Enabling NFV Elasticity Control With Optimized Flow Migration

Chen Sun, Jun Bi, *Senior Member, IEEE*, Zili Meng, Tong Yang, *Member, IEEE*, Xiao Zhang, and Hongxin Hu, *Member, IEEE*

***Abstract*— Network function virtualization (NFV) together with software defined networking (SDN) offers the potential for enhancing service delivery flexibility and reducing overall costs. Based on the capability of dynamic creation and destruction of network function (NF) instances, NFV provides great elasticity in NF control, such as NF scaling out, scaling in, and load balancing. To realize NFV elasticity control, network traffic flows need to be redistributed across NF instances. However, deciding which flows are suitable for migration is a critical problem for efficient NFV elasticity control. In this paper, we propose to build an innovative flow migration controller, `OFM controller`, to achieve optimized flow migration for NFV elasticity control. We identify the trigger conditions and control goals for different situations, and carefully design models and algorithms to address three major challenges including *buffer overflow avoidance*, *migration cost calculation*, and *effective flow selection for migration*. We implement the `OFM controller` on top of NFV and SDN environments. Our evaluation results show that `OFM controller` is efficient to support optimized flow migration in NFV elasticity control.**

***Index Terms*— Elasticity control, optimized flow migration, service level agreement, SLA, NFV.**

I. Introduction

NETWORK Function Virtualization (NFV) [2] was recently introduced to replace traditional dedicated hardware middleboxes with software based Network Functions (NFs) to offer the potential for both enhancing service delivery flexibility and reducing overall costs. Based on the capability of dynamic NF creation and destruction, NFV could support elastic control over NF instances to adapt to frequent and substantial dynamics of network traffic volumes [3], [4].

Manuscript received May 5, 2018; revised August 18, 2018; accepted August 28, 2018. Date of publication September 13, 2018; date of current version November 28, 2018. This work was supported in part by the National Key R&D Program of China under Grant 2017YFB0801701 and in part by the National Science Foundation of China under Grant 61472213. This paper was presented at the IEEE IWQoS 2018, Banff, AB, Canada, June 4, 2018 [1]. *(Corresponding authors: Jun Bi; Tong Yang.)*

C. Sun, J. Bi, Z. Meng, and X. Zhang are with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China, also with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China, and also with the Beijing National Research Center for Information Science and Technology, Beijing 100084, China (e-mail: c-sun14@mails.tsinghua.edu.cn; junbi@tsinghua.edu.cn; mengzl15@mails.tsinghua.edu.cn; zhang-x16@mails.tsinghua.edu.cn).

T. Yang is with the Department of Computer and Science, Peking University, Beijing 100871, China (e-mail: yangtongemail@gmail.com).

H. Hu is with the School of Computing, Clemson University, Clemson, SC 29634 USA (e-mail: hongxih@clemson.edu).

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

Digital Object Identifier 10.1109/JSAC.2018.2869953

The elasticity of NFV has been widely exploited in real world networks such as data centers [5] and 5G networks [6], [7]. For example, 5G networks provide isolated network slices featuring high throughput, ultra-low latency, or massive connections. Each slice contains network services (NSs) that are composed of virtual network functions (VNFs) deployed in virtual machines. To adapt to increasing traffic loads in a network slice, recent research has proposed to scale out a single VNF [8] or an entire NS [9] by dynamically deploying new virtual machines (VMs) that carry VNF or NS instances. After scaling, flows are distributed across a set of identical instances using a front-end physical or virtual switch as a load balancer [5], [9].

Furthermore, NFs typically have to maintain *state* information for processed flows [10], [11]. To ensure the correctness of packet processing after flow redistribution, some research efforts [5], [12]–[16] have proposed to transfer flow states alongside the flow migration. Split/Merge [14] and OpenNF [5] rely on a centralized controller to transfer states between NF instances and buffer incoming packets to realize loss-free and order-preserving migration. On the other hand, enhanced OpenNF [12] and other recent works [13], [15] performed migration directly among NF instances to improve the scalability and performance of flow migration in NFV networks. Above research efforts mainly focus on designing mechanisms for *safe* migration of flow states among NF instances.

However, *selecting suitable flows to migrate* is also a significant problem in NFV elasticity control. A careless selection of flows for migration would incur three major problems:

- **Buffer overflow.** From the system's perspective, flow migration requires a preallocated buffer in the destination NF [12], [15] to store in-flight traffic. In-flight traffic refers to the traffic that arrives at the source instance after the states have been migrated, or the traffic that arrives at the destination instance before corresponding states become available. A careless selection of flows could result in migrating several elephant flows together, which might overflow the buffer space and incur packet loss or service degradation.
- **High migration cost.** From the network tenant's perspective, NFV networks should satisfy Service Level Agreements (SLAs). A breach of certain SLAs would incur penalties. However, flow migration might bring additional processing latency (tens of milliseconds in [15]), which may be unacceptable for flows that demand tight latency

TABLE I
NFV ELASTICITY CONTROL SITUATIONS

Situations	When to Migrate	Why to Migrate	Where to Migrate	Which Flows to Migrate
NF Overload	NF load $>$ peak load threshold	Avoid performance degrading	Instances that are not overloaded or newly created instances (Scale out)	Some (Which flows?)
NF Underload	NF load $<$ bottom load threshold	Save resources for reusing and achieve energy efficiency	Merge current instances (Scale in)	All flows of some instances (Which instances?)
Load Balancing	NF instances have imbalanced load	Prevent possible overload	Among current instances	Some (Which flows?)
Failure Recovery	NF instance failure occurs	Realize failure recovery	Non-failed instances	All
NF Upgrading	NF features require upgrading	Carry out network policies	Upgraded instances	All

SLAs (such as flows of algorithmic stock trading or high performance distributed memory caches [17]), while acceptable for flows with looser latency constraints (such as P2P transmission flows). Thus, randomly selecting flows to migrate may result in serious SLA violation and increase migration costs significantly.

- **Ineffective migration.** From the network operator's perspective, realizing NFV elasticity control without a proper flow selection mechanism may fail to achieve the control expectation. For instance, when an NF instance is overloaded, selecting too few flows to migrate might not effectively alleviate the hot spot, while migrating too many flows might create new hot spots.

To address the above problems, in this paper, we propose a novel flow migration controller, `OFM Controller`, for optimized flow migration in NFV elasticity control. To the best of our knowledge, we are the first to design such a controller that performs optimized flow selection for NFV elasticity control. We analyze NFV elasticity control situations and carefully design the `OFM Controller` to fully achieve control goals, minimize migration costs, and avoid buffer overflow. We make the following contributions in this paper:

- We categorize typical NFV elasticity control *situations* including NF scaling, NF load balancing, NF failure recovery, and NF upgrading. We analyze in detail the trigger conditions and flow selection goals of each situation, and present the design challenges. (§ II)
- We propose the design of `OFM Controller` for optimized flow migration in NFV elasticity control. The `OFM Controller` collects flow statistics and NF loads during runtime, and identifies situations where flow migration is required. By effectively modeling the buffer requirements and migration latency (§ III), `OFM` could select proper flows to achieve *control goals* while minimizing the *migration costs* and avoiding *buffer overflow* (§ IV).
- We implement the `OFM Controller` based on Floodlight and perform extensive evaluations. Experimental results show that `OFM` could achieve optimized flow migration in NFV elasticity control, while ensuring full achievement of control goals. (§ V)

The rest of this paper is organized as follows. Section II analyzes situations for NFV elasticity control. Section III elaborates the `OFM` design. Section IV presents optimized flow migration calculation in `OFM`. We present the implementation and evaluation results in Section V. We summarize the related work in Section VI, and conclude this paper in Section VII.

II. ELASTICITY CONTROL SITUATIONS ANALYSIS

This section first summarizes the situations where flow migration is required for NFV elasticity control. Then we analyze the control goals and constraints of each situation as well as the design challenges, which guide the design of `OFM`.

A. NFV Elasticity Control Situations

We list five typical situations of NFV elasticity control in Table I, and analyze those situations in this section.

1) NF Scaling Out: This happens anytime when the load of an NF instance exceeds the NF processing load threshold [5], [14], [18], [19]. Network operators could perform NF scaling out in the runtime to alleviate the hot spot and avoid performance degrading by migrating *some* flows from the overloaded instance to other instances or to the newly created instance. However, flows on the overloaded instance have various SLA constraints and sizes. Proper flows should be selected to alleviate the hot spot and create no new hot spots while incurring minimal SLA violations and avoiding buffer overflow.

2) NF Scaling in: To save resources and achieve energy efficiency, when one or multiple NF instances are underloaded, NF scaling in is performed by destroying *some* VMs and migrating *all flows on those instances* to the remaining ones [5], [14], [18], [19], which could reduce Operating Expenditures (OPEX). In this article, we refer to the reduction of OPEX as gaining "revenue benefit" [20]. However, flow migration incurs additional latency and could violate SLA of some flows. Therefore, we should select proper NF instances to destroy for maximum revenue benefit and minimum migration costs.

3) NF Load Balancing: NF load balancing redistributes flows across current NF instances to prevent potential NF overload situations. NF load balancing brings no revenue benefits since it does not shut down VMs. However, flow migration might bring additional forwarding latency and incur SLA violation penalties. Thus, we should select *proper* flows for migration to both balance the load and minimize migration costs.

4) NF Failure Recovery: When an instance fails, we need to recover from the failure by rerouting *all* flows on the failed instance to healthy instances or by creating new instances [5].

5) NF Upgrading: For maximum security, a network provider may want traffic to always be processed by the latest NF software [5]. NFV provides the capability to dynamically

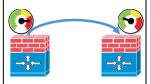
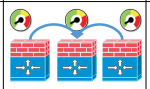
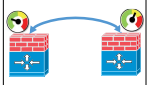
Situations		Flow Selection Goals
Overload (Scale Out)		- Quick hot spot alleviation - New hot spot avoidance - Minimal migration costs - Buffer overflow avoidance
Underload (Scale In)		- Maximal revenue benefit (Minimal migration costs) - New hot spot avoidance - Reasonable calculation time - Buffer overflow avoidance
NF Load Balancing		- Balanced NF load - Minimal migration costs - New hot spot avoidance - Quick balancing and converging - Buffer overflow avoidance

Fig. 1. Flow selection goals for different control situations.

launch updated NF instances. We need to migrate *all* flows and states to the updated instances.

