the product of packet arrival rate and consumed CPU cycles per packet) of different VNFs, the objective is to balance the processing capability of multiple CPU cores and improve service chain throughput. Once the server-level deployment is complete, the OS scheduler will assign CPU time for each running VNF proportional to its computation cost. Now we formulate the *Multi-Core Deployment Problem (MCDP)* as an optimization program shown in (2).

$$\text{minimize} \max_{k\in K_l, j\in NF_l} w_j \cdot q_{j,k} \tag{2}$$
$$\text{subject to } w_j = y_{f,p}\cdot d_f \cdot \alpha_{f,i}\cdot \beta_{i,j}\cdot \sigma_j, \forall j \in NF_l, \tag{2a}$$
$$\sum_{k\in K_l} q_{j,k} = 1, \quad \forall j \in NF_l, \tag{2b}$$
$$q_{j,k} \in \{0,1\}, \quad \forall j \in NF_l,\ \forall k \in K_l. \tag{2c}$$

The objective of formulation (2) aims to minimize the maximum CPU load on a physical core. The CPU load w_j is defined by the product of $y_{f,p}\cdot d_f$ and σ_j.

$$w_j = y_{f,p}\cdot d_f \cdot \sigma_j$$

where $y_{f,p}\cdot d_f$ is the flow rate (packet arrival rate) and σ_j is the consumed CPU cycles per packet. The constant parameters $\alpha_{f,i}$ and $\beta_{i,j}$ in constraint (2a) describes the correlation among the flow f, service chain i and network function j. The zero-one integer variable $q_{j,k}$ indicates that which network function j should be placed onto which CPU core k.

The majority of VNFs in the cellular core is stateful, which need to maintain different state information even for the same VNF. An example of such service chain is Non-Access

Stratum (NAS) procedures [30]: MME → HSS → MME → SGW → PGW → PCRF → PGW → SGW → MME → SGW → PGW → SGW → MME. There are multiple instances especially for MME, PGW and SGW, where different instances hold for different session information. The required number of VNF instances of a service chain depends on the specific interactive procedure in the cellular core. In addition, we need to handle both control traffic and data traffic. For example, the control traffic originates from the source node s and traverses NAS service chain. After the traversal, the control traffic reaches back to the source node s to notify that data path is setup. Then the data traffic can transmit traffic from the source s to the destination t. This is the main difference compared with traditional traffic engineering and VNF placement.

B. Hardness Analysis

We establish the hardness of MPCP and MCDP below.

Theorem 1: MPCP is NP-hard.

Proof: Consider a special case of MPCP that is illustrated in Fig. 5(b), where white nodes in the left side represent the sources, white nodes in the right side represent the destinations, and gray nodes represent the servers. All n flows from the source to the destination share one common path (connected by a string of gray nodes) and each server on this path has the identical resource capacity R. Each flow is associated with one service chain i, which is required to be deployed onto at least one server l ($l \in \{1, 2, \cdots, m\}$) to perform virtualized network functions. If the service chain i is deployed onto the server l, it will incur a provisioning cost $c_{i,l}$. The required CPU resource for the service chain i at the server l is $r_{i,l}$. Our objective aims to minimize the total provisioning cost such that the CPU resource capacity at each server cannot be overbooked.

We construct a reduction with polynomial time from the Generalized Assignment Problem (GAP) [34] to the special case of MPCP. As shown in Fig. 5(a), the GAP aims to assign n jobs to m machines, where each job can only be assigned to exactly one machine. If the job i is assigned to the machine l, the processing time units and the incurred cost in machine l is $r_{i,l}$ and $c_{i,l}$ respectively. A processing time bound T for each machine is given to limit the total processing time. The objective of GAP is to find a cost-minimizing assignment such that the total processing time on each machine is less than or equal to T. The job i, machine l and processing time bound T in GAP correspond to the service chain i, server l and CPU resource capacity R, respectively. Therefore, any feasible solution of GAP corresponds to the special case of MPCP in Fig. 5(b), and vice versa. ■

Theorem 2: MCDP is NP-hard, even for a server only consisting of two CPU cores with identical capacities.

Proof: Given a special case of MCDP, where we have only one server consisting of two CPU cores with identical capacities, we construct a polynomial reduction from the set partition problem [35] to it. Consider a partition instance $\mathbb{A}$ consisting of n items, each with a value a_i, $a_j \in \mathbb{R}, j \in \{1, 2, \ldots, n\}$. The objective is to partition $\mathbb{A}$ into two subsets $\mathbb{A}_1$ and $\mathbb{A}_2$

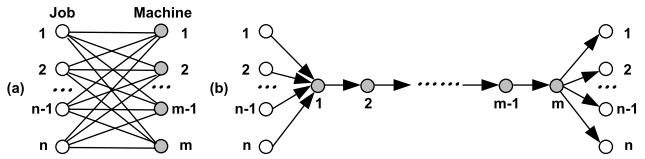


Fig. 5. Reduction from GAP to MPCP.

($\mathbb{A}_1 \cup \mathbb{A}_2 = \mathbb{A}$ and $\mathbb{A}_1 \cap \mathbb{A}_2 = \emptyset$) such that $|A_1 - A_2|$ is minimized, where A_1 and A_2 denote the sums of the elements in each of the two subsets $\mathbb{A}_1$ and $\mathbb{A}_2$. Accordingly, for each item in set $\mathbb{A}$ we introduce one network function j, where $w_j = a_j$. There are n items in total and thus we introduce n network functions in MCDP.

The partition results indicate that which network function should be placed onto which CPU cores. Therefore, any partition with minimum difference between set A_1 and A_2 corresponds to MCDP with two identical CPU cores, and vice versa. The network functions placed onto the first CPU core forms one set of the partition, and that placed onto the second CPU core forms the other. ■

IV. Network-level Optimization Algorithms

In this section, we design two network-level optimization algorithms in Plutus. The first one is a bicriteria approximation algorithm, where the flow information and the resource consumption can be known as a priori. This algorithm can be as the baseline to measure the performance of the second algorithm. The second algorithm is a Q-learning based algorithm, where the priori information is not required and more general than the first one.

A. A bicriteria approximation algorithm

The mixed integer program (1) aims to minimize the total provisioning cost, which can be relaxed to a linear program by replacing the constraint (1e) with $x_{i,l} \geq 0$. Since constraint (1c) and (1d) hold, $x_{i,l}$ are in fact real numbers between 0 to 1. The optimal fractional solutions $\{\tilde{x}_{i,l}\}$ and $\{\tilde{y}_{f,p}\}$ of the relaxed LP of (1) can be obtained in polynomial time using standard solvers. However, solving this program is time-consuming especially in large-scale production networks with thousands of flows. Thus we set out to find a scalable algorithm to solve this program instead. Inspired by the framework of multiple-block ADMM [36], we develop a proximal Jacobian ADMM algorithm that can converge to an optimal solution at the rate of $o(\frac{1}{t})$ in Algorithm 1, where t is the number of iteration times. As parts of the constraints in program (1) are inequalities and the variables $x_{i,l}$ are coupled together, we require to transform program (1) to program (5) in order to apply 5-block ADMM (line 1). The detailed transformation is illustrated in Appendix A. Furthermore, we initialize the original variables $\boldsymbol{y}$, the introduced auxiliary variables $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, $\boldsymbol{\gamma}$, $\hat{\boldsymbol{x}}$, and multipliers $\boldsymbol{\theta}$, $\boldsymbol{\varphi}$, $\boldsymbol{\sigma}$ to zero (line 2) and solve each subprogram in parallel (lines 3-5).

