`OFM`: Optimized Flow Migration for NFV Elasticity Control

Chen Sun[†], Jun Bi[†], Zili Meng[†], Xiao Zhang[†], Hongxin Hu[§]
[†]Institute for Network Sciences and Cyberspace, Tsinghua University
[†]Department of Computer Science, Tsinghua University
[†]Beijing National Research Center for Information Science and Technology (BNRist)
[§]School of Computing, Clemson University

***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, load balancing, etc. 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.**

I. Introduction

Network Function Virtualization (NFV) [1] 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 [2], [3]. To realize NFV elasticity control such as NF scaling [4], [5], [6] or NF load balancing [5], [7], [8], flows need to be distributed across NF instances, requiring an efficient and flexible approach to control traffic steering. Currently, Software Defined Networking (SDN) [9] is used to steer flows through NFs to enforce network policies [4], [7]. Together, NFV and SDN can support dynamic flow distribution across NF instances.

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 [12], [5], [13], [14], [15] 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 SLAs (such as flows of algorithmic stock trading or high performance distributed memory caches [16]), 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 penalties 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

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	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

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, `OFM` could select proper flows to achieve *control goals* while minimizing the *migration costs* and avoiding *buffer overflow*. (§ III)
- 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. (§ IV)

II. Elasticity Control Situations Analysis

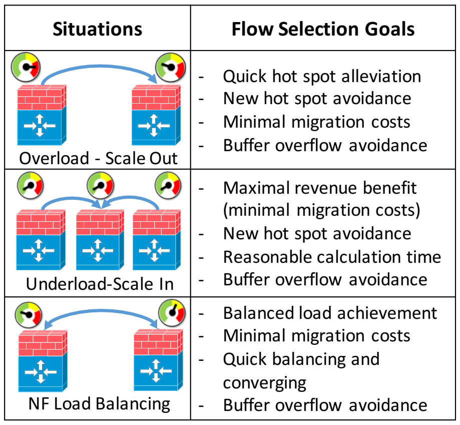
This section first summarizes the situations where flow migration is required for NFV elasticity control. Then we analyze the control goals and constraints of each situation as well as the design challenges, which guide the design of `OFM`.

A. NFV Elasticity Control Situations

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

NF scaling out: This happens anytime when the load of an NF instance exceeds the NF processing load threshold [5], [8], [14], [17]. By dynamically deploying NF instances in NFV, 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 the newly created one. However, flows on the overloaded instance have various SLA constraints and sizes. In order to achieve control goals, 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.

NF scaling in: To save resources and achieve energy efficiency, when 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], [8], [17]. However, flow migration incurs additional latency and could violate SLA constraints of some flows. Therefore, we should select proper NF instances to destroy to achieve maximum revenue benefit and minimum migration costs.



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 load achievement - Minimal migration costs - Quick balancing and converging - Buffer overflow avoidance

Fig. 1: Flow selection goals for different control situations

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.

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].

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 and quickly [18] 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.

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.

NF *scaling in*: As merging multiple instances into fewer ones and destroying free Virtual Machines (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.

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 [8] 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 §V.

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`.

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*).

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

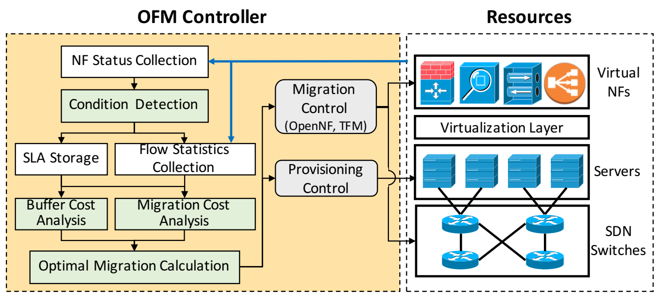


Fig. 2: `OFM Controller` components and workflow

for flow migration latency based on the number of flows to migrate and use it to calculate migration costs (§ III.*C*).

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 respectively. However, *massive parameters* including NF load, flow size (elephant or mice flows as defined in [19], [20]), 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 unique algorithms for different situations with respect to all above parameters. (§ III.*D*).

III. `OFM` Design

To address the above challenges, we design the `OFM Controller` to realized 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 each NF instance and detects traffic overload, underload, and imbalance *conditions*. At the same time, the `OFM Controller` collects the statistics of flows 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* (III.*D*) 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 [12], [5], [14], [15].

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 [21], 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 § IV-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.