B. Flow Selection Goals for NFV Elasticity Control

From the above analysis, we observe that situations including NF scaling out, scaling in, and load balancing require a careful selection of flows to achieve control goals while minimizing migration costs and avoiding buffer overflow. Therefore, we next analyze the detailed flow selection goals when coping with each situation, and show them in Fig. 1.

1) NF Scaling Out: When an NF is overloaded, NF scaling out *must* be performed to avoid packet loss or performance degradation. Operators expect a *quick* load alleviation without creating new hot spots. Besides, minimal migration costs are desired and buffer overflow should be avoided.

2) NF Scaling In: As merging multiple instances into fewer ones and destroying free VMs could improve energy efficiency and bring *revenue benefits*, we want to minimize the number of remaining instances. However, flows on different instances have different SLA constraints, and we want to minimize the migration costs simultaneously. Therefore, we need to compare *SLA penalties* for migrating flows on each instance with the *revenue benefit* brought by destroying the VM, and find the optimal migration plan. Besides, merging multiple instances onto one requires a safe scaling in without creating new hot spots. Finally, buffer overflow should be avoided during migration.

3) NF Load Balancing: Load balancing could balance the load among NF instances and prevent potential NF overload situations. However, NF load balancing is neither compulsory (like scaling out) nor directly rewarding (like scaling in). Therefore, to minimize the flow migration costs, we should only redistribute flows with loose SLAs that would not be violated during migration. Thus, only a limited set of flows could be reallocated, which might not result in a completely balanced final load. However, we could ameliorate the load imbalance situation to some extent with no costs.

A strawman solution for NFV elasticity control proposed in E2 [18] adopts a strategy of *migration avoidance*. Existing flows are still processed by the previously assigned NF instance, while only new flows are differentially handled. In this way, no flow migration occurs for NFV elasticity control. For example, for NF scaling out, we could simply instantiate a new NF instance and redirect new flows to it. While the migration avoidance strategy introduces no migration penalty, it may still result in penalty during runtime. If an NF is overloaded, we should quickly migrate flows away from the instance to avoid performance degradation and SLA violation. We analyze the migration avoidance strategy in detail in § VI.

C. Design Challenges

To achieve above flow selection goals, we design the `OFM Controller` for NFV elasticity control. We encounter three major challenges in the design of `OFM`.

1) Buffer Overflow Avoidance: A safe elasticity control requires buffering in-flight traffic in the destination instance [12], [15] during migration. However, buffer space is not unlimited. We observe that migration of different flows incurs different amount of in-flight traffic. Therefore, care must be taken while selecting flows to migrate to avoid buffer overflow. To this end, `OFM` dynamically measures the size of flows on NF instances *without intrusion into NF logic*, and models the buffer space requirement for the migration of each flow. (§ III.*B*).

2) Migration Cost Calculation: Flow migration could bring additional forwarding latency, violate SLA constraints, and incur a penalty. However, the penalty depends on the extent to which the SLA is violated, *i.e.* the exceeding delay time over the SLA constraints. Therefore, `OFM` is challenged to precisely estimate the migration latency, which could vary significantly with the number of flows to migrate [5], [15]. In response, through experiments on real world NFs, `OFM` builds models for flow migration latency based on the number of flows to migrate and use it to calculate migration costs (§ III.*C*).

3) Effective Flow Selection for Migration: As analyzed in § II.*B*, different control situations have *unique control goals*. Therefore, we are challenged to design optimized flow selection mechanisms for the three situations. However, *massive parameters* including NF load, flow size (elephant or mice flows as defined in [21] and [22]), migration latency of different sized flows, VM revenue benefit, and buffer cost should be considered to find an optimized migration plan for each situation, making it challenging to design algorithms for optimized flow selection. Furthermore, the calculation could consume significant time, which may be unacceptable for situations like NF scaling out that requires a quick hot spot alleviation. To address the above challenges, `OFM` carefully designs a unified, optimal, but complex algorithm that could handle the three situations simultaneously, and propose three fast but sub-optimal algorithms for the three situations respectively while taking into account all above parameters. (§ IV).

III. OFM Design

To address the above challenges, we design the `OFM Controller` to implement optimized flow migration in NFV elasticity control situations. Components and workflow of the `OFM Controller` are shown in Fig. 2. `OFM Controller` monitors the status of every NF instance and detects traffic overload, underload, and imbalance *conditions*. At the same time, the `OFM Controller` collects the statistics of flows

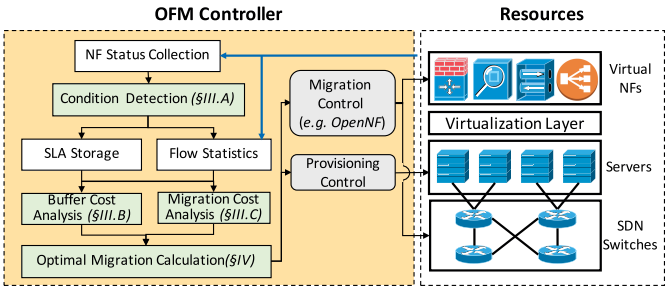


Fig. 2. `OFM Controller` components and workflow.

on each NF for further selection (§ III.*A*). Once a condition is detected, based on flow SLA constraints and dynamically gathered flow statistics, `OFM Controller` first performs *Buffer Cost Analysis* (§ III.*B*) and *Migration Cost Analysis* (§ III.*C*). The analysis results are inputted into *Optimal Migration Calculation* (§ IV) to create the optimized migration plan. Finally, `OFM` Provisioning Control and Migration Control modules would interact with underlying resources to perform flow migration in the same way as introduced in [5], [12], [14], and [15], *etc*.

However, a natural concern would be the practicality of calculating an optimized plan for *future* migration based on the *current* flow statistics. Actually, as mentioned in [23], we should be able to use routes based on historical traffic patterns for the last 1 second for effective flow scheduling. Furthermore, as shown in § V-B, `OFM` can finish gathering statistics and calculating within 1 second for all situations, which demonstrates the timeliness of `OFM`. Next we introduce each module of `OFM` in detail.

A. NF & Flow Status Collection and Condition Detection

The `OFM Controller` needs to collect NF processing load, *i.e.*, throughput, for elasticity control condition detection, as well as the flow sizes for flow selection. A naive approach to obtaining these statistics is to modify NF logic to maintain flow-level packet counters. However, doing so would intrude NF logic and increase NF development burden of statistics gathering and communication with the controller. To precisely collect above statistics in a light-weight manner, we exploit the *flow table entry counters* of OpenFlow [24]. Physical or virtual OpenFlow switches are widely used to connect physical servers or VMs in the same server in NFV networks [5], [25]–[27]. OpenFlow switches maintain a byte counter for each flow table entry, while the controller queries counters from switches during runtime. However, flow entries in OpenFlow flow tables are usually aggregated [28]. Directly querying counters cannot provide flow-level byte counters. Therefore, we utilize OpenFlow's multi-stage flow tables [24], assign the *first* flow table of an *edge* switch connected with NFs as the *counter table*, and issue fine-grained rules to it to maintain flow-level counters. The action of each entry in the counter table is to directly send packets to the next flow table. As shown in Fig. 3, we periodically query flow counters from the counter table, and calculate the flow size by dividing the counter difference by the query interval. Since the `OFM Controller` can acknowledge the target NF of each flow, it groups the flows based on the target NF and adds up the sizes of flows targeting at the same NF to get the real-time throughput of the NF.

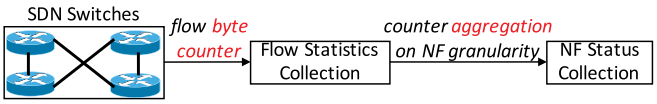


Fig. 3. NF status collection in `OFM`.

Suppose there are n NF instances of the same type, such as firewalls, running in the NFV network. The `OFM Controller` periodically queries flow statistics from the data plane, and calculates the load l_j of instance $j \in [1, n]$. For condition detection, we define th_{top} as the peak processing load threshold of an NF instance, and th_{bottom} as the bottom load threshold. We use the *variance* of the NF loads $var(l_1, ..., l_n)$ to quantify the load imbalance grade. We define the maximum allowed variance of NF loads as th_{var}. We define conditions for NFV elasticity control as:

- Overload: $l_j \geqslant th_{top,j}$ *for any* $j \in [1, n]$
- Underload: $l_j \leqslant th_{bottom,j}$ *for any* $j \in [1, n]$
- Imbalance: $var(l_1, ..., l_n) \geqslant th_{var}$

Based on above rules, NFV elasticity control conditions can be detected, which would trigger optimal flow migration calculation to handle the situation.

B. Buffer Cost Analysis

During the migration, in-flight traffic needs to be buffered until the end of the state installation. Then, in-flight traffic will be flushed to the destination NF instance for processing. The `OFM Controller` adopts the distributed buffering mechanism in [12] and buffers the in-flight traffic in the destination instance. We target at avoiding buffer overflow by estimating the in-flight traffic in the following way.

Suppose flow k of byte rate $size_k$ needs to be migrated, and the migration time of flow k is denoted as $la_{migration,k}$. During flow migration, all in-flight packets of this flow are buffered at the destination instance. Therefore, the total buffered packet size required could be modeled as:

$$buffer_k = size_k \times la_{migration,k} \tag{1}$$

In this way, we could calculate the buffer requirement for migrating each flow, and select proper set of flows to avoid buffer overflow in the destination instance. The estimation of the flow migration time will be introduced later in this section.

C. Migration Cost Analysis

Due to the additional latency incurred by flow migration, NFV elasticity control might break flow SLAs [20] and cause penalty [29]. Furthermore, for NF scaling in, shutting down underloaded VMs could bring revenue benefit and ameliorate the migration cost. Next we introduce the SLA violation penalties and revenue benefit estimation in detail.

1) Penalty for SLA Violations: Latency related SLAs in cloud services regulate maximum processing latency for specific *request types* [29]. Similarly, we define latency related SLAs in NFV: it regulates the maximum latency for each

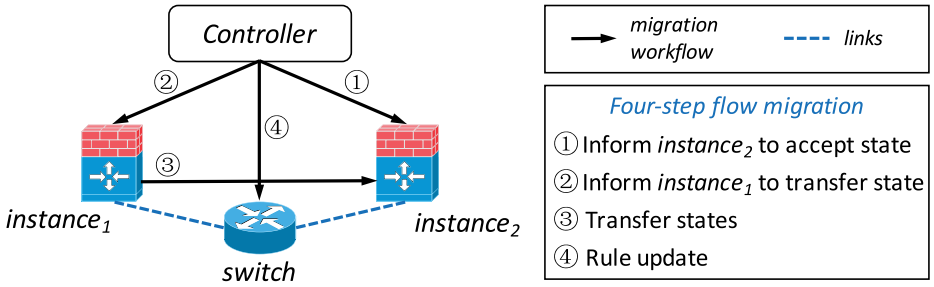


Fig. 4. Four-step flow migration workflow from [12]. The switch could be physical/virtual.

flow processed by NFV networks, where NFV provides *packet processing* services by NFs including firewall, IDS, VPN, load balancing, etc [18], [19].