Based on the fractional solution in Algorithm 1, we design a $(\delta, 2)$-bicriteria approximation algorithm that rounds the fractional solution to a feasible integer solution. The complete

Algorithm 1: A Proximal Jacobian ADMM Algorithm

Input : Network graph $G = (V \cup L, E)$; the set of flow f; the set of service chain i.
Output: A fractional solution $\{\tilde{x}_{i,l}\}$ and $\{\tilde{y}_{f,p}\}$ to the relaxed LP of (1)

1 Transform the program (1) to the program (5).
2 Initialize the variables $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, $\boldsymbol{\gamma}$, $\hat{\boldsymbol{x}}$, $\boldsymbol{y}$ and multipliers $\boldsymbol{\theta}$, $\boldsymbol{\varphi}$, $\boldsymbol{\sigma}$ to zero.
3 **for** $t = 1, 2, \cdots$ **do**
4 Update $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, $\boldsymbol{\gamma}$, $\hat{\boldsymbol{x}}$, $\boldsymbol{y}$ from programs (7), (8), (9), (10), (11) in parallel.
5 Update $\boldsymbol{\theta}$, $\boldsymbol{\varphi}$, $\boldsymbol{\sigma}$ from equations (12), (13), (14).

Algorithm 2: A Bicriteria Approximation Algorithm

Input : Network graph $G = (V \cup L, E)$; the set of flow f; the set of service chain i.
Output: A solution $\{\hat{x}_{i,l}\}$ and $\{\tilde{y}_{f,p}\}$ to (1).

1 Obtain the optimal fractional solution $\{\tilde{x}_{i,l}\}$ and $\{\tilde{y}_{f,p}\}$ to the relaxed LP of (1).
2 $S = U = \emptyset$
3 **for** *each* $f \in F$ **do**
4 **for** *each* $p \in P_f$ **do**
5 **if** $\tilde{y}_{f,p} > 0$ **then**
6 $S = S \cup \tilde{y}_{f,p}$
7 **for** *each* $l \in L$ **do**
8 $k_l = \left\lceil \sum_{i \in SC} \tilde{x}_{i,l} \right\rceil$
9 Sort $\tilde{x}_{i,l}$ in descending order according to $r_{i,l}$ into set X
10 **for** $j = 1$ *to* k_l **do**
11 $U = U \cup u_l^j$
12 $\Phi = 1$
13 **repeat**
14 Get the first element $\tilde{x}_{i,l}$ from set X
15 $\Delta = \max(\tilde{x}_{i,l}, \tilde{x}_{i,l} - \Phi)$
16 $\Phi = \Phi - \Delta$
17 $u_{i,l}^j = \Delta$
18 $\tilde{x}_{i,l} = \tilde{x}_{i,l} - \Delta$
19 **if** $\tilde{x}_{i,l} = 0$ **then**
20 Remove $\tilde{x}_{i,l}$ from set X
21 **until** $\Phi = 0$;
22 **for** *each* $\tilde{y}_{f,p} \in S$ **do**
23 Determine service chain i corresponding to flow f
24 **for** *each* $u_l^j \in U$ **do**
25 **if** $u_{i,l}^j > 0$ **then**
26 Add an edge $(\tilde{y}_{f,p}, u_l^j)$ with weight $c_{i,l}$ into E'
27 Construct a bipartite graph (S, U, E') and compute a complete matching M with the minimum total weight.
28 **if** *there exists an edge in the matching* M **then**
29 $\hat{x}_{i,l} = 1$
30 **else**
31 $\hat{x}_{i,l} = 0$

algorithm is shown in Algorithm 2. We now explain the high-level working of this algorithm. We first obtain the optimal fractional solution $\{\tilde{x}_{i,l}\}$ and $\{\tilde{y}_{f,p}\}$ to the relaxed LP of (1) (line 1). For the solution $\{\tilde{y}_{f,p}\}$, it is already a feasible solution, while for the solution $\{\tilde{x}_{i,l}\}$. For the solution $\{\tilde{x}_{i,l}\}$, it indicates that the service chain i can be fractionally placed onto all servers l on its path p ($l \in p$), and accordingly the required CPU resource is proportional to the fractional solution. We require to round it to an integer solution by constructing a complete bipartite graph (S, U, E') (lines 7-31). Initially, the set S and U are both empty set (line 2). We first add each solution $\tilde{y}_{f,p}$ into the set S (lines 3-6). And then, for each server, we assign k_l slots to accommodate the fractional solution $\{\tilde{x}_{i,l}\}$ one by one according to its required resource $r_{i,l}$ (line 8). Next we add u_l^j into set U, whose cardinality is the product of the number of servers l and the number of assigned slots k_l (line 11). Note that $\tilde{x}_{i,l}$ could be split into two adjacent slots, and we use notation $u_{i,l}^j$ to represent the corresponding parts. The value of $u_{i,l}^j$ is calculated from a loop procedure (lines 13-21). If $u_{i,l}^j$ is greater than zero, we add an edge $(\tilde{y}_{f,p}, u_l^j)$ with weight $c_{i,l}$ into E' and finish the construction procedure of complete bipartite graph (S, U, E') (line 22-26). Based on this graph, we compute a complete matching M with the minimum total weight (line 27). If there exists an edge in the matching M, we set $\hat{x}_{i,l}$ is equal to one that indicates the service chain i can be placed onto server l; otherwise, we set it to zero (lines 29-31).

Now we analyze the performance of the algorithm by introducing the related definition.

Definition 1: Let OPT_1 be the optimal solution to (1), which gives a lower bound of total provisioning cost.

Theorem 3: Algorithm 2 is a $(\delta, 2)$-bicriteria approximation algorithm, which has a constant approximation ratio of δ, while overbooking the resource capacity at each server by at most a factor of 2, where δ is the number of pre-defined paths between ingress and egress switch.

The proof can be found in Appendix B.

B. A Q-learning based algorithm

The bicriteria approximation algorithm requires all network flow's information and resource consumption as a priori, which cannot always hold in practice. In this part, we design a reinforcement learning [37] based service chain deployment algorithm. In our model, each flow enters the network unpredictably, which also means that we need to deploy the required service chain for each flow with the time horizon. In addition, the consumed CPU resource and provisioning cost by a certain service chain cannot be known until it is deployed on a server. The agent in reinforcement learning learns the provisioning cost and the resource usage actively and makes service chain deployment decision in an dynamic manner.

Reinforcement learning is one of the most important approach in machine learning [38]. What distinguishes reinforcement learning from supervised and unsupervised learning lies in its interaction with the environment. Unlike most of other machine learning methods, it discovers the optimal strategy independently. Through trial-and-error and receiving feedback from the environment, the goal of the agent is to maximize the value of the reward function. The basic principle of reinforcement learning is that, if the agent's action results in the positive reward from the environment, the tendency of the agent to produce this action will be strengthened in the future. Otherwise, it will be weakened. This technology has already been proved to be applicable in many fields.

Here we choose Q-learning [39], a common reinforcement learning algorithm to solve our dynamic service chain deployment problem. Q means a matrix with M rows and N columns, which stores the expectation of gains by taking action a ($a \in A$) in state s ($s \in S$) at a certain time, where the notation M and N represent the number of states in S and the actions in A, respectively. Q-table is used to evaluate the reward of taking an action in a specific state. It actually stores the memory of agent. The goal of the training phase is to make the Q-table converge. Q-table has been proved to converge for continuous decision making problems in environment that satisfies the requirements of reinforcement learning [39].

The reasons why we use Q-learning are as follows. Firstly, most machine learning algorithms are time-consuming in the training phase. Whereas Q-learning can greatly reduce the training time and it's also easier to understand. Most importantly, the reinforcement learning usually uses Markov decision process as a mathematical description model. Since the arrival of flows is unpredictable in our problem, the state transition matrix in Markov decision process of system modeling is unknown. Q-learning uses value iteration algorithm to determine the optimal strategy. There is no need to know the state transition matrix in advance.

When using Q-learning algorithm, what's the most important is to transform our problem into a Q-learning model. That's to say, we need to define the state set, action set and reward function. Based on our model, we define our own state set, action set and reward function as follows.