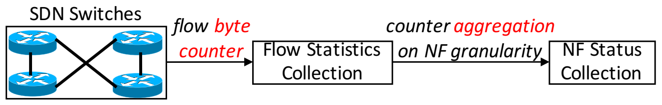

SDN Switches
flow byte counter
Flow Statistics Collection
counter aggregation on NF granularity
NF Status Collection

Fig. 3: NF status collection in `OFM`

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 obtain 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 [22]. 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. Directly querying counters cannot provide flow-level byte counters. Therefore, we utilize OpenFlow's multi-stage flow tables [22], 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.

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

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 [23] and cause penalty [24]. 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* [24]. In comparison, the service provided by NFV is advanced *packet processing* by NFs including firewall, IDS, VPN, load balancing, etc [8], [17]. Latency related SLAs should then regulate the maximum latency for each flow processed by NFV networks.

Suppose there are m flows on NF instance j. The latency SLA of a flow k is LA_k. Therefore, the latency la_k of this flow should satisfy: $la_k \leq LA_k\ for\ k \in [1, m]$. 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:

$$la_{migration,k} \leq LA_k - la_{processing,j}\ for\ k \in [1, m] \tag{2}$$

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 [24], 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:

$$Penalty = \alpha + \beta \times buffer \times DT \tag{3}$$

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:

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

The migration penalty of flow k could be modeled as:

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

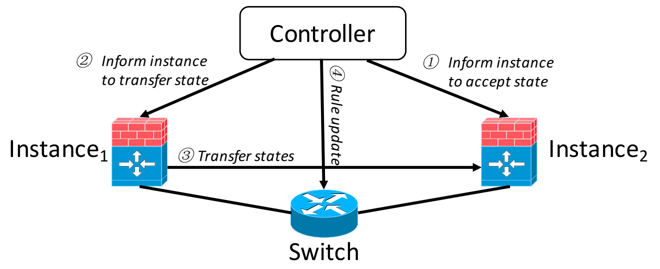


Fig. 4: Flow migration workflow from [12]

Next, we need to estimate the migration time of a flow to calculate the SLA violation penalty. Fig. 4 shows the 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:

$$la_{migration,k} = t1 + t2 + ts_k + tu \tag{6}$$

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 IV 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:

$$ts_k = \gamma + \eta \times f_n \tag{7}$$

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:

$$DT_k = (t1+t2+tu+\gamma+\eta * f_n) + la_{processing,k} - LA_k \tag{8}$$

$$buffer_k = size_k \times (t1 + t2 + tu + \gamma + \eta \times f_n) \tag{9}$$

$$cost_k = Penalty_k = \begin{cases} \alpha + \beta \times buffer_k \times DT_k & DT_k > 0 \\ 0 & DT_k = 0 \end{cases} \tag{10}$$

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 [24]) 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:

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

$$cost_j = \sum_{k=1}^{n} Penalty_k - benefit_j \tag{12}$$

D. Optimal Flow Migration Calculation

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, which motivates us to design different mechanisms for these three situations respectively. NF scaling out requires *quick* hot spot alleviation with *minimal* migration costs. Therefore, we design a greedy algorithm for NF scaling-out that could quickly generate an optimized migration plan. NF scaling-in pursues *maximal* revenue benefit within a *reasonable* calculation time. In response, we propose an Integer Linear Programming (ILP) algorithm to bring the maximum benefit. Finally, NF load balancing aims to achieve relatively balanced load and minimal migration costs. For these purposes, we design a three-step heuristic for NF load balancing to effectively mitigate the load imbalance situation.

1) NF Scaling Out: When an NF instance is overloaded, `OFM Controller` performs NF scaling out by creating a new instance and migrating some flows to it. A strawman solution for NF scaling out is to migrate *half* of the traffic load to the newly created instance, in order to achieve a balanced load while alleviating the hot spot. However, flows on the overloaded instance may have tight latency SLAs, and migrating half of these flows will incur large SLA violation penalties.

Actually, the basic control goal of NF scaling out is to migrate some flows away to drop the NF load *below the peak threshold*. To achieve this control goal, we introduce the *peak safe threshold* $th_{safe,j}$ for NF instance j, which regulates the peak load of the overloaded instance after scaling out. For example, suppose the peak threshold $th_{peak,j} = 80\%$ of the total capacity while $th_{safe,j} = 60\%$. Suppose there is an overloaded (80%) instance. Instead of having to migrate 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. Therefore, we formulate the ILP algorithm for NF scaling out as follows.