Given an Instance j with m flows, given a flow k, let la_k be the latency of flow k, and let LA_k be the SLA latency of flow k. Obviously, la_k should be no larger than LA_k. During runtime without flow migration, the total latency of flow k on NF instance j is equal to the NF processing latency, *i.e.* $la_k = la_{processing,j}\ for\ k \in [1, m]$. However, flow migration might introduce additional latency overhead. Therefore, in order to meet the latency related SLA during migration, the migration latency should satisfy:

$$la_{migration,k} \leqslant LA_k - la_{processing,j}\ for\ k \in [1, m] \quad (2)$$

During flow migration, the above inequality might be breached and incur penalty. The untimely-processed traffic of a flow k is exactly the buffered in-flight traffic, *i.e.* $buffer_k$. According to [29], we could model the SLA violation penalty as a linear function. We denote the penalty rate as β, and the delay time for migrating a flow as DT. We have:

$$Penalty = \alpha + \beta \times buffer \times DT \quad (3)$$

However, for a flow k, if its latency SLA is not violated, the delay time is set to zero, and the penalty should be zero. Otherwise, the delay time is the exceeded latency over the SLA constraint. Therefore, we have:

$$DT_k = max\ (0,\ la_{migration,k} + la_{processing,k} - LA_k) \quad (4)$$

The migration penalty of flow k could be modeled as:

$$Penalty_k = \begin{cases} \alpha + \beta \times buffer_k \times DT_k & DT_k > 0 \\ 0 & DT_k = 0 \end{cases} \quad (5)$$

Next, we need to estimate the migration time of a flow to calculate the SLA violation penalty. Fig. 4 shows the four-step workflow of state migration in [12] with four major time usages.

- $t1$: the time of the controller informing the destination instance to accept state.
- $t2$: the time of the controller informing the source instance to transfer state.
- ts_k: the state transfer time for the flow k
- tu: the flow rule update time.

The total migration time for flow k could be represented as:

$$la_{migration,k} = t1 + t2 + ts_k + tu \quad (6)$$

Among them, $t1$, $t2$, and tu are not related to the specific flow to migrate. We could easily measure them in NFV networks and consider them as constants. However, as illustrated in [12], the state transfer time depends on the number of flows to migrate, regardless of flow sizes. Our evaluation in Section V demonstrates a linear relationship between state transfer time of a flow (ts_k) and the total number of flows (f_n) to migrate. We describe their relationship as:

$$ts_k = \gamma + \eta \times f_n \quad (7)$$

γ and η are two constants and could vary for different NF types. In this way, we could estimate the migration time of the *selected flows* and calculate the penalty. When migrating f_n flows, for an individual flow k, the delayed time, buffer requirement, and migration cost are modeled as:

$$\begin{aligned} DT_k &= (t1 + t2 + tu + \gamma + \eta * f_n) \\ &\quad + la_{processing,k} - LA_k \quad (8) \\ buffer_k &= size_k \times (t1 + t2 + tu + \gamma + \eta \times f_n) \quad (9) \\ cost_k &= Penalty_k \\ &= \begin{cases} \alpha + \beta \times buffer_k \times DT_k & DT_k > 0 \\ 0 & DT_k = 0 \end{cases} \quad (10) \end{aligned}$$

2) Revenue Benefit Estimation: For NF *scaling in* situations, shutting down VMs could bring revenue benefit and neutralize the migration cost. We denote the price (*i.e.* the cost of the VM per time slot [29]) of the VM j as $PriVM_j$. Furthermore, we need to estimate the VM runtime saved by VM scaling in. Suppose we always destruct a VM when it is underloaded. The running time saved in this approach is the time when VM load is under th_{bottom}. Therefore, we collect the historical data and calculate the average time interval $TINT_{avg}$ when an VM is underloaded, and take it as the estimated saved time. Therefore, we model the revenue benefit and the total migration cost of destructing VM j as:

$$benefit_j = PriVM_j \times TINT_{avg}\ for\ j \in [1, m] \quad (11)$$

$$cost_j = \sum_{k=1}^{n} Penalty_k - benefit_j \quad (12)$$

IV. Optimized Flow Migration Calculation

Based on above modeling and analysis, in this section, we present algorithms used by the `OFM Controller` to achieve optimized NFV elasticity control. As analyzed in § II,

the three NFV elasticity control situations have unique flow selection goals. Before going to the algorithm design, we first analyze the flow selection goals in detail.

First, when one or multiple NF instances are overloaded, `OFM Controller` performs NF scaling out by migrating some flows from the overloaded instances to other instances or to the newly created instances. A strawman solution for NF scaling out is to migrate *half* of the traffic load away to effectively alleviate the hot spot. However, flows on the overloaded instance may have tight latency SLAs, and migrating half of these flows could incur large penalties. Actually, the basic control goal of NF scaling out is to migrate some flows away to reduce the NF load *below the peak threshold*. To achieve this control goal, we introduce the *peak safe threshold* th_{safe}, which regulates the peak load of the overloaded instance after scaling out. For example, suppose the peak threshold $th_{peak} = 80\%$ of the total capacity while $th_{safe} = 60\%$. Suppose there is an overloaded (80%) instance. Instead of migrating half (40%) load, we could simply ensure that 20% is migrated away for effective overload mitigation. Note that the actual threshold values could be dynamically configured by network operators based on network traffic statistics. The determination of the threshold values is out of the scope of this paper.

Second, when there are $N_{us} \geqslant 1$ underloaded NF instances of the same type, `OFM Controller` would perform NF scaling in by merging some instances onto one and shutting down the free VMs. During instance merging, we pursue the maximum revenue benefit and avoid creating no new hot spots by ensuring the load of the remaining instances are below th_{safe}.

Finally, we periodically perform NF load balancing to prevent potential NF overload situations. An optimal migration plan should ensure that the variance of NF load is below the peak variance threshold th_{var} while resulting in minimal migration penalty.

Based on the above analysis, below we first propose the unified optimal flow migration calculation algorithm based on Integer Linear Programming (ILP) (§ IV-A). However, we observe that the ILP formulation cannot be quickly solved within a limited time. Therefore, we exploit the opportunity of relaxing the constraints in the three situations respectively. For each situation, we introduce an optimal formulation for flow selection, as well as a heuristic algorithm to guarantee the timeliness of `OFM` (§IV-B for NF scaling out, §IV-C for NF scaling in, and §IV-D for NF load balancing). Finally, as we use different algorithms for different situations, we propose a coordination mechanism to handle different combinations of the above situations (§IV-E).

A. Optimal Flow Migration Calculation

First, we propose an algorithm that could produce the optimal flow migration plan for the NFV elasticity control situations. Let N_s be the number of NF instances, and let N_d be the number of instances after migration. During NF scaling out, at most N_s new NF instances will be created in the worst case when all N_s NF instances are overloaded at the same time. During NF scaling in, at least 1 NF instance will remain. Therefore, $1 \leqslant N_d \leqslant 2 \times N_s$. We use $x_{fsd} \in \{0, 1\}$ as an indicator of whether flow f is migrated from source instance s to destination instance d. if $s = d$, a flow is considered to be not migrated and incurs zero penalty. For simplicity, we assume that recycling a VM brings a constant revenue benefit of $benefit$. Suppose there are m_s flows on instance s. The ILP formulation to solve x is:

$$min\ (Penalty - Benefit), \tag{13}$$

where

$$Penalty = \sum_{s=1}^{N_s} \sum_{d=1, d\neq s}^{N_d} (\sum_{f=1}^{m_s} x_{fsd} \times Penalty_{fsd}) \tag{14}$$

$$Benefit = (N_s - \sum_{d=1}^{N_d} sgn(\sum_{f,s} x_{fsd})) \times benefit \tag{15}$$

s.t.

(1) $x_{fsd} \in \{0,1\}\ for\ s\ \in [1, N_s],\ d\ \in [1, N_d],\ f\ \in [1, m_s]$

(2) $\sum_{d=1}^{N_d} x_{fsd} = 1\ for\ s\ \in [1, N_s],\ f\ \in [1, m_s]$

(3) $\sum_{s=1}^{N_s} \sum_{f=1}^{m_s} x_{fsd} \times buffer_f \leqslant Buffer_d\ for\ all\ d\ \in [1, N_d]$

(4) $load_s + \sum_{s=1}^{N_s} \sum_{f=1}^{m_s} x_{fid} \times size_f - \sum_{j=1}^{N_d} \sum_{f=1}^{m_s} x_{fsj} \times size_f \leqslant th_{safe}\ for\ s\ \in [1, N_s]$

(5)* $var(load_s + \sum_{i=1}^{N_s} \sum_{f=1}^{m_s} x_{fid} \times size_f - \sum_{j=1}^{N_d} \sum_{f=1}^{m_s} x_{fsj} \times size_f) \leqslant th_{var}$

where $size_f$ denotes the size of flow f, $buffer_f$ denotes the buffer required for migrating flow f, $Buffer_d$ denotes the buffer size in destination instance d, and $load_s$ denotes the current load of instance d.

The objective function (Eq. 13) minimizes the total migration cost. The penalty (Eq. 14) comes from the SLA violation penalty during migration, and the revenue benefit (Eq. 15) comes from the destruction of VM instances. We consider a VM is destructed if all flows on it are migrated away, while no flows are migrated to it. Constraint (1) regulates that a flow f is either migrated or not migrated, while Constraint (2) ensures that a flow is migrated once and only once to any instance including the instance it belongs to. Constraint (3) avoids buffer overflow in the destination instances. Constraint (4) ensures that no new hot spots are created. Constraint (5)* balances NF loads by constraining the variance of NF loads under threshold th_{var}.