State Set: Since the provisioning cost for different service chains at a certain server is different. It depends on the type of service chains. When there is a new flow f arrives, with a required service chain i to pass through, the key to choose which server to deploy the service chain i is the provisioning cost and resource capacity on each server. So we use the service chain i that the current flow f needs to pass through as the current state. Suppose there are M types of service chain to deploy, there are M states in total. Concretely,

$$S = \{s_1, s_2, s_3, \ldots, s_M\}.$$

Action Set: For a given flow f with a required service chain i to pass through, the service chain i should be placed onto which server needs to be determined. In our problem, the action means that how to choose a server for service chain placement at a certain state. For a topology $G = (V \cup L, E)$ with g nodes including servers and switches totally, suppose v represents the number of switches. There are N actions in total, where $N = g - v$.

$$A = \{a_1, a_2, a_3, \ldots, a_N\}.$$

Reward function: The design of the reward function characterizes the optimization objective in the system. In the fist stage, we aim to minimize the total provisioning cost of service chains. For different actions, we will set different reward values. When the required CPU resource for a service chain i exceeds the resource capacity of the server it choose, R is set to a fixed negative value (we set it to be -100 in our experiments). When the action a in state s does not exceed the resource capacity of the server it choose, we have

$$R(s,a) = \frac{\sum_{l \in L} \sum_{i \in SC} c_{i,l} \cdot \hat{x}_{i,l}}{count} - c_{i,l}$$

where the variable $count$ represents the number of service chains that have been successfully deployed before. In this way, the variable R is doomed to increase its value if the agent chooses a server with a relatively low provisioning cost.

In Q-learning algorithm, the key point is to produce the state-action Q-table. When the agent belongs in a certain state s, the next action a is chosen from Q-table as the following equation.

$$a = \arg\max_{\hat{a} \in A} Q(s, \hat{a}). \tag{3}$$

Since this behavior may be easy to fall into a local optimum, we explore the next action using ϵ-greedy policy. This policy randomly and uniformly chooses an action with a small probability ϵ, and select a best current action according to Q-table according to equation (3) with probability $1 - \epsilon$. This mechanism is able to jump out of local optimum by exploring the environment with a small probability. Note that the parameter ϵ is not fixed and unchanged all the time in our online algorithm. We set ϵ equals $\frac{1}{count}$, where $count$ represents the number of service chains that have been successfully deployed before. The parameter ϵ will get larger at the beginning and decreased gradually, which is good for a better exploration in the earlier stage of learning, while depending more on experience in the later stage.

After a certain action a is selected, the agent will execute this decision. And then it enters into a new state s'. Meanwhile, it will get a reward $R(s,a)$ from the environment to evaluate action a in state s. Next the agent updates Q-table as follows:

$$Q(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma \cdot Q' - Q(s,a)] \tag{4}$$

where $Q' = \max_{a'} Q(s', a')$, α is the learning rate and γ is a discount factor. Generally speaking, a larger α indicates less impact from the previous training results, while a smaller γ represents the function pays more attention to current benefit $R(s,a)$ and ignores experience. Note that any two states are independent each other in our problem due to the independent arrival of flows. Here we set the parameter γ to be zero.

The complete algorithm is shown in Algorithm 3. Now we explain the main idea of the algorithm. We first initialize key variables before the arrival of flows (lines 1-2). For each arrived flow, we first determine the current state s according

Algorithm 3: A Q-Learning Based Deployment Algorithm

Input : Network graph $G = (V \cup L, E)$; the set of arrived flows f; their required service chain i; the state set S, action set A and reward function R.

Output: A solution $\{\hat{x}_{i,l}\}$ and $\{\tilde{y}_{f,p}\}$ to (1).

1 Initialize the value in $Q(s,a)$ as zero, for all $s \in S, a \in A$
2 Initialize $\epsilon = 0$, $count = 0$
3 **for** *each arrival of flow* f **do**
4 Determine a current state s corresponding to the service chain i
5 Obtain $\{\tilde{y}_{f,p}\}$ by routing the flow f on its available paths and the demand of flow f on each path is proportional to the vacant capacity
6 Construct the path set $\tilde{P}_f = \{p | \tilde{y}_{f,p} > 0\}$
7 **for** *each* $p \in \tilde{P}_f$ **do**
8 Generate a random number r uniformly in the interval between 0 and 1
9 **if** $r \leq \epsilon$ **then**
10 Choose an action a randomly, i.e., select an arbitrary server $l \in p$ to deploy the service chain i
11 **else**
12 $a = \arg\max_{\hat{a} \in A} Q(s, \hat{a})$
13 Find the corresponding server l for action a
14 **if** $r_{i,l} \leq R_l$ **then**
15 $count = count + 1$
16 $\epsilon = \frac{1}{count}$
17 $R(s,a) = \frac{\sum_{l \in L} \sum_{i \in SC} c_{i,l} \cdot \hat{x}_{i,l}}{count} - c_{i,l}$
18 $\hat{x}_{i,l} = 1$
19 $R_l = R_l - r_{i,l}$
20 $\tilde{P}_f = \tilde{P}_f \setminus p$
21 **else**
22 Reject to deploy service chain i for flow f;
23 $R(s,a) = -100$
24 update $Q(s,a)$ according to (4)

to the type of the service chain that this flow requires to pass through (line 4). We split the flow on its pre-defined paths and the demand on each path is proportional to the remaining capacity (line 5). This can be implemented by the weighted ECMP [40] and supported by today's commercial switches. Next we determine the action a to be taken using ϵ-greedy policy (lines 8-12). Action a will match a server l to deploy the service chain i (line 13). If the remaining resource on the server l is more than the required, this service chain can be placed on the server l (lines 14-20). At the same time, we compute the reward (line 17) and update R_l and $\tilde{P}_f$ (lines 19-20). Otherwise, the service chain will not be deployed successfully, the agent will get a negative reward (lines 21-23). At last, we update Q-table (line 24) and the algorithm begins to process the next flow.

V. Server-level Optimization Algorithms

In this section, we design server-level optimization algorithms in Plutus. Given the routing configurations and the resulting flow rate from the network-level optimization algorithms, the server-level optimization seek to find a network function deployment solution to balance the processing capability of multiple CPU cores.

Algorithm 4: A Randomized Algorithm

Input : The set of CPU cores at the server l; the consumed CPU cycles for network function j; the indicator variables $\alpha_{f,i}$ and $\beta_{i,j}$; the fractional flow demand $\tilde{y}_{f,p}$ for flow f on path p.

Output: A solution $\{\hat{q}_{j,k}\}$ to (2).

1 Obtain the optimal fractional solution $\{\tilde{q}_{j,k}\}$ to the relaxed LP of (2).
2 **for** *each* $j \in NF_l$ **do**
3 $K'_l = \emptyset$
4 **for** *each* $k \in K_l$ **do**
5 $\hat{q}_{j,k} = 0$
6 $K'_l = K'_l \cup k$
7 $l_{j,k} = \sum_{k' \in K'_l} \tilde{q}_{j,k'}$
8 Generate a number r in (0,1] uniformly at random
9 Find $\hat{p}$ such that $r \leq l_{j,k}$ and $l_{j,k} - r$ is minimum
10 $\hat{q}_{j,k} = 1$

As shown in Algorithm 4, we first obtain the optimal fractional solution $\{\tilde{q}_{j,k}\}$ to the relaxed LP of (2) by replacing the constraint (2c) with $q_{j,k} \geq 0$ (line 1). For each $j \in NF_f$, we apply randomized rounding to obtain an integer solution $\{\hat{q}_{j,k}\}$ (lines 2–10). To ensure that only one CPU core is chosen for a network function $j \in NF_f$, the optimal fractional solution can be viewed as partitioning the interval $[0, 1]$ to intervals of lengths $\{\tilde{q}_{j,k}\}$ (lines 4–7). A real number is generated uniformly at random in $(0, 1]$ and the interval in which it lies determines the CPU core (lines 8–10).

Before analyzing the performance of Algorithm 4, we introduce the following definition.

Definition 2: Let OPT_2 be the optimal solution to (2), which gives a lower bound of maximum CPU load.

Theorem 4: [34] Algorithm 4 outputs a feasible solution with maximum CPU load bounded by $\mathcal{O}(\log |K_l|) \cdot OPT_2$ with the probability $1 - \frac{1}{|K_l|^2}$, where $|K_l|$ is the number of CPU cores at the server l.

The proof of Theorem 4 can be found in [34].