Suppose NF instance $j1$ with load l_{j1} is overloaded and some of its flows will be migrated to a new instance $j2$ with buffer size $Buffer_{j2}$. Suppose there are m_{j1} flows on instance $j1$, namely $f1, ..., f_{m_{j1}}$. $x_k \in \{0, 1\}$ is an indicator of whether flow k is selected. ILP formulation to solve x is:

$$min \sum_{k=1}^{m_{j1}} x_k \times Penalty_k \tag{13}$$

s.t.
(1) $x_k \in \{0, 1\} \ for \ all \ j \ \in [1, m_{j1}]$
(2) $\sum_{k=1}^{m_{j1}} x_k \times buffer_k \leq Buffer_{j2}$

(3) $(load_k - th_{safe,k}) \times capacity < \sum_{k=1}^{m_{j1}} size_k < load_k/2$

Constraint (1) regulates that A flow is either migrated or not migrated. Constraint (2) avoids buffer overflow in the destination instance. Constraint (3) ensures that enough flows are selected with minimum penalty.

Algorithm 1: *Penalty Greedy* Algorithm for Scaling Out

```
Input: Flow Parameters: size, $SFMT$, $LA$, $la_{processing}$
Input: NF Parameters: load, $th_{safe}$, $th_{top}$, $Buffer_{dst}$.
Output: Flows to Migrate: $f_m$.
1  $size_{fm} = 0$ ;
2  foreach $k \in m_{j1}$ do
3      // Calculate migration penalty for each flow
4      $Penalty[k] = size[k] \times (la_{processing}[k] + SFMT - LA[k])$ ;
5  while $true$ do
6      // Find the flow with the minimum penalty
7      $index = find\_min\_index(Penalty)$ ;
8      $Penalty[index] = MAX\_VALUE$ ;
9      $size_{fm} = size_{fm} + size[index]$ ;
10     if $size_{fm} > Buffer_{dst}$ then
11         // Buffer in the dst NF is overloaded
12         break ;
13     if $size_{fm} > load - th_{safe}$ & $find\_min(Penalty) > 0$ then
14         // Enough flows are selected. Selecting
           one more flow introduces extra penalty
15         break ;
16     if $size_{fm} > load/2$ then
17         // Half load has been selected
18         break ;
19     $f_m$.append($index$);
```

However, we observe from Eq. 5 that $Penalty_k$ is a *piece-wise function* depending on DT_k, making the ILP problem unsolvable in a short time of a few milliseconds [25]. However, according to the control goals of NF scaling out in Section II, efficient calculation is required to quickly alleviate the hot spot. Therefore, we exploit the *penalty greedy* algorithm to accelerate the calculation, as shown in Algorithm 1. 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 IV. We use the migration penalty (line 4) as the greedy variable. We greedily pick flows with small penalties, and accumulate the total size of selected flows until constraints are violated (lines 10, 13, and 16). Finally, we select a optimized set of flows for NF scaling out.

2) NF Scaling in: Suppose there are n $(n > 1)$ 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. `OFM Controller` applies an ILP algorithm to minimize the migration cost. Suppose NF instances $j_1, ..., j_n$ with load $l_1, ..., l_n$ are underloaded. Instance $j_h (1 \leq h \leq n)$ carries m_h flows. x_{sd} is an indicator of migrating *all flows* on instance j_s to instance x_d and destroying instance j_s. Instance j_h has a buffer size of $buffer_{j_h}$. The ILP formulation to solve x is presented below.

$$min \sum_{s=1}^{n} \sum_{d=1}^{n} x_{sd} \times (\sum_{k=1}^{m_{j_s}} Penalty_k - benefit_s) \quad (14)$$

s.t.

(1) $x_{sd} \in \{0,1\}\ for\ s \in [1,n],\ d \in [i,n]$

(2) $x_{ss} = 0\ for\ s \in [1,n]$

(3) $\sum_{s=1}^{n} x_{sd} + \sum_{s=1}^{n} x_{ds} \leq 1$

(4) $l_d + \sum_{s=1}^{n} x_{sd} \times l_s < th_{safe,d},\ d \in [1,n]$

(5) $\sum_{s=1}^{n} x_{sd} \times (\sum_{k=1}^{m_s} buffer_k) < buffer_{j_d},\ d \in [1,n]$

Constraint (1) regulates that instance s is either merged onto instance d or not. Constraint (2) ensures that an instance cannot be merged to itself. Constraint (3) ensures that an instance can either be merged onto the other one, or accommodate the other instance, or neither happens. Constraint (4) ensures that remaining NF instances after merging are not overloaded. The final constraint ensures that the buffer of each remaining instance is not overflowed.

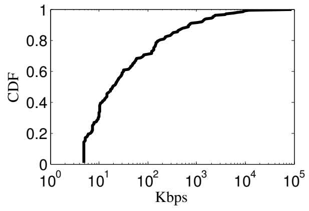
By solving the above ILP formulation, we could calculate the *optimal* flow selection for NF scaling in within acceptable time. We evaluate this algorithm in Section IV.