However, with Constraint (1) to (4), the algorithm could generate at least one feasible solution as we could dynamically create or destruct NF instances to safely accommodate all flows. Meanwhile, Constraint (5) tries to limit the variance of NF instance load after migration. Under an extreme situation where there exist two NF instances, each carrying only one flow, and the load of the two instances are 50% and 10% respectively. The loads of the two instances cannot be balanced, leaving the ILP formulation unsolvable. Therefore, we exploit the *goal programming* [30] technique by allocating Constraint (1) to (4) high goal values and assigning a low goal value to Constraint (5). This ensures that the feasible

solutions that satisfy Constraint (1) to (4) can still be generated if Constraint (5) makes the problem unsolvable.

The above ILP formulation could handle the three situations simultaneously and generate the optimal migration plan. However, the ILP problem cannot be quickly solved due to two major reasons. First, we observe from Eq. 5 that $Penalty_{fsd}$ is a *piecewise function* depending on DT_{fsd}, making the ILP formulation unsolvable in a short time (*e.g.*, a few milliseconds [31]). Second, we allocate a parameter x_{fsd} for every single flow to indicate whether flow f is migrated from instance s to d or not. The number of parameters is linearly proportional to the number of NF instances (100s [32]) and flows on the instances (millions [33]). Massive number of parameters make it difficult to quickly solve the ILP formulation. However, according to the control goals of NF scaling out and load balancing in Section II, efficient calculation is required to quickly generate the migration plan. To address this challenge, we exploit the opportunity of relaxing the constraints in the three situations respectively and propose a fast algorithm for each situation. We introduce the design of the three fast algorithms in the rest of this section.

B. NF Scaling Out

1) Optimum Formulation for NF Scaling Out: When one or multiple NF instances are overloaded, `OFM Controller` scales out NF instances by migrating some flows from the overloaded instances to other instances or to the newly created instances. As analyzed in Section II, NF scaling out requires a quick hot spot alleviation without creating a new hot spot or leading to buffer overflow. We could identify overloaded instances and *consider them as the source instances of the flows to be migrated*. Therefore, the source instance set only includes the overloaded instances, which brings a significant reduction of the number of parameters. Furthermore, NF scaling out does not pursue balanced load after migration. The relaxation of constraints could also help accelerate the ILP solving. We use N_{os} to denote the number of overloaded NF instances. We have $N_s \leqslant N_d \leqslant N_s + N_{os}$. Based on above observations, we modify the objective function for NF scaling out as follows.

$$min\ (Penalty - Benefit), \tag{16}$$

where

$$Penalty = \sum_{s=1}^{N_{os}} \sum_{d=1, d\neq s}^{N_d} (\sum_{f=1}^{m_s} x_{fsd} \times Penalty_{fsd}) \tag{17}$$

$$Benefit = (N_s - \sum_{d=1}^{N_d} sgn(\sum_{f,s} x_{fsd})) \times benefit \tag{18}$$

We adopt Constraint (1) to (4) in the previous ILP formulation as the constraints here.

2) Fast Heuristic Algorithm for NF Scaling Out: However, the objective function above is still piecewise, making the ILP unsolvable in a limited time of a few milliseconds. According to the control goals of NF scaling out in Section II, efficient calculation is required to quickly alleviate the hot spot. Therefore, we propose a three-step heuristic to accelerate the calculation. We first select flows to be migrated away from the overloaded instances to ensure effective alleviation of the overload situation. Next, we calculate whether we can place the selected flows to other currently deployed instances. If current instances cannot hold all flows with respect to the th_{safe} constraint, we place as many flows into current instances as possible. Finally, we deploy new NF instances to accommodate the remaining flows that cannot be placed on current instances. The intuition is to avoid starting new NF instances to reduce the migration penalty. Next we introduce the three steps in detail.

Step 1 (Flow Selection): We start by picking the set of flows on each overloaded instance to alleviate the hot spot with minimum migration cost. Since we cannot pre-acknowledge the total number of flows to migrate, we assume that each flow is migrated individually and consumes a Single Flow Migrate Time (SFMT). The SFMT can be measured and calculated for different NF types, which will be introduced in Section V. For each overloaded instance s, we perform the following simple ILP algorithm.

$$min\ \sum_{f=1}^{m_s} x_{fs} \times Penalty_f, \tag{19}$$

where

$$Penalty_f = size_f \times (la_{processing,f} + SFMT - LA_f) \tag{20}$$

s.t.

(1) $x_{fs} \in \{0,1\}\ for\ s\ \in [1, N_{os}],\ f\ \in [1, m_s]$

(2) $load_s - th_{safe} \leqslant \sum_{f=1}^{m_s} x_{fs} \times size_f \leqslant load_s/2$

Constraint (1) regulates that a flow is either migrated or not migrated. Constraint (2) ensures that enough flows are selected for effective hot spot alleviation. According to our evaluation in Section V, the above ILP could be quickly solved within a few milliseconds.

Step 2 (Flow Placement Onto Current Instances): Next we place previously selected flows to current instances that are not overloaded. During placement, we try to place as many flows on current instances as possible so as to minimize the number of new instances to be deployed. In this process, the constraints are avoiding new hot spot creation and buffer overflow. Suppose there are m_f flows to be redistributed onto N_d current instances that are not overloaded. We present an ILP formulation for this step as follows.

$$max\ \sum_{d=1}^{N_d} \sum_{f=1}^{m_f} x_{fd} \times size_f \tag{21}$$

s.t.

(1) $x_{fd} \in \{0,1\}\ for\ d\ \in [1, N_d],\ f\ \in [1, m_f]$

(2) $\sum_{d=1}^{N_d} x_{fd} \leqslant 1\ for\ f\ \in [1, m_f]$

(3) $load_d + \sum_{d=1}^{N_d} \sum_{f=1}^{m_f} x_{fd} \times size_f \leqslant th_{safe}\ for\ d\ \in [1, N_d]$

(4) $\sum_{f=1}^{m_f} x_{fd} \times buffer_f \leqslant Buffer_d for\ d\ \in [1, N_d]$

We will demonstrate that the above formulation can also be solved in a short time of a few milliseconds in Section V. We examine the solution x to check if all flows are placed onto current instances. If $\sum_{d=1}^{N_d} x_{fd} = 1\ for\ f\ \in [1, m_f]$,

current instances can accommodate all flows and the algorithm finishes. Otherwise, we start new instances for the remaining flows.

Step 3 (New Instance Deployment): Finally, we deploy new instances to accommodate the remaining flows. We denote the number of remaining flows as m_{rf}. The objective of this step is to minimize the number of new instances to hold all remaining flows, so as to minimize the penalty. The load of the new instances after migration should be below th_{safe} and buffer overflow should be avoided. The solution is to iteratively create new instances to carry the maximum possible size of flows, until all flows are packed into NF instances. For each iteration, we formulate the flow packing process into the following ILP problem.

$$max \sum_{f=1}^{m_{rf}} x_f \times size_f \tag{22}$$

s.t.
(1) $x_f \in \{0,1\}\ for\ f\ \in [1, m_{rf}]$
(2) $\sum_{f=1}^{m_{rf}} x_f \times size_f \leqslant th_{safe}$
(3) $\sum_{f=1}^{m_{rf}} x_f \times buffer_f \leqslant buffer$

After each iteration, selected flows are removed from the remaining flow set, until all flows are removed. We will demonstrate that the above ILP problem can be quickly solved in Section V.

C. NF Scaling in

1) Optimum Formulation for NF Scaling in: `OFM Controller` would perform NF scaling in by migrating flows on underloaded instances to the remaining instances and shutting down the free VMs. `OFM Controller` applies an ILP algorithm to minimize the migration cost. Suppose there are N_{us} underloaded NF instances. Instance s carries m_s flows. x_{fsd} is an indicator of whether flow f is migrated from instance s to instance d. The ILP formulation to solve x aims at maximizing the revenue benefit (Eq. 13. The number of remaining instances N_d, satisfies $0 < N_d \leqslant N_s$. We omit the repeated ILP formulation here for brevity.

2) Fast Heuristic Algorithm for NF Scaling in: However, the objective function of the above formulation is still piecewise since only when all flows on one instance are migrated away can we destroy the instance and gain revenue benefit. To accelerate the calculation, we no longer migrate each flow individually to other instance. Instead, we *merge* current instances by migrating *all* flows on one underloaded instance to another instance and destroying the free instance. To achieve this goal, we propose a two-step heuristic algorithm for NF scaling in.

Step 1 (Beneficial Instance Identification): First we need to identify the NF instances whose destruction brings a higher revenue benefit than the migration cost of all flows on it. We filter instances that satisfy $benefit - penalty > 0$ and name them *candidate instances*. Suppose there are N_{cs} such instances.

Step 2 (Optimal Instance Merging): Next we merge the candidate instances to other instances with respect to the load constraint th_{safe} and buffer size constraint $buffer$. We use $x_{sd} \in 0, 1$ to indicate whether instance s is merged onto instance d. We model the problem as:

$$max \sum_{s=1}^{N_{cs}} \sum_{d=1}^{N_d} x_{sd} \times (benefit_s - Penalty_s) \tag{23}$$

s.t.
(1) $x_{sd} \in \{0,1\}\ for\ s \in [1, N_{cs}],\ d\ \in [1, N_d]$
(2) $\sum_{d=1}^{N_d} x_{sd} = 1\ for\ s \in [1, N_{cs}]$
(2) $load_d + \sum_{s=1}^{N_{cs}} x_{sd} \times size_s \leqslant th_{safe}$
(3) $\sum_{s=1}^{N_{cs}} x_{sd} \times buffer_s < buffer\ for\ d\ \in [1, N_d]$

By solving the above ILP formulation, we could calculate the optimized flow selection for NF scaling in within acceptable time. We evaluate the algorithm in Section V.