In spite of the guaranteed approximation ratio in Algorithm 4, it occasionally produces a bad solution with the probability $\frac{1}{|K_l|^2}$. Hence, we develop a local search algorithm that improves upon the randomized solutions of Algorithm 4 by greedily moving each network function to another CPU core with less load. The local search algorithm achieves a constant approximation ratio of 2.

We are now ready to describe our local search algorithm shown in Algorithm 5. We first run Algorithm 4 and obtain an initial solution $\{\hat{q}_{j,k}\}$ (line 1). Next we iteratively move

Algorithm 5: A Local Search Algorithm

Input : The set of CPU cores at the server l; the consumed CPU cycles for network function j; the indicator variables $\alpha_{f,i}$ and $\beta_{i,j}$; the fractional flow demand $\tilde{y}_{f,p}$ for flow f on path p.

Output: A solution $\{\hat{q}_{j,k}\}$ to (2).

1 Apply Algorithm 4 to obtain the initial solution $\{\hat{q}_{j,k}\}$.
2 **repeat**
3 $w^+ = \max_{k \in K_l}\{\sum_j w_j \cdot \hat{q}_{j,k}\}$
4 $w^- = \min_{k \in K_l}\{\sum_j w_j \cdot \hat{q}_{j,k}\}$
5 $g^+ = |K^+|$, where $K^+ = \{k | \sum_j w_j \cdot \hat{q}_{j,k} = w^+\}$
6 $g^- = |K^-|$, where $K^- = \{k | \sum_j w_j \cdot \hat{q}_{j,k} = w^-\}$
7 $\forall k^- \in K^-$
8 **for** *each* $k^+ \in K^+$ **do**
9 **for** *each* $j \in NF_l$ **do**
10 **if** $q_{j,k^+} = 0$ **then**
11 **continue**
12 Move network function j from k^+ to k^-
13 Re-calculate w^+ and g^+
14 **if** *the* w^+ *value or the* g^+ *value decreases* **then**
15 $q_{j,k^+} = 0$
16 $q_{j,k^-} = 1$
17 $NF_l = NF_l \setminus \{j\}$
18 **until** $NF_l = \emptyset$;

network function to another CPU core with less load to balance the CPU processing capability until we cannot find a better solution (lines 2-18). The notation w^+ and w^- indicate the maximum and minimum CPU load corresponding to the current solution $\{\hat{q}_{j,k}\}$ (lines 3-4), while g^+ and g^- indicate the number of CPU cores with maximum and minimum load, respectively (lines 5-6). For each network function j, we try to move it to CPU k^- since CPU k^- has the least CPU load currently (line 12). If this movement results in the decrease of w^+ value or g^+ value, we move network function j from CPU k^+ to k^- (lines 15-16). Finally we remove the network function j from the set NF_l and the algorithm enters into the next loop (line 17). Based on the analysis above, we have Theorem 5 and its proof can be found in [34].

Theorem 5: [34] After at most $|NF_l|$ iterations, Algorithm 5 terminates and approximates MCSP with a factor of 2, where $|NF_l|$ is the number of network functions at server l.

VI. Experimental Evaluation

We evaluate our two-stage optimization algorithms using both prototype implementation and large-scale simulation.

Benchmark schemes: We compare the following schemes with our algorithm.

- **HW**: The network function of each type relies on traditional hardware middleboxes.
- **Greedy**: Each service chain is greedily deployed onto the server with the minimum cost.

TABLE II
The Throughput With Different Packet Arrival Rates

Request rate (%)	20	40	60	80	100
OPT_2 (Mbps)	84.79	127.78	143.20	168.37	174.79
LSA (Mbps)	80.93	121.56	138.43	144.12	155.35
Random (Mbps)	75.13	97.60	101.21	102.90	104.61

TABLE III
The Throughput With Different Number of VNFs

Service chain length	8	16	32	64	128
OPT_2 (Mbps)	741.77	260.03	218.47	174.79	92.34
LSA (Mbps)	732.82	256.47	198.29	155.35	88.93
Random (Mbps)	703.03	201.77	141.10	104.61	50.21

- **Random**: For a service chain consisting of a set of VNFs, each VNF is randomly deployed onto one of the CPU cores.
- **BAA**: Our bicriteria approximation algorithm shown in Algorithm 2.
- **ODA**: Our Q-learning based algorithm shown in Algorithm 3.
- **LSA**: Our Local Search algorithm shown in Algorithm 5.
- **OPT**: The optimal solutions OPT_1 and OPT_2 for MPCP and MCDP in the integer program (1) and (2) obtained using branch and bound.

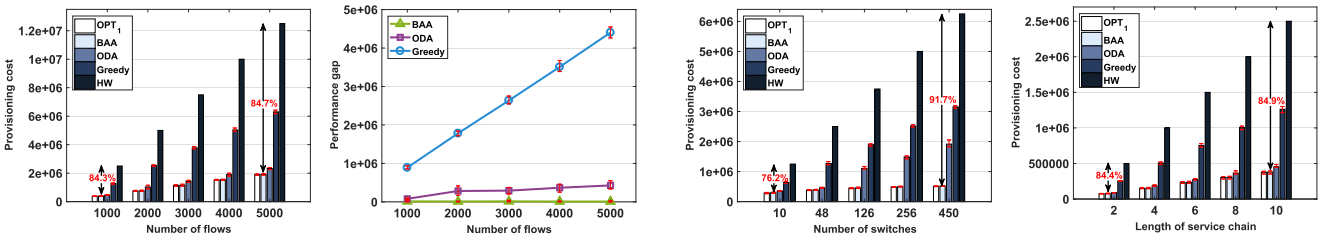
Unless stated otherwise, we configure ρ, w and ι to be 0.1, 0.02 and 1.0 respectively in Algorithm 1 as suggested in [36].

A. Implementation and Testbed Emulations

1) Implementation: We develop a prototype of our algorithms on the DPDK-based OpenNetVM platform [15], where the polling mechanism is used in RX and TX threads for receiving and sending packets from NIC. Now we describe how to perform VNF deployment in the service chains in our experiments. We first obtain solutions to MPCP and MCDP using Algorithm 2 and 5 respectively. According to these solutions, we bind each VNF to a dedicated CPU core. The `CORELIST` parameter in OpenNetVM specify the index of CPU core, and the index parameters `SERVICE_ID` and `DST` indicate two adjacent VNFs in a service chain, i.e., once the packets have already been processed by an upstream VNF indexed by `SERVICE_ID`, they would be forwarded to a downstream VNF indexed by `DST`.

2) Testbed Setup: In our experimental setup, we use two severs, each of which has dual Xeon(R) E5-2630 @ 2.40GHz CPUs (2x8 physical cores), an Intel 82599ES 10G dual port NIC and 128GB memory. Each server runs Ubuntu 14.04.3 with kernel version 3.19.0. We use `pktgen` to generate different UDP flows with 64-byte packet size in each run, where all of them are required to pass through a pre-defined ordered VNFs deployed onto different CPU cores. The VNFs we used perform forwarding and monitor functions, which form a linear service chain.

3) Experiment Results: We study the throughput variations with different packet arrival rates and different number of VNFs in Table II and Table III. In Table II, the number of



(a) Provisioning cost with different number of flows (b) Additive optimality gap with different number of flows (c) Provisioning cost with different number of switches (d) Provisioning cost with different length of service chain

Fig. 6. Total provisioning cost comparison.

running VNFs is fixed at 64. The 100% arrival rate corresponds to around 7 Gbps in our testbed. We can observe that the achieved throughput of LSA consistently outperforms that of Random by 31.5% on average when the packet arrival rate becomes larger. Specifically, LSA can improve the throughput by 48.5% compared with Random when the number of packet arrival rate is 100%. The improvement of LSA is more significant compared with Random. The reason is that the VNF deployment using LSA can better balance the resource consumption on each CPU core. Unbalanced VNF deployment leads to packet loss and hurts the throughput. Table III shows the throughput variations with different numbers of VNFs. Intuitively, more VNFs will consume more CPU resources. We vary the number of VNF from 2^3 to 2^7 at the increment of double times. LSA can reduce the throughput loss by 39.5% compared with Random. This demonstrates that LSA can mitigate the resource competition on a single CPU core.