3) 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. However, this may lead to the migration of massive flows and incur large penalty. Thus, our intuition is to fetch flows from instances whose loads are above average and relocate the extra flows to instances with lower loads. We design the following three-step heuristic.

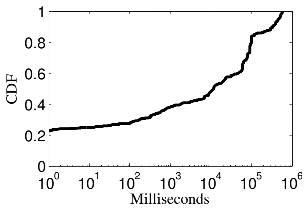
Step 1: Instance classification. We calculate the average load l_{avg} of NF instances and put the NF instances whose load is greater than l_{avg} (heavily loaded instances) into $NFList_{heavy}$ and the others (lightly loaded instances) into $NFList_{light}$.

Step 2: Flow selection. For each heavily loaded instance j, we calculate the extra NF load above average as $l_{extra,j}$. We select flows on instance j 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 $FlowList_j$, and sort the flows with a *descending* order by flow sizes. Then we select flows one by one for migration, and stop when adding one more flow would overflow the extra load $l_{extra,j}$. 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



(a) Flow size distribution



(b) Flow duration distribution

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

the final $FlowList$ and split them onto light-loaded instances to achieve a balanced load. We fill up the processing load below average $l_{below,j}$ of each lightly loaded instance with selected flows in $FlowList$ using the *bin-packing algorithm*. Finally, some flows in $FlowList$ might not be assigned to any destination instance. In this situation, these flows are placed back to the original NF instances.

Based on our evaluation in Section IV, the above three-step algorithm could quickly generate a migration plan to achieve relatively balanced load among NF instances.

IV. Implementation and Evaluation

A. Implementation

We implemented the `OFM Controller` on top of the Floodlight [26] 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. 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 [27].

B. Evaluation

We evaluate `OFM` based on a testbed with several 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. The topology for evaluation is depicted in Fig. 4. We use a server to run the `OFM Controller`, a server for Open vSwitch (OVS) [28], and the rest eight servers that are directly connected to the OVS server for eight NF instances of the same type. To avoid affecting performance, each software NF runs on bare metal without VM or Docker encapsulation.

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 [29], 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.

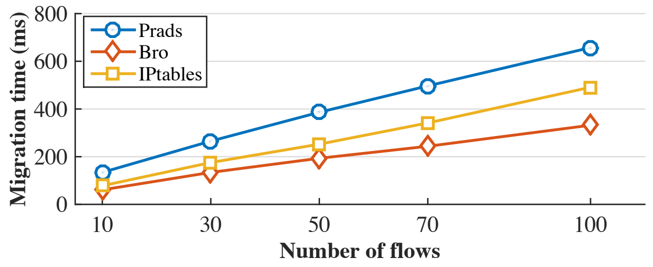


Fig. 6: Relationship between migration time and flow number

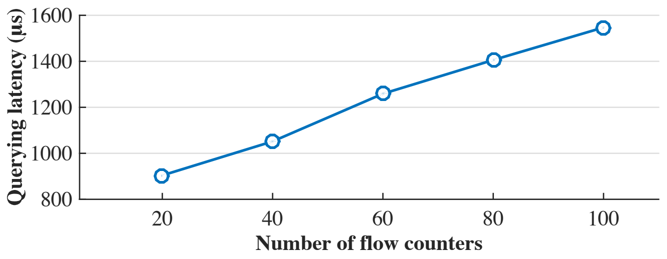
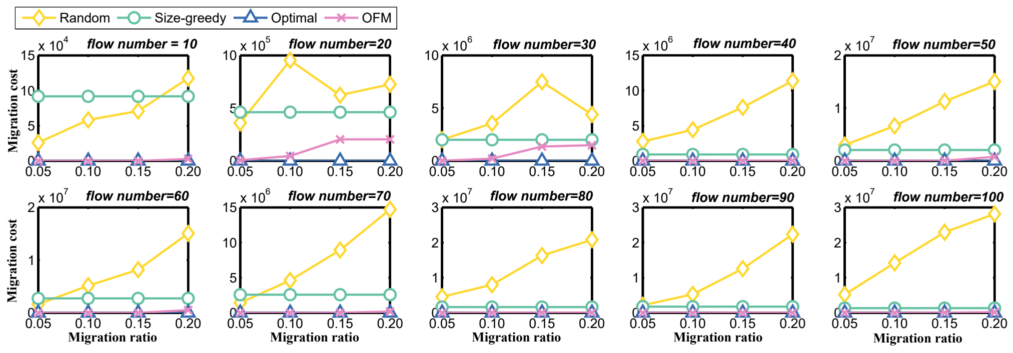


Fig. 7: Latency of the `OFM Controller` querying counters