D. NF Load Balancing

1) Optimum Formulation for NF Load Balancing: Despite that NF load balancing could prevent potential NF overload situations, it is neither compulsive (like NF scaling out to alleviate the hot spot) nor immediately rewarding (like NF scaling in which brings revenue benefit). Therefore, we migrate flows on NF instances with heavier load, *i.e.* greater than the average load, to NF instances with lighter load *under the condition that no SLA violations occur*, incurring *zero migration costs*. Note that flows that are migrated away from one instance might be placed onto different NFs. `OFM Controller` is challenged to avoid generating hot spots and achieve a relatively balanced load. A straightforward solution is to divide flows into several groups of equal size, and redistribute all flows to all instances according to the division, in order to minimize the load variance of NF instances. We use x_{fsd} as a indicator of whether flow f is migrated from instance s to instance d. Here we only migrate flows f whose migration penalty is *zero*. We present the ILP formulation for this solution as follows.

$$min\ var(load_d + \sum_{s=1}^{N_s} \sum_{f=1}^{m_{fs}} x_{fsd} \times size_f - \sum_{f=1}^{m_{fd}} x_{fsd} \times size_f) \tag{24}$$

s.t.
(1) $x_{fsd} \in \{0,1\}\ for\ s\ \in [1, N_s],\ d\ \in [1, N_d],\ f\ \in [1, m_s]$
(2) $\sum_{d=1}^{N_d} x_{fsd} = 1\ for\ s\ \in [1, N_s],\ f\ \in [1, m_s]$
(3) $\sum_{s=1}^{N_s} \sum_{f=1}^{m_s} x_{fsd} \times buffer_f \leqslant Buffer_d\ for\ d\ \in [1, N_d]$

2) Fast Heuristic Algorithm for NF Load Balancing: Above global flow redistribution may lead to the migration of massive flows, which may bring negative impact on normal packet processing. Furthermore, the piecewise object function prevents the problem from being solved within a limited time. In response, our key idea is to fetch flows from instances whose loads are above the sum of the average load l_{avg} plus the standard deviation l_{stdev} and relocate the selected flows to instances with loads lower than $l_{avg} - lstdev$. We design a three-step heuristic, and the pseudo code is presented in Algorithm 1.

Step 1 (Instance Classification): We calculate the average load l_{avg} of NF instances and put the NF instances whose

Algorithm 1: Heuristic Algorithm for NF Load Balancing

Input: Flow Parameters: $size$, $SFMT$, LA, $la_{processing}$
Input: NF Parameters: $load$, l_{avg}, l_{stdev}, $Buffer$.
Output: Flows to Migrate, their sources, and Their Targets: $migration_plan[f, s, d]$.

$NFList_{heavy} = [\,]$, $NFList_{light} = [\,]$, $FlowList = [\,]$, $migration_{plan} = [\,]$;
// Step 1: Instance Classification
foreach $s \in [0, N_s]$ **do**
  if $load_s > l_{avg} + l_{stdev}$ **then**
    $NFList_{heavy}.append(s)$;
  if $load_s < l_{avg} - l_{stdev}$ **then**
    $NFList_{light}.append(s)$;
// Step 2: Flow selection
foreach $s \in NFList_{heavy}$ **do**
  $l_{extra,s} = load_s - (l_{avg} + lstdev)$;
  $totalSize = 0$;
  // Sort flows in according to flow sizes in decending order;
  foreach $f \in [1, m_{fs}]$ **do**
    $Penalty_f = size_f \times (la_{processing,f} + SFMT - LA_f)$;
    if $Penalty_f == 0$ **then**
      if $totalSize + size_f > l_{extra,s}$ **then**
        $break$;
      $FlowList.append(f, s)$;
      $totalSize = totalSize + size_f$;
// Step 3: Destination NF selection
foreach $d \in NFList_{light}$ **do**
  // Place flows to instance d using Eq. (22),
  // while ensuring $load_d$ does not exceed l_{avg};
  foreach $f \in FlowList$ **do**
    if $flow\ f$ should be placed on instance d **then**
      $migration_{plan}.append(f, s, d)$;

load is greater than $l_{avg}+l_{stdev}$ (heavily loaded instances) into $NFList_{heavy}$ and the others with loads lowever than $l_{avg} - lstdev$ (lightly loaded instances) into $NFList_{light}$.

Step 2 (Flow Selection): For each heavily loaded instance s, we calculate the extra NF load above $l_{avg}+l_{stdev}$ as $l_{extra,s}$. We select flows on instance s whose SLA would not be violated during migration. Considering the fact that flows migrated away from one instance might be placed on multiple other instances, we assume that each flow is migrated individually and consumes a SFMT. We store the qualified flows into the $FlowList$, and sort the flows with a *descending* order of flow sizes. Then we select flows one by one for migration, and stop when adding one more flow would overflow the extra load $l_{extra,s}$. The intuition here is to *quickly* reduce the load of the overloaded instance, since network traffic could vary significantly, and a fast load balancing is desired to avoid potential hot spots. Migrating large flows would reduce the total number of flows to migrate and accelerate the balancing.

Step 3 (Destination NF Selection): In this step, we mix up selected flows of all heavily loaded instances from Step 2 into the final $FlowList$ and split them onto light-loaded instances to achieve a balanced load. We fill up the processing load below l_{avg} of each lightly loaded instance with selected flows in $FlowList$ using the *bin-packing algorithm*. The reason why we use l_{avg} instead of $l_{avg}-l_{stdev}$ as the peak threshold is that we want to accommodate the maximum number of flows on lightly loaded instances. However, some flows in $FlowList$ might still not be assigned to any destination instance. These flows are placed back to the original NF instances.

Based on our evaluation in Section V, the above three-step algorithm could quickly generate a migration plan to achieve relatively balanced load among NF instances with no penalty.

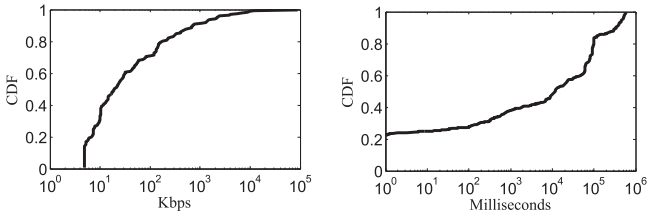
E. Coexistence of Multiple Situations

Finally, we discuss how `OFM Controller` reacts when two or three NFV elasticity control situations appear simultaneously. Our strategy is simple: when there is coexistence of multiple situations, we first strive to make network work well and then consider the benefit of shutting down VMs. First, if there are NFs overloaded. We choose to first handle NF overload since NF overload probably results in high SLA violation penalty. Specifically, we move flows from heavily loaded instances to lightly loaded instances, which could mitigate the underload and load imbalance situations to some extent. If there is no lightly loaded instance, we create new NFs. After NF overload is well handled, we need to decide which situation to handle first if underload and load imbalance situations coexist in the NFV network. NF load balancing prevents potential NF overload situations. However, NF scaling in merges instances for revenue benefit and may augment the load of heavily loaded instances, leading to a higher possibility of NF overload. Therefore, we choose to balance NF load first. Note that the process of balancing NF load will not incur NF overload. After load balancing, if any instance carries a processing load lower than th_{low}, `OFM` performs NF scaling in and recycles free VMs. To conclude, the reaction order to the three NFV elasticity control situations is: $overload \rightarrow load\ imbalance \rightarrow underload$.

V. Implementation and Evaluation

A. Implementation

We implemented the `OFM Controller` on top of the Floodlight [34] controller. Specifically, we maintain flow SLAs in a simple key-value storage data structure, and expose REST interfaces which can be used to dynamically append, modify, and delete SLAs. The *NF Status Collection* and *Flow Statistics Collection* modules collect NF loads and flow statistics through OpenFlow interfaces during runtime, which are utilized by the *Condition Detection* module to detect situations for NFV elasticity control. Note that for the deployment of `OFM`, no modification to OpenFlow is required. `OFM` simply requires the switches to be able to report flow statistics and forward flows according to flow table rules. However, NFs should expose state management interfaces to the controller for safe and efficient state migration, as required by [5], *etc*.



(a) Flow size distribution (b) Flow duration distribution

Fig. 5. LBNL/ICSI trace statistics (broken down by srcIP-dstIP pairs).

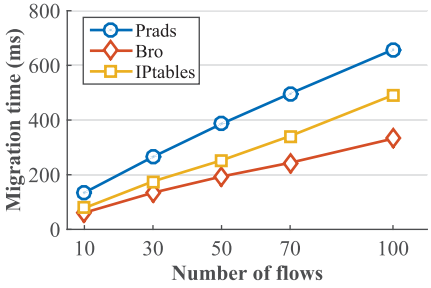


Fig. 6. Relationship between migration time and flow number.

The *Buffer Cost Analysis* module calculates required buffer costs, and the *Migration Cost Analysis* module calculates migration costs for different situations. Then, the *Optimal Migration Calculation* module would calculate the optimized set of flows using algorithms presented in Section III. To solve the ILP formulation for NF scaling in situations, we use *lpsolve*, a Java based mixed integer linear programming (MILP) solver [35].

B. Evaluation

We evaluate OFM based on a testbed with ten servers, each of which is equipped with two Intel(R) Xeon(R) E5-2690 v2 CPUs (3.00GHz, 10 physical cores), 256G RAM and two 10G NICs. The servers run Linux kernel 4.4.0-31. We use a server to run the OFM Controller, a server for Open vSwitch (OVS) [36], and eight servers for eight NF instances of the same type. To avoid affecting performance due to virtualization and demonstrate OFM's feasibility on real world physical devices, each software NF runs on *bare metal* servers without VM or Docker encapsulation, and all servers are connected to a Pica8 P-3922 *physical switch*. However, in a real world virtualization environment, operators often use VMs to carry software NFs, and multiple VMs may co-locate in the same physical server. In this case, *virtual switches* such as Open vSwitch are used for packet delivery within the server [26], [27]. We could use the virtual switch to collect flow statics and handle flow distribution across multiple identical instances.

For test traffic, we use a DPDK based packet generator that runs on the fifth server and is directly connected to the server carrying OVS. The generator sends and receives traffic to measure the forwarding latency. We use two types of traffic patterns including (1) *Real-world traffic trace*: we use the LBNL/ICSI enterprise trace [37], a typical traffic trace collected from real-world enterprise networks, whose flow size distribution and flow duration distribution are shown as Fig. 5, and (2) *Randomly generated traffic trace*, in which we create flows with random source and destination addresses. We configure the generator to create traffic according to the pattern type, flow number, and flow size.

We evaluate OFM with the following goals.

- Demonstrate the relationship between flow migration time and the number of flows to migrate. This justifies OFM's estimation of flow migration latency (Section V-B1).
- Demonstrate the timeliness and practicability of OFM status collection during runtime measurement (Section V-B2).
- Demonstrate that the OFM scaling out algorithm can find an optimized migration plan that effectively alleviates the hot spot (Section V-B3).
- Demonstrate that the OFM scaling in algorithm can find an optimal migration plan that brings the maximum migration benefit (Section V-B4).
- Demonstrate OFM load balancing algorithm's capability to effectively mitigate the load imbalance situation (Section V-B5).
- Demonstrate that OFM algorithms can efficiently solve the problem within limited calculation time and are scalable in real world network scales (Section V-B6).