B. Simulation

We also conduct extensive simulations to thoroughly evaluate our algorithms at scale.

1) Setup: In addition to the OpenNetVM experiments in our testbed, here we use a large-scale synthetic cellular network topology [41]. The topology can be divided into access layer, aggregation layer and core layer, where each layer includes a set of switches and servers. The access layer includes the base station clusters, each of which has ten base stations interconnected into a ring. The aggregation and core layer are complete graphs with τ and τ^2 servers respectively. In the aggregation layer, the $\frac{\tau}{2}$ switches are connected to $\frac{\tau}{2}$ clusters in the access layer respectively. The remaining switches in the aggregation layer are connected to the switches in the core layer one by one. We generate different numbers of flows to measure the performance in our experiments. We run our algorithms on a server with Intel(R) Xeon(R) CPU E5-2650 and 64 GB memory. Each data point is an average of at least 30 runs.

2) Experiment Results: We first investigate the total provisioning cost during the service chain deployment generated by Greedy, ODA and BAA compared with HW — the traditional hardware middlebox cost. In addition, we compare our algorithms against a branch and bound method that solves the program (1) optimally, denoted as OPT_1. The hardware middlebox cost used in our simulation comes from [42]. The VNF provisioning cost is the product of the baseline cost [33] and normalized CPU capacities. Tab. IV shows the normalized CPU capacities for five types used in our experiments.

TABLE IV
THE NORMALIZED CPU CAPACITIES IN GOOGLE'S CLOUD [43], [44]

	1	2	3	4	5
Normalized CPU capacities	0.25	0.50	0.75	0.80	**1.0**

We can see that in Fig. 6(a), as the number of flows increases, HW and Greedy yield significant cost, while that of BAA is below 2.0×10^6 all the time and can achieve near optimal. Making the deployment decision in an online manner, ODA performs worse than BAA. Specifically, the provisioning cost for HW, Greedy, BAA, ODA and OPT_1 is 1.25×10^7, 6.30×10^6, 1.92×10^6, 2.50×10^6, 1.92×10^6 and 1.90×10^6, when the number of flows is 5000. Furthermore, BAA can reduce the provisioning cost by 84.6% and 69.6% respectively, compared to HW and Greedy. This demonstrates that our algorithms take full advantage of virtualization and reduce the provisioning cost by flexibly deploying different VNFs.

Fig. 6(b) shows the additive optimality gap for different schemes — BAA, ODA and Greedy compared to OPT_1. For this simulation we vary the number of flows from 1000 to 5000. Intuitively, a larger additive optimality gap indicates more provisioning cost resulting from a worse solution. We can see that, as the number of flows increases, Greedy yields significantly larger optimality gap compared to BAA and ODA, where BAA can guarantee that the additive optimality gap is less than 1.3×10^4 and its provisioning cost is always less than 2.0×10^6. ODA can guarantee that the additive optimality gap is less than 4.3×10^5 and its provisioning cost is always less than 2.4×10^6, even though the number of flows become larger. Furthermore, we evaluate our algorithms in large-scale networks in Fig. 6(c), where the value of x-axis represents the number of switches and that of y-axis represents the provisioning cost. We fix the length of service chain at ten in this setting. We can see that BAA can reduce the provisioning cost by 86.0% and 72.4% compared to HW and Greedy respectively. And ODA can also reduce the provisioning cost by 72.6% and 45.9% compared to HW and Greedy respectively. Fig. 6(d) shows that the provisioning cost varies with the length of service chain. We can see that the cost of HW and Greedy increases significantly and that of BAA and ODA increases slowly. The reason is that BAA relies on the

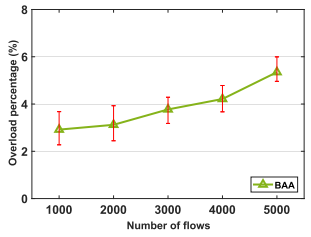


Fig. 7. Overload resource percentage.

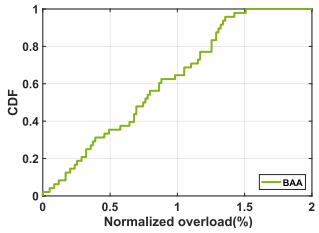


Fig. 8. Overbook ratio CDF.

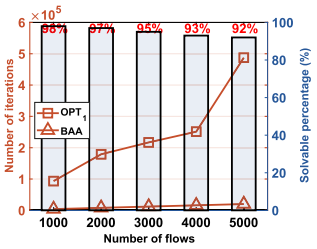


Fig. 9. Solvable percentage for OPT_1 and numbers of iterations.

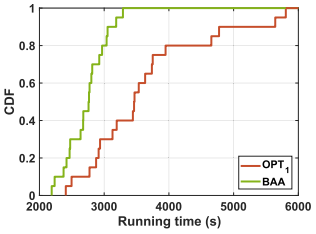


Fig. 10. Running time.

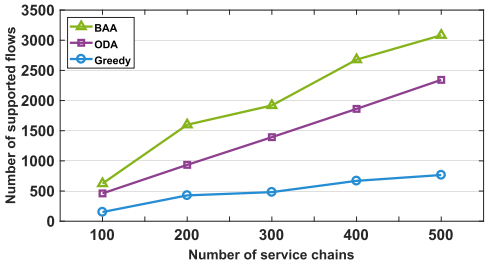


Fig. 11. Number of supported flows.

weighted matching algorithm and ODA takes the advantage of Q-learning, which is essentially better than Greey and can achieve near optimal.

We measure the overload resources percentage of BAA in Fig. 7. We define this percentage Θ as the ratio between the number of overload resources and that of total resources among all servers, *i.e.*, the ratio between the number of violated constraints η' and that of total constraints η,

$$\Theta = \frac{\eta'}{\eta}$$

In this setting, the length of VNF service chains is set to ten on average. The overload percentage becomes larger when the number of flows increases since large number of running VNFs requires more resources. Specifically, the overload percentage for BAA is at most around 6%, when the number of flows is 4000. We can reduce the flow demand or increase the resource capacity in practice to guarantee no overload event happens.

Fig. 8 shows the CDFs of normalized overbook ratio when the consumed resource is larger than the resource capacity. We define the overbook ratio Ψ as the following equation.

$$\Psi = \frac{\sum_{i \in SC} \hat{x}_{i,l} \cdot r_{i,l} - R_l}{R_l}$$

The resource violation for BAA is provably bounded and we can see that its overbook ratio is always less than 1.0% in our experiment. Note that the traffic we generated aims to maximize the network throughput. This indicates that the resource violation is relatively small even though the network becomes congested. In practice, we can scale down the resource capacity as an input to guarantee that our solution cannot beyond the physical resource capacity, which can be applied to the scenarios that the application performance can be significantly degraded once the resource is overloaded.

We now look at the percentage of solvability for OPT_1 and the number of solver iterations. In Fig. 9, the number of flows varies from 1000 to 5000 at the increment of 1000 for each run. We found that the number of solvable instances decreases when the number of flows becomes larger. Specifically, when the number of flows is 5000, around 8% instances cannot be solved by the standard solver. This demonstrates that OPT_1 cannot perfectly solve all instances, and it's going to get worse especially when the number of flows is large. Fig. 9 also shows the number of solver iterations for different schemes. We can see that the number of iterations for OPT_1 increases significantly than that for BAA, when the number of flows become large. The convergence rate of BAA in general can be faster than that of OPT_1. This is due to its polynomial-time complexity.

We evaluate the running time of our algorithm which is illustrated as CDFs in Fig. 10, when the number of flows is fixed at 5000. We can see that most cases (90%) using BAA finish within 3200 seconds while OPT_1 takes 4800 seconds. The running time of OPT_1 can be around twice than that of BAA in the worst case. We can observe that BAA is able to offer near equivalent performance and has more faster running time compared to OPT_1.