1) Flow Migration Time: In this experiment, we examine the relationship between the flow migration time, $la_{migration}$ and the number of flows to migrate, n. We start two NFs instances of the same type on two servers. We randomly generate and send a different number of flows into one of the NF instances to create initial flow states in it. Then we configure the OFM Controller to perform flow and state migration of all flows on this instance to the other free instance, and measure the migration time. We have tested three types of NFs including Prads [38], Bro [39], and IPtables [40]. Prads maintains the state of flow meta data, end-host operating system and service details. Bro maintains the connection information of TCP, UDP, and ICMP. IPtables tracks the 5-tuple, TCP state, security marks, *etc.* for all active flows. For each NF type, we vary the number of flows to migrate from 10 to 100, and randomly vary the flow rate. Evaluation results are presented in Fig. 6 reals a linear positive correlation between the migration time and the number of flows for migration, regardless of the flow rate. Furthermore, we present the result of linear regression for each NF type.

- **Prads:** $la_{migration} = 86.982 + 5.7892 \times n,\ R^2 = 0.998$
- **Bro:** $la_{migration} = 39.205 + 2.9545 \times n,\ R^2 = 0.997$
- **IPtables:** $la_{migration} = 32.595 + 4.5222 \times n,\ R^2 = 0.998$

R^2 is a measure of accuracy of fit with a value of 1 denoting a perfect fit. Above regression expressions demonstrate a strong linear correlation between the migration time and the number of flows to migrate and can be utilized to estimate the migration latency. Especially, we could use the regression expression to estimate the SFMT (by assigning $n = 1$), which can be used for the scaling in and load balancing algorithms.

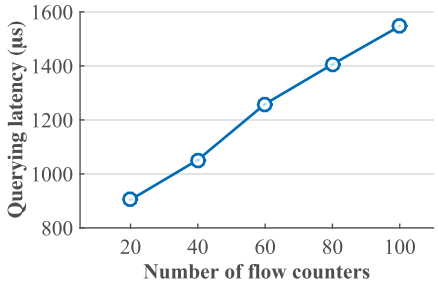


Fig. 7. Latency of the OFM Controller querying counters.

2) Timeliness of OFM: OFM collects flow statistics and calculates a migration plan during runtime. To ensure timeliness, statistics have to be gathered quickly and the algorithms should run efficiently. We enable the OFM Controller to query different numbers of flow counters from one underlying switch. As shown in Fig. 7, the OFM Controller could fetch 100 counters within 1.5 ms. Furthermore, as shown in the rest of this section, the entire control loop of statistics gathering and calculation could finish within 1 second, which demonstrates the timeliness and practicality of OFM.

3) NF Scaling out Algorithm: We evaluate the optimization effect and computation time of the NF scaling out algorithm using the Prads NF. In order to simulate NF overload situations, we assume that there are 10 NF instances, and set the number of overloaded instance as 2, 4, 6, 8. As the number of overloaded instances increases, the new instance deployment phase of the OFM scaling out algorithm is more likely to be triggered, which introduces a higher penalty. We set the th_{safe} as 60%, 55%, and 50%, respectively, and assume that in each situation, 5%, 10%, 15%, and 20% of flows (size-wise) need to be migrated for each overloaded instance. The decrease of th_{safe} indicates that current instances can hold fewer flows and new instances are more likely to be deployed. In order to quantify the migration cost, due to the lack of real world SLA settings for NFV networks, we set the SLAs of the flows by following the uniform random distribution in $[0.5\times(SFMT+la_{processing}), 1.5\times(SFMT+la_{processing})]$. This could ensure that some flow SLAs are violated during migration, and some are not.

We compare the OFM scaling out algorithm with an *optimal* algorithm and a *size-greedy* algorithm. The optimal solution produces the flow set that covers enough flows for migration with minimal cost. For the size-greedy solution, it picks the flow with the largest size until enough flows are selected to alleviate the hot spot, so that the flow selection process can finish as quickly as possible. For the *flow placement onto current instance* step, it chooses large flows to place on current instances over small flows. The intuition here is to maximize the size of flows placed on current instances. Finally, for the new instance deployment process, the size-greedy strategy always packs large sized flows into new instances so as to minimize the number of new instances. As shown in Fig. 8, OFM scaling out algorithm could reduce the migration cost to a large extent compared with the size-greedy algorithm, while suffering slightly higher cost compared with the optimal solution. This proves the effectiveness of the OFM algorithm.

4) NF Scaling in Algorithm: OFM exploits ILP to calculate an optimal solution that could minimize the migration cost for NF scaling in situations. In order to evaluate the optimization effect, we set th_{bottom} as 10%, 15%, 20%, 25% and th_{safe} as 40%, 50%, 60%. Above thresholds could be dynamically configured by the operator during the runtime. We scatter flows from the LBNL/ICSI enterprise trace to NF instances to ensure that a certain number of NF instances are underloaded. We configure the SLA of the flows following the same uniform random distribution as in the NF scaling out experiment.

The performance of this approach depends almost fully on the ILP formulation and solving. The ILP performance is mainly influenced by the number of underloaded NF instances of the same type. We set the number of underloaded instances as 10, 20, 30, 40, and 50 out of a total number of 100 instances. We use the Prads NF to perform the evaluation. We compare NF scaling in algorithm in OFM with a random solution that randomly picks NF pairs to merge while assuring that the total NF load after merging does not exceed the th_{safe}. As shown in Fig. 9, the OFM solution could achieve a linear increase in the migration benefit with the increase of the underloaded instance number and always outperforms the random solution significantly.

5) NF Load Balancing Algorithm: NF load balancing in OFM targets on reducing the load variance of NF instances belonging to the same NF type. Therefore, we vary the number of NF instances from 10 to 50, calculate the load variance of NF instances before (var_{before}) and after (var_{after}) the load balancing algorithm, and calculate the variance reduction $ratio = var_{before}/var_{after}$. We randomly arrange flows from the LBNL/ICSI enterprise trace on NF instances to ensure that no overload or underload situations happen.

We compare the NF load balancing algorithm in OFM with a *pairwise* solution that greedily pairs the overloaded and underloaded NF instances by sorting the load of NF instances and iteratively picking instances with the lowest and highest loads as pairs. It then redistributes flows between the two instances in each pair for load balancing. As shown in Fig. 10, the load variance of NF instances could be reduced by a factor of 1.5 to 2.5 by the OFM load balancing algorithm, which is 20% to 60% better than the *pairwise* solution.

6) Scalability of OFM Algorithms: We evaluate the scalability of OFM with respect to real world network parameters. According to the design of OFM algorithms, *the number of flows*, *packet rate*, and *the number of instances* may affect the computation time. As *packet rate* can be easily converted to the *number of flows* based on flow size distribution of real world network traffic [3], below we focus on the scalability of OFM algorithms with respect to different numbers of flows and numbers of instances. For test flows in our experiment, we use

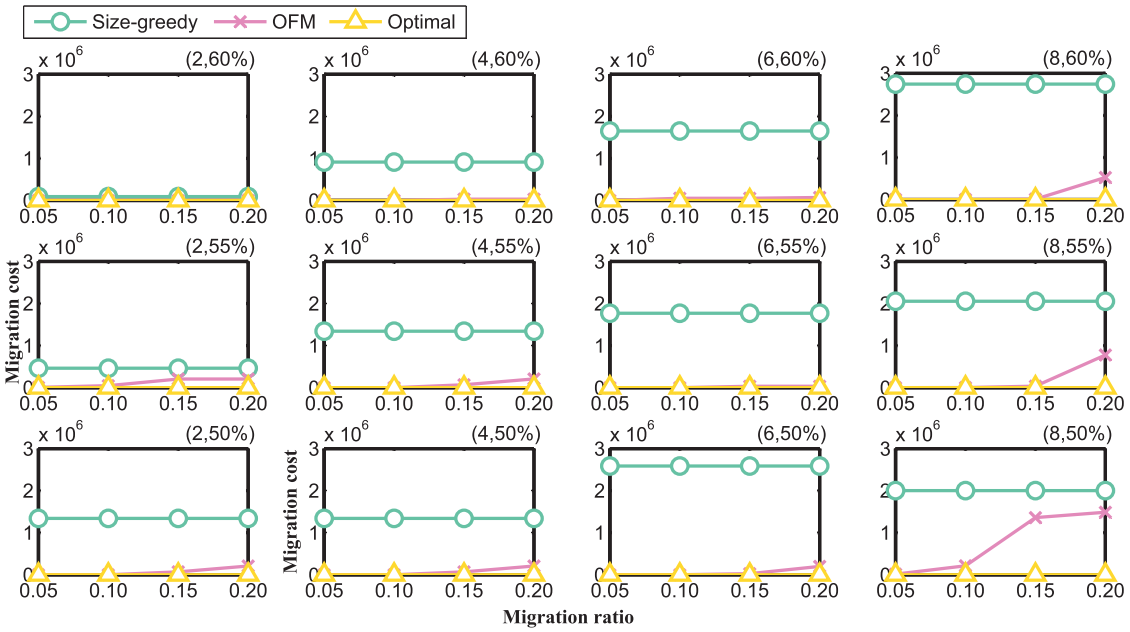


Fig. 8. Effect of OFM scaling out algorithm. We mark the ($\#overloaded\ instances$, th_{safe}) on subfigures' top-right corner.

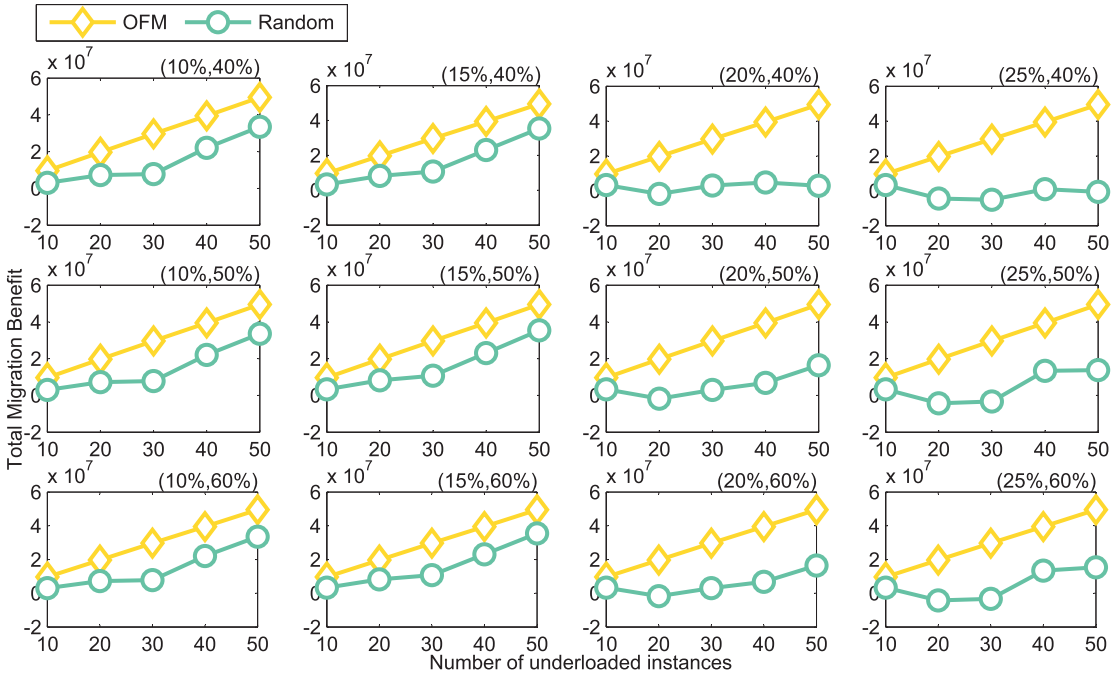


Fig. 9. Effect of OFM scaling in algorithm. We mark the (th_{bottom}, th_{safe}) on subfigures' top-right corner.

TABLE II
THE SCALABILITY OF OFM ALGORITHMS WITH RESPECT TO DIFFERENT PARAMETERS

Parameter	Scale out	Scale in	Load balancing
Number of flows	Demonstrated in Fig. 12(b).	Not related.	Demonstrated in Fig. 11(b).
Packet rate	*Packet rate* can be converted to *Number of flows* based on real world flow size distribution [3].		
Number of instances	Demonstrated in Fig. 12(a)	Demonstrated in Fig. 13	Demonstrated in Fig. 11(a)

real world flows from the LBNL/ICSI enterprise trace, whose statistics is presented in Fig. 5. We summarize the scalability evaluation of OFM in Table II.

For OFM scaling out algorithm, we first measure the computation time under different number of instances with the same evaluation setup as Section V-B3. As shown in Fig. 12(a),

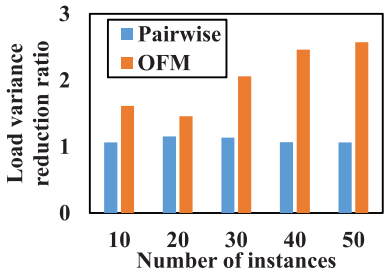


Fig. 10. Effect of OFM NF load balancing algorithm.

the OFM algorithm consumes less than 1 ms computation time, which occupies only a tiny portion of the entire migration time. Despite the size-greedy algorithm could finish more quickly within 0.1 ms, its optimization effect falls far behind OFM, as mentioned in Section V-B3. Second, we vary the number of flows on one instance and set the number of instances to one. As shown in Fig. 12(b), the computation time of OFM is below 0.1 *ms* under different numbers of flows and increases slowly as the number of flows increases. Moreover, OFM calculation time is significantly shorter than that of the optimal solution. Above results demonstrate the scalability and efficiency of the OFM scaling out algorithm.

For OFM scaling in algorithm, as the number of flows is not considered in algorithm design, we vary the number of underloaded instances to evaluate its scalability. According to Fig. 13, the computation time of the OFM NF scaling in algorithm is below 200 ms when handling 10 to 50 underloaded instances, which is acceptable in real-world network scale.

For OFM load balancing algorithm, we first set the average number of flows on each instance to 10, vary the number of instances, and measure the computation time. We then set the number of instances to 10, vary the average number of flows on each instance, and measure the computation time. As shown in Fig. 11(a) and Fig. 11(b), the computation time of the OFM algorithm is well below 100 μs for all parameter configurations, which demonstrates the scalability of the algorithm to quickly balance NF load in real world NFV networks.

VI. Related Work

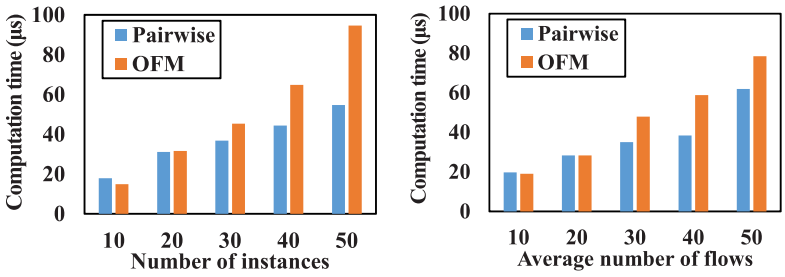
Some research efforts [5], [12]–[16], [41] have addressed the necessity of state migration to support NFV elasticity control. Split/Merge [14] and OpenNF [5] rely on a centralized control plane to buffer states during migration, while enhanced OpenNF [12] and other efforts [13], [15], [16], [41] perform state and packet transfer entirely in the data plane to improve scalability and performance. Above efforts mainly focus on *safe and efficient* state migration in NFV. In contrast, OFM addresses the challenge of *optimized flow selection* for NFV elasticity control to minimize penalty, and is complementary to above works.

Kablan *et al.* [42] proposed to extract state from NFs and store state in a data store layer, thus eliminating the necessity to migrate flows for NFV elasticity control. However, such a design could add to the NF processing latency by a maximum of 500 μs, which might be unbearable for latency sensitive applications [16], [17]. In comparison, OFM carefully considers the SLA requirements of flows and selects appropriate flows to migrate to achieve optimized NFV elasticity control.

A strawman solution for NFV elasticity control proposed in E2 [18] adopts a strategy of *migration avoidance*. Existing flows are still processed by previously assigned NF instances, while new flows are differentially handled. In this way, no flow migration occurs for NFV elasticity control. For NF scaling out, we simply instantiate a new NF instance and redirect new flows to it. For NF scaling in, we coalesce new flows on a few selected NFs and terminate other servers after all of their residual flows are served. For NF load balancing, we exploit consistent hashing to balance new flows. While the migration avoidance strategy introduces no migration penalty, it may still result in penalty. For NF scaling out, flows on existing NF instances may grow larger, which increases NF loads, degrades NF performance, and incurs SLA violations. For NF scaling in, many flows in data centers are long-lived flows that could last for minutes to hours [3]. The migration avoidance strategy prevents timely destruction of underloaded instances and therefore cannot bring as high revenue benefit as OFM. For NF load balancing, as flows on existing NF instances grow in sizes, NF instances may become overloaded and trigger NF scaling out, which would also introduce SLA violation penalty without careful flow selection.

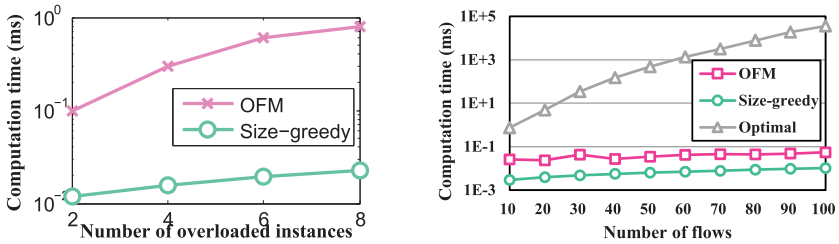
Flow migration, which is also called flow *handover*, is also a significant problem in 5G networks. Two typical recent researches [43], [44] have illustrated the necessity for the flow admission control from 5G macro cell networks to small cells. Above works targeted at minimizing the affection to the experience of other users during flow handover. Similarly, OFM aims at minimizing SLA violation penalty during NF elasticity control and flow migration. However, OFM differs from above works significantly. For 5G networks, the macro cell network itself can process all packets, while small cells are adopted to reduce power and cost. Therefore, migrating flows to small cells is optional, and an admission control mechanism is required to ensure user experience. However, in NFV networks, one NF instance is typically not adequate to process all flows, making elasticity control inevitable. OFM could optimize migration penalty.

Finally, comparing to the previous version of this paper [1], we have made substantive enhancements in this manuscript. First, we provide a unified ILP formulation for the three NFV elasticity control situations to generate an optimal flow migration plan. However, the optimal algorithm cannot be quickly solved in a limited time when not violating the control goals of the NFV elasticity control situations. Therefore, we need to design unique algorithms to handle the three situations respectively. Second, we have thoroughly modified the optimized flow migration algorithms for the three situations. For each situation, we first present the optimal algorithm and then the acceleration solution to effectively and efficiently generate the migration plan. Furthermore, we discuss how OFM reacts when two ore three NFV elasticity control situations happen



(a) With different number of instances. (b) With different number of flows.

Fig. 11. Scalability of OFM load balancing algorithm.



(a) With different number of instances. (b) With different number of flows.

Fig. 12. Scalability of OFM scaling out algorithm.

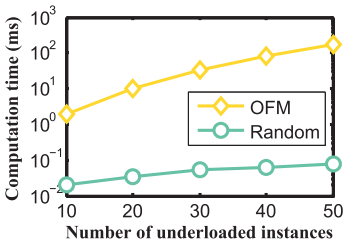


Fig. 13. Scalability of OFM NF scaling in algorithm.

simultaneously. Finally, we have updated the evaluation to demonstrate that OFM can enable NFV elasticity control with optimized flow migration.

VII. Conclusion and Future Work

We have proposed the design of OFM Controller to realize optimized flow migration for NFV elasticity control. We have analyzed different NFV elasticity control situations including NF scaling out, scaling in, and load balancing, and identified their control goals and challenges. After modeling buffer and migration cost, we have introduced a unified optimal formulation for all three situations, which cannot be solved within limited time. Thus, to achieve the unique control goals of each situation, we have designed a unique optimal formulation and a fast heuristic algorithm for each situation. Finally, we have introduced that OFM chooses to handle NF overload, load imbalance, and NF underload in order when two or three situations coexist in NFV. We have implemented the OFM Controller on top of NFV and SDN environments. Extensive evaluation results show that OFM could achieve near optimal flow migration within reasonable calculation time. As our future work, we will implement more NFs and integrate OFM into popular open-source NFV platforms to further demonstrate its effectiveness and efficiency.