Finally, we measure the number of supported flows in Fig. 11. We randomly generate a large number of flows with different source and destination pairs and produce the number of supported flows when using BAA, ODA and Greedy. Essentially this measures the efficiency for different

deployment schemes. We observe that BAA and ODA can support more flows than Greedy as the number of service chains increases. The reason is that the probability of the successful deployment for BAA and ODA is higher than that for Greedy. Specifically, when the number of service chains is 500, the number of supported flows of BAA, ODA and Greedy is around 3100, 2350 and 760, respectively. We can see that the number of supported flows for BAA is almost four times as that of Greedy and even ODA can support three times numbers of flows as that for Greedy, when the total number of service chains is 500.

VII. Conclusion

We studied the problem of orchestrating service chain deployment in cellular networks. We proposed a two-stage optimization framework: the network-level optimization aiming to minimize the service chain deployment cost and the server-level optimization with the objective of balancing the CPU processing capability. We developed a set of algorithms — a multi-block ADMM algorithm, a bicriteria algorithm, a learning-based algorithm and a local search algorithm — to solve our problems. Large-scale simulations and DPDK-based OpenNetVM platform results show that our algorithms can reduce the capital cost and increase the throughput.

Appendix A
A Proximal Jacobian ADMM Algorithm

Now we reformulate the program (1) in order to apply ADMM. We first introduce slack variables α_l, β_p and γ_p to transform the inequality constraints (1a), (1b) and (1c) to equality constraints (5a), (5b) and (5c) required by ADMM. Second, all variables in the constraints of ADMM problem must be separable for each group of variables. To comply with this condition, we introduce auxiliary variables $\hat{x}_{i,l}$ and rewrite the constraints based on each pre-defined path. Towards this end, we add the original constraint (1d) and reformulate the program (1) to program (5) as follows.

$$\text{minimize} \sum_{l\in L}\sum_{i\in SC} c_{i,l}\cdot \hat{x}_{i,l} \tag{5}$$

$$\text{subject to } R_l - \sum_{i\in SC}\hat{x}_{i,l}\cdot r_{i,l} - \alpha_l = 0, \quad \forall l, \tag{5a}$$

$$b_p - d_f \cdot y_{f,p} - \beta_p = 0, \quad \forall f,p, \tag{5b}$$

$$\sum_{l\in p}\hat{x}_{i,l} - y_{f,p} - r_p = 0, \quad \forall f,p, \tag{5c}$$

$$(1d),$$

$$\hat{x}_{i,l}, y_{f,p}, \alpha_i, \beta_p, \gamma_p \geq 0, \quad \forall i,l,f,p. \tag{5d}$$

The new program (5) is equivalent to the original program (1). The variables $\hat{x}_{i,l}$ in constraint (5c) ensure that the service chain i should be deployed onto server l (l is one of the nodes in path p) if and only if the flow f passes through the path p.

Let $\mathscr{L}_\rho$ be the augmented Lagrangian of program (5) with dual variables θ, φ and σ. i.e., introducing an extra $\mathscr{L}$-2 norm term into the objective:

$$\begin{aligned}\mathscr{L}_\rho = &\sum_{l\in L}\theta_l\cdot\Delta_1 + \frac{\rho}{2}\sum_{\in L}\Delta_1^2 + \sum_{f\in F}\sum_{p\in p(f)}\varphi_{f,p}\cdot\Delta_2\\ &+\frac{\rho}{2}\sum_{f\in F}\sum_{p\in p(f)}\Delta_2^2 + \sum_{f\in F}\sum_{p\in p(f)}\sigma_{f,p}\cdot\Delta_3\\ &+\frac{\rho}{2}\sum_{f\in F}\sum_{p\in p(f)}\Delta_3^2 \end{aligned}\tag{6}$$

where $\rho > 0$ is the penalty parameter. Note that $\mathscr{L}_0$ (when $\rho = 0$) is the standard Lagrangian for our problem. The reason why we introduce the penalty term is to speed up the convergence rate [45]. In addition, to simplify the notation, we introduce Δ_1, Δ_2 and Δ_3.

$$\begin{aligned}\Delta_1 &= R_l - \sum_{i\in SC}\hat{x}_{i,l}\cdot r_{i,l} - \alpha_l\\ \Delta_2 &= b_p - d_f\cdot y_{f,p} - \beta_p\\ \Delta_3 &= \sum_{l\in p}\hat{x}_{i,l} - y_{f,p} - \gamma_p\end{aligned}$$

Distributed 5-block ADMM. We initialize the variables α, β, γ, $\hat{x}$, y and multipliers θ, φ, σ to zero. For $t = 1, 2, \cdots$, repeat the following steps.

1. α-update: Each server l solves the following subproblem for obtaining α_l^{t+1}:

$$\begin{aligned}\text{minimize } \theta_l^t\cdot\alpha_l + \frac{\rho}{2}\left(R_l - \sum_{i\in SC}\hat{x}_{i,l}^t\cdot r_{i,l} - \alpha_l\right)^2\\ + \frac{w}{2}\left(\alpha_l - \alpha_l^t\right)^2\end{aligned}\tag{7}$$

$$\text{subject to } \alpha_l \geq 0, \quad \forall l\in L. \tag{7a}$$

This per-server subproblem is a small-scale quadratic program and can be solved efficiently.

2. β-update Each generated p of the corresponding flow f solves the following subproblem for obtaining β_p^{t+1}:

$$\begin{aligned}\text{minimize } \sum_{f\in F}\varphi_{f,p}^t\cdot\beta_p + \frac{\rho}{2}\sum_{f\in F}\left(b_f - d_f\cdot y_{f,p}^t - \beta_p\right)^2\\ + \frac{w}{2}\left(\beta_p - \beta_p^t\right)^2\end{aligned}\tag{8}$$

$$\text{subject to } \beta_p \geq 0, \quad \forall p\in P(f). \tag{8a}$$

This subproblem can be solved by the standard solvers for quadratic program.

3. γ-update: Each generated p of the corresponding flow f solves the following subproblem for obtaining γ_p^{t+1}:

$$\begin{aligned}\text{minimize } \sum_{f\in F}\sigma_{f,p}^t\cdot\gamma_p + \frac{\rho}{2}\sum_{f\in F}\left(\sum_{l\in p}\hat{x}_{i,l}^t - y_{f,p}^t - \gamma_p\right)^2\\ + \frac{w}{2}\left(\gamma_p - \gamma_p^t\right)^2\end{aligned}\tag{9}$$

$$\text{subject to } \gamma_p \geq 0, \quad \forall p\in P(f). \tag{9a}$$

This subproblem can be solved by the standard solvers for quadratic program.

4. $\hat{x}$-update: Each server l solves the following subproblem for obtaining $x_{i,l}^{t+1} = (x_{1,l}^{t+1}, x_{2,l}^{t+1}, \cdots, x_{|SC|,l}^{t+1})$:

$$\begin{aligned}\text{minimize} \sum_{i\in SC} c_{i,l}\cdot\hat{x}_{i,l} + \theta_l^t \cdot \sum_{i\in SC}\hat{x}_{i,l}\cdot r_{i,l} \\ +\frac{\rho}{2}\left(R_l - \sum_{i\in SC}\hat{x}_{i,l}\cdot r_{i,l} - \alpha_l^t\right)^2 \\ +\frac{w}{2}\left(\hat{x}_{i,l} - \hat{x}_{i,l}^t\right)^2 \end{aligned} \tag{10}$$

$$\text{subject to } \hat{x}_{i,l} \geq 0, \quad \forall l \in L. \tag{10a}$$

This subproblem can be solved by the standard solvers for quadratic program.

5. y-update: Each generated p of the corresponding flow f solves the following subproblem for obtaining $y_{f,p}^{t+1} = (y_{1,p}^{t+1}, y_{2,p}^{t+1}, \cdots, y_{|F|,p}^{t+1})$:

$$\begin{aligned}\text{minimize } \frac{\rho}{2}\sum_{f\in F}\left(b_p - d_f\cdot y_{f,p} - \beta_p^t\right)^2 - \sum_{f\in F}\varphi_{f,p}^t\cdot d_f\cdot y_{f,p} \\ +\frac{\rho}{2}\sum_{f\in F}\left(\sum_{l\in p}\hat{x}_{i,l}^t - y_{f,p} - \gamma_p^t\right)^2 - \sum_{f\in F}\sigma_{f,p}^t\cdot y_{f,p} \\ +\frac{w}{2}\left(y_{f,p} - y_{f,p}^t\right)^2 \end{aligned} \tag{11}$$

$$\text{subject to (1d)},$$

$$y_{f,p} \geq 0, \quad \forall p \in P(f). \tag{11a}$$

This subproblem can be solved by the standard solvers for quadratic program.