References

[1] C. Sun, J. Bi, Z. Meng, X. Zhang, and H. Hu, "OFM: Optimized flow migration for NFV elasticity control," in *Proc. IEEE/ACM 21st Int. Symp. Qual. Service (IWQoS)*, Jun. 2018, pp. 1–10.

[2] R. Guerzoni, "Network functions virtualisation: An introduction, benefits, enablers, challenges and call for action, introductory white paper," in *Proc. SDN OpenFlow World Congr.*, 2012.

[3] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. 10th ACM SIGCOMM Conf. Internet Meas.*, 2010, pp. 267–280.

[4] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," *ACM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 123–137, 2015.

[5] A. Gember-Jacobson *et al.*, "OpenNF: Enabling innovation in network function control," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 163–174.

[6] P. Demestichas *et al.*, "5G on the horizon: Key challenges for the radio-access network," *IEEE Veh. Technol. Mag.*, vol. 8, no. 3, pp. 47–53, Sep. 2013.

[7] B. Blanco *et al.*, "Technology pillars in the architecture of future 5G mobile networks: NFV, MEC and SDN," *Comput. Standards Interfaces*, vol. 54, pp. 216–228, Nov. 2017.

[8] W. Haeffner, J. Napper, M. Stiemerling, D. R. Lopez, and J. Uttaro, "Service function chaining use cases in mobile networks," in *Proc. Internet Eng. Task Force*, 2016, pp. 1–26.

[9] S. Dutta, T. Taleb, and A. Ksentini, "QoE-aware elasticity support in cloud-native 5G systems," in *Proc. IEEE Int. Conf. Commun. (ICC)*, May 2016, pp. 1–6.

[10] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "Openstate: Programming platform-independent stateful openflow applications inside the switch," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 44–51, 2014.

[11] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan, "Flow-level state transition as a new switch primitive for SDN," in *Proc. ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, 2014, pp. 61–66.

[12] A. Gember-Jacobson and A. Akella, "Improving the safety, scalability, and efficiency of network function state transfers," in *Proc. ACM SIGCOMM Workshop Hot Topics Middleboxes Netw. Function Virtualization*, 2015, pp. 43–48.

[13] B. Kothandaraman, M. Du, and P. Sköldström, "Centrally controlled distributed VNF state management," in *Proc. ACM SIGCOMM Workshop Hot Topics Middleboxes Netw. Function Virtualization*, 2015, pp. 37–42.

[14] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/Merge: System support for elastic execution in virtual middleboxes," in *Proc. 10th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2013, pp. 227–240.

[15] Y. Wang, G. Xie, Z. Li, P. He, and K. Salamatian, "Transparent flow migration for NFV," in *Proc. IEEE 24th Int. Conf. Netw. Protocols (ICNP)*, Nov. 2016, pp. 1–10.

[16] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker, "Elastic scaling of stateful network functions," in *Proc. NSDI*, 2018, pp. 299–312.

[17] R. Gandhi *et al.*, "Duet: Cloud scale load balancing with hardware and software," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 27–38.

[18] S. Palkar *et al.*, "E2: A framework for NFV applications," in *Proc. 25th Symp. Operating Syst. Princ.*, 2015, pp. 121–136.

[19] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: Network processing as a cloud service," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 13–24, 2012.

[20] M. Alhamad, T. Dillon, and E. Chang, "Conceptual SLA framework for cloud computing," in *Proc. 4th IEEE Int. Conf. Digit. Ecosyst. Technol. (DEST)*, Apr. 2010, pp. 606–610.

[21] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proc. NSDI*, vol. 10, 2010, p. 19.

[22] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling flow management for high-performance networks," *Comput. Commun. Rev.*, vol. 41, no. 4, pp. 254–265, Aug. 2011.

[23] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: Fine grained traffic engineering for data centers," in *Proc. 7th Conf. Emerg. Netw. Exp. Technol.*, 2011, p. 8.

[24] *Openflow Switch Specification 1.4.0*, Open Netw. Found., 2013.

[25] S. Miteff and S. Hazelhurst, "NFShunt: A Linux firewall with openflow-enabled hardware bypass," in *Proc. IEEE Conf. Netw. Function Virtualization Softw. Defined Netw. (NFV-SDN)*, Nov. 2015, pp. 100–106.

[26] J. Martins *et al.*, "ClickOS and the art of network function virtualization," in *Proc. 11th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Seattle, WA, USA, 2014, pp. 459–473.

[27] J. Hwang, K. K. Ramakrishnan, and T. Wood, "NetVM: High performance and flexible networking using virtualization on commodity platforms," *IEEE Trans. Netw. Service Manage.*, vol. 12, no. 1, pp. 34–47, Mar. 2015.

[28] N. Foster *et al.*, "Frenetic: A network programming language," *ACM SIGPLAN Notices*, vol. 46, no. 9, pp. 279–291, 2011.

[29] L. Wu, S. K. Garg, and R. Buyya, "SLA-based resource allocation for software as a service provider (SAAS) in cloud computing environments," in *Proc. 11th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (CCGrid)*, May 2011, pp. 195–204.

[30] J. P. Ignizio, *Goal Programming and Extensions*. Lanham, MD, USA: Lexington Books, 1976.

[31] A. Schrijver, *Theory of Linear and Integer Programming*. Hoboken, NJ, USA: Wiley, 1998.

[32] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu, "NFP: Enabling network function parallelism in NFV," in *Proc. Conf. ACM Special Interest Group Data Commun.*, 2017, pp. 43–56.

[33] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching ASICs," in *Proc. Conf. ACM Special Interest Group Data Commun.*, 2017, pp. 15–28.

[34] *Project Floodlight*. Accessed: Aug. 28, 2018. [Online]. Available: http://www.projectfloodlight.org/floodlight

[35] M. Berkelaar, J. Dirks, K. Eikland, P. Notebaert, and J. Ebert. (2007). *Lpsolve: A Mixed Integer Linear Programming (MILP) Solver*. [Online]. Available: http://sourceforge.net/projects/lpsolve

[36] *Open vSwitch*. Accessed: Aug. 28, 2018. [Online]. Available: http://openvswitch.org

[37] *LBNL/ICSI Enterprise Tracing Project*. Accessed: Aug. 28, 2018. [Online]. Available: http://www.icir.org/enterprise-tracing

[38] *Passive Real-Time Asset Detection System*. Accessed: Aug. 28, 2018. [Online]. Available: http://gamelinux.github.io/prads/

[39] V. Paxson, "Bro: A system for detecting network intruders in real-time," *Comput. Netw.*, vol. 31, nos. 23–24, pp. 2435–2463, 1999.

[40] *Netfilter/Iptables Project*. Accessed: Aug. 28, 2018. [Online]. Available: http://www.netfilter.org

[41] Y. Lin, U. C. Kozat, J. Kaippallimalil, M. Moradi, A. C. K. Soong, and Z. M. Mao, "Pausing and resuming network flows using programmable buffers," in *Proc. SOSR*, 2018, p. 7.

[42] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless network functions: Breaking the tight coupling of state and processing," in *Proc. 14th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Boston, MA, USA, 2017, pp. 97–112.

[43] T. Taleb and A. Ksentini, "QoS/QoE predictions-based admission control for femto communications," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Jun. 2012, pp. 5146–5150.

[44] A. Ksentini, T. Taleb, and K. B. Letaif, "QoE-based flow admission control in small cell networks," *IEEE Trans. Wireless Commun.*, vol. 15, no. 4, pp. 2474–2483, Apr. 2016.

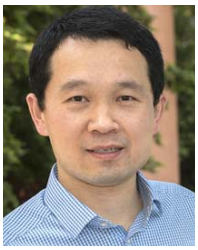
Chen Sun received the B.S. degree from the Department of Electronic Engineering, Tsinghua University, in 2014. He is currently pursuing the Ph.D. degree with the Institute for Network Sciences and Cyberspace, Tsinghua University. He has published papers in SIGCOMM, ICNP, SOSR, IWQoS, the *IEEE Communications Magazine*, and the *IEEE Network Magazine*. His research interests include Internet architecture, software-defined networking, and network function virtualization.

Jun Bi (S'98–A'99–M'00–SM'14) received the B.S., C.S., and Ph.D. degrees from the Department of Computer Science, Tsinghua University, Beijing, China. He is currently a Changjiang Scholar Distinguished Professor of Tsinghua University and the Director of the Network Architecture Research Division, Institute for Network Sciences and Cyberspace, Tsinghua University. His current research interests include Internet architecture, SDN/NFV, and network security. He successfully led 10 of research projects, published over 200 research papers and 20 Internet RFCs or drafts, owned 30 innovation patents, received national science and technology advancement prizes, IEEE ICCCN outstanding leadership award, and best paper awards. He is a Distinguished Member of the China Computer Federation.

Zili Meng is currently pursuing the bachelor's degree with the Department of Electronic Engineering, Tsinghua University. He has authored or co-authored papers in SIGCOMM, SOSR, IWQoS, and ICC. His research interest includes network function virtualization and software defined networks.

Tong Yang (M'18) received the Ph.D. degree in computer science from Tsinghua University in 2013. He visited the Institute of Computing Technology, Chinese Academy of Sciences, China, from 2013 to 2014. He is currently an Assistant Professor with the Computer Science Department, Peking University. He has published over 40-refereed technical papers, many of which appeared in top conferences and journals, including SIGCOMM, SIGMOD, VLDB, ATC, ICDE, ToN, INFOCOM, SOSR, and ICNP. His research interests include routers and switches, Bloom filters, sketches, and Openflow.

Xiao Zhang received the B.S. degree from the Department of Computer Science and Technology, Beijing University of Posts and Telecommunications, Beijing, China, in 2016. She is currently pursuing the master's degree with the Institute for Network Sciences and Cyberspace, Tsinghua University. Her research interests include software-defined networking and network functions virtualization.



Hongxin Hu (S'10–M'12) received the Ph.D. degree in computer science from Arizona State University, Tempe, AZ, USA, in 2012. He is currently an Assistant Professor with the Division of Computer Science, School of Computing, Clemson University. He has published over 100 refereed technical papers, many of which appeared in top conferences and journals. His current research interests include security in emerging networking technologies, security in Internet of Things (IoT), security and privacy in social networks, and security in cloud and mobile computing. He was a recipient of the Best Paper Awards from ACM CODASPY 2014 and ACM SIGCSE 2018, and the Best Paper Award Honorable Mentions from ACM SACMAT 2011, IEEE ICNP 2015, and ACM SACMAT 2016.