6. Dual update: Each server j updates θ for the constraint (5a):

$$\theta_l^{t+1} = \theta_l^t + \iota\cdot\rho\cdot\left(R_l - \sum_{i\in SC}\hat{x}_{i,l}^{t+1}\cdot\hat{r}_{i,l}^{t+1} - \alpha_l^{t+1}\right) \tag{12}$$

Each generated p of the corresponding flow f updates φ for the constraint (5b):

$$\varphi_{f,p}^{t+1} = \varphi_{f,p}^t + \iota\cdot\rho\cdot\left(b_p - d_f\cdot y_{f,p}^{t+1} - \beta_p^{t+1}\right) \tag{13}$$

Each generated p of the corresponding flow f updates σ for the constraint (5c):

$$\sigma_{f,p}^{t+1} = \sigma_{f,p}^t + \iota\cdot\rho\cdot\left(\sum_{l\in p}\hat{x}_{i,l}^{t+1} - y_{f,p}^{t+1} - \gamma_p^{t+1}\right) \tag{14}$$

where $\iota\cdot\rho$ is the step size for the dual update.

APPENDIX B
PROOF OF THEOREM 3

Proof: We first introduce Theorem 6 to facilitate our proof.

Theorem 6: [34] For any bipartite graph $B = (V, W, F)$, each extreme point of the feasible region has integer coordinates. Furthermore, given edge cost $c_{v,w}$, $(v,w) \in F$, and a feasible fractional solution $y_{v,w}$, $(v,w) \in F$, we can find, in polynomial time, a feasible integer solution $\hat{y}_{v,w}$ such that

$$\sum_{(v,w)\in F} c_{v,w}\cdot\hat{y}_{v,w} \leq \sum_{(v,w)\in F} c_{v,w}\cdot y_{v,w}$$

Without loss of generality, we assume there are δ paths in path set P_f for flow f, i.e., the common node in the path set is at most δ. From Theorem 6, we obtain,

$$\sum_{l\in L}\sum_{i\in SC} c_{i,l}\cdot\hat{x}_{i,l} \leq \delta\cdot\sum_{l\in L}\sum_{i\in SC} c_{i,l}\cdot x_{i,l}^* = \delta\cdot OPT_1 \tag{15}$$

From the definition of complete matching, the provisioning resource Φ_l at each server l is

$$\Phi_l \leq \sum_{j=1}^{k_l}\hat{r}_l^j \tag{16}$$

where $\hat{r}_l^j = \max\{r_{i,l} | u_l^j \in U\}$.

We give the upper bound of $\sum_{j=1}^{k_l}\hat{r}_l^j$ as following.

$$\begin{aligned}\sum_{j=1}^{k_l}\hat{r}_l^j &= \hat{r}_l^1 + \sum_{j=2}^{k_l}\hat{r}_l^j \leq \hat{r}_l^1 + \sum_{j=2}^{k_l}\sum_i u_{i,l}^{j-1}\cdot r_{i,l} \\ &\leq \hat{r}_l^1 + \sum_{j=1}^{k_l}\sum_i u_{i,l}^j\cdot r_{i,l} \\ &= \hat{r}_l^1 + \sum_i\sum_{j=1}^{k_l} u_{i,l}^j\cdot r_{i,l} \\ &= \hat{r}_l^1 + \sum_i x_{i,l}\cdot r_{i,l} \end{aligned} \tag{17}$$

From constraint (1a), both (18) and (19) hold.

$$\hat{r}_l^1 \leq R_l \tag{18}$$

$$\sum_i x_{i,l}\cdot r_{i,l} \leq R_l \tag{19}$$

Combining (17), (18) and (19), we have the following inequation and conclude the proof.

$$\Phi_l \leq \sum_{j=1}^{k_l}\hat{r}_l^j \leq 2\cdot R_l \tag{20}$$

■

REFERENCES

[1] J. Zheng *et al.*, "Orchestrating service chain deployment with plutus in next generation cellular core," in *Proc. IWQoS*, 2019, p. 10.

[2] Y. Li, Z. Yuan, and C. Peng, "A control-plane perspective on reducing data access latency in lte networks," in *Proc. MOBICOM*, 2017, pp. 56–69.

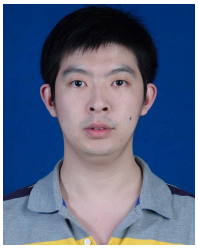
[3] (2016). *Cisco Visual Networking Index*. [Online]. Available: https://goo.gl/i6rd9e

[4] (2016). *Nokia. Signaling is Growing 50% Faster Than Data Traffic*. [Online]. Available: http://goo.gl/uwnRiO

[5] J. Sherry *et al.*, "Rollback-recovery for middleboxes," in *Proc. SIGCOMM*, 2015, pp. 227–240.

[6] A. Mohammadkhan, K. K. Ramakrishnan, A. S. Rajan, and C. Maciocco, "Cleang: A clean-slate EPC architecture and controlplane Protocol for next generation cellular networks," in *Proc. CAN*, 2016, pp. 31–36.

[7] V. Nagendra, H. Sharma, A. Chakraborty, and S. R. Das, "Lte-xtend: Scalable support of M2M devices in cellular packet core," in *Proc. 5th Workshop Things Cellular, Oper. Appl. Challenges*, 2016, pp. 43–48.
[8] R. Cohen, L. Lewin-Eytan, J. Naor, and D. Raz, "Near optimal placement of virtual network functions," in *Proc. INFOCOM*, 2015, pp. 1346–1354.
[9] B. Addis, D. Belabed, M. Bouet, and S. Secci, "Virtual network functions placement and routing optimization," in *Proc. CloudNet*, Oct. 2015, pp. 171–177.
[10] Z. A. Qazi, P. K. Penumarthi, V. Sekar, V. Gopalakrishnan, K. Joshi, and S. R. Das, "KLEIN: A minimally disruptive design for an elastic cellular core," in *Proc. SOSR*, 2016, p. 2.
[11] Z. A. Qazi, M. Walls, A. Panda, V. Sekar, S. Ratnasamy, and S. Shenker, "A high performance packet core for next generation cellular networks," in *Proc. SIGCOMM*, 2017, pp. 348–361.
[12] S. G. Kulkarni *et al.*, "Nfvnice: Dynamic backpressure and scheduling for NFV service chains," in *Proc. SIGCOMM*, 2017, pp. 71–84.
[13] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu, "NFP: Enabling network function parallelism in NFV," in *Proc. SIGCOMM*, 2017, pp. 43–56.
[14] W. Zhang, J. Hwang, S. Rajagopalan, K. K. Ramakrishnan, and T. Wood, "Flurries: Countless fine-grained nfs for flexible per-flow customization," in *Proc. CoNEXT*, 2016, pp. 3–17.
[15] W. Zhang *et al.*, "Opennetvm: A platform for high performance network service chains," in *Proc. HotMiddlebox*, 2016, pp. 26–31.
[16] X. Jin, L. E. Li, L. Vanbever, and J. Rexford, "Softcell: Scalable and flexible cellular core network architecture," in *Proc. CoNEXT*, 2013, pp. 163–174.
[17] M. Moradi, W. Wu, L. E. Li, and Z. M. Mao, "Softmow: Recursive and reconfigurable cellular WAN architecture," in *Proc. CoNEXT*, 2014, pp. 377–390.
[18] K. Nagaraj and S. Katti, "Procel: Smart traffic handling for a scalable software EPC," in *Proc. HotSDN*, 2014, pp. 43–48.
[19] A. Basta, W. Kellerer, M. Hoffmann, K. Hoffmann, and E. Schmidt, "A virtual sdn-enabled LTE EPC architecture: A case study for s-/p-gateways functions," in *Proc. SDN4FNS*, 2013, pp. 1–7.
[20] J. Kempf, B. Johansson, S. Pettersson, H. Luning, and T. Nilsson, "Moving the mobile evolved packet core to the cloud," in *Proc. WiMob*, 2012, pp. 784–791.
[21] K. Pentikousis, Y. Wang, and W. Hu, "Mobileflow: Toward software-defined mobile networks," *IEEE Commun. Mag.*, vol. 51, no. 7, pp. 44–53, Jul. 2013.
[22] M. R. Sama, L. M. Contreras, J. Kaippallimalil, I. Akiyoshi, H. Qian, and H. Ni, "Software-defined control of the virtualized mobile packet core," *IEEE Commun. Mag.*, vol. 53, no. 2, pp. 107–115, Feb. 2015.
[23] A. Banerjee, R. Mahindra, K. Sundaresan, S. K. Kasera, K. van der Merwe, and S. Rangarajan, "Scaling the LTE control-plane for future mobile access," in *Proc. CoNEXT*, 2015, pp. 19:1–19:13.
[24] H. Moens and F. D. Turck, "VNF-P: A model for efficient placement of virtualized network functions," in *Proc. CNSM*, 2014, pp. 418–423.
[25] M. Ghaznavi, A. Khan, N. Shahriar, K. Alsubhi, R. Ahmed, and R. Boutaba, "Elastic virtual network function placement," in *Proc. CloudNet*, 2015, pp. 255–260.
[26] S. Palkar *et al.*, "E2: A framework for NFV applications," in *Proc. SOSP*, 2015, pp. 121–136.
[27] S. I. Kim and H. S. Kim, "A research on dynamic service function chaining based on reinforcement learning using resource usage," in *Proc. 9th Int. Conf. Ubiquitous Future Netw. (ICUFN)*, Jul. 2017, pp. 582–586.
[28] J. Sun, G. Huang, G. Sun, H. Yu, A. K. Sangaiah, and V. Chang, "A q-learning-based approach for deploying dynamic service function chains," *Symmetry*, vol. 10, no. 11, p. 646, 2018.
[29] H. R. Khezri, P. A. Moghadam, M. K. Farshbafan, V. Shah-Mansouri, H. Kebriaei, and D. Niyato, "Deep q-learning for dynamic reliability aware NFV-based service provisioning," 2018, *arXiv:1812.00737*. [Online]. Available: https://arxiv.org/abs/1812.00737
[30] *3GPP Specifications*. Accessed: 2019. [Online]. Available: http://www.3gpp.org/
[31] A. S. Rajan *et al.*, "Understanding the bottlenecks in virtualizing cellular core network functions," in *Proc. LANMAN*, 2015, pp. 1–6.
[32] A. Banerjee *et al.*, "Phantomnet: Research infrastructure for mobile networking, cloud computing and software-defined networking," *GetMobile*, vol. 19, no. 2, pp. 28–33, 2015.
[33] P. Patel *et al.*, "Ananta: Cloud scale load balancing," in *Proc. SIGCOMM*, 2013, pp. 207–218.
[34] D. P. Williamson and D. B. Shmoys, *The Design Approximation Algorithms*. Cambridge, U.K.: Cambridge Univ. Press, 2011.
[35] S. Chopra and M. R. Rao, "The partition problem," *Math. Program.*, vol. 59, pp. 87–115, 1993.
[36] W. Deng, M. Lai, Z. Peng, and W. Yin, "Parallel multi-block ADMM with o(1/k) convergence," *J. Sci. Comput.*, vol. 71, no. 2, pp. 712–736, 2017.
[37] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 2018.
[38] S. Marsland, *Machine Learning: An Algorithmic Perspective*. London, U.K.: Chapman & Hall, 2011.
[39] C. J. Watkins and P. Dayan, "Q-learning," *Mach. Learn.*, vol. 8, nos. 3–4, pp. 279–292, 1992.
[40] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. A. Maltz, "Zupdate: Updating data center networks with zero loss," in *Proc. SIGCOMM*, 2013, pp. 411–422.
[41] N. Ron and T. Naveh, "Wireless backhaul topologies: Analyzing backhaul topology strategies," *Ceragon White Paper*, Aug. 2010, pp. 1–15.
[42] *A Simple Model for Determining True Total Cost of Ownership for Data Centers*. Accessed: 2019. [Online]. Available: http://tinyurl.com/kznlhn2/
[43] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proc. SOCC*, 2012, p. 7.
[44] C. Reiss, J. Wilkes, and J. L. Hellerstein, *Google Cluster-Usage Traces*. Accessed: 2019. [Online]. Available: http://code.google.com/p/googleclusterdata/
[45] S. P. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Found. Trends Mach. Learn.*, vol. 3, no. 1, pp. 1–122, 2011.



Jiaqi Zheng (M'15) received the Ph.D. degree from Nanjing University, China, in 2017. He was a Research Assistant with the City University of Hong Kong in 2015. He was also a Visiting Scholar with Temple University in 2016. He is currently a Research Assistant Professor with the Department of Computer Science and Technology, Nanjing University. His current research interests include computer networking, particularly data center networks, SDN/NFV, and machine learning systems. He is a member of the ACM. He received the Best Paper Award from IEEE ICNP 2015, the Doctoral Dissertation Award from the ACM SIGCOMM China 2018, and the First Prize of Jiangsu Science and Technology Award in 2018.

Chen Tian received the B.S., M.S., and Ph.D. degrees from the Department of Electronics and Information Engineering, Huazhong University of Science and Technology, China, in 2000, 2003, and 2008, respectively. From 2012 to 2013, he was a Post-Doctoral Researcher with the Department of Computer Science, Yale University. He was an Associate Professor with the School of Electronics Information and Communications, Huazhong University of Science and Technology, China. He is currently an Associate Professor with the State Key Laboratory for Novel Software Technology, Nanjing University, China. His research interests include data center networks, network function virtualization, distributed systems, Internet streaming, and urban computing.

Haipeng Dai (M'15) received the B.S. degree from the Department of Electronic Engineering, Shanghai Jiao Tong University, Shanghai, China, in 2010, and the Ph.D. degree from the Department of Computer Science and Technology, Nanjing University, Nanjing, China, in 2014. He is currently an Associate Professor with the Department of Computer Science and Technology, Nanjing University. He is an ACM member. He received the Best Paper Award from IEEE ICNP'15, the Best Paper Award Runner-up from IEEE SECON'18, and the Best Paper Award Candidate from IEEE INFOCOM'17.

Qiufang Ma received the B.S. degree from the College of Information Engineering, Nanjing University of Finance and Economics, Nanjing, China, in 2016, and the master's degree from the Department of Computer Science and Technology, Nanjing University, in 2019. Her research interests focus on network function virtualization.

Wei Zhang (M'15) received the Ph.D. degree from George Washington University in 2018. She is currently a Software Engineer with Microsoft Azure Networking. Her research interests include cloud computing, distributed system, resource disaggregation, and network and storage system. She has received the One Best Paper Award, second prize in 2015 GENI Networking Competition, Cisco fellowship, Beihang National Scholarship, and the Outstanding Graduate Student Award.

Guihai Chen (SM'18) received the B.S. degree in computer software from Nanjing University in 1984, the M.E. degree in computer applications from Southeast University in 1987, and the Ph.D. degree in computer science from The University of Hong Kong in 1997. He is currently a Distinguished Professor with Nanjing University. He has a wide range of research interests with focus on parallel computing, wireless networks, data centers, peer-to-peer computing, high-performance computer architecture, and data engineering.

Gong Zhang (M'12) was a System Engineer for L3+ Switch Product in 2000. He was a Product Development Team (PDT) Leader of Smart Devices, pioneering a new consumer business for the company in 2002. He was a Senior Researcher, leading future Internet research and cooperative communication, and did Mobility Research Program in 2005. He is currently a Chief Architect Researcher and the Director of Theory Lab of Huawei 2012 Labs. He is over 15 years Research experience of system architect in Network, distributed system, and communication system. He had contributed more than 90 patents globally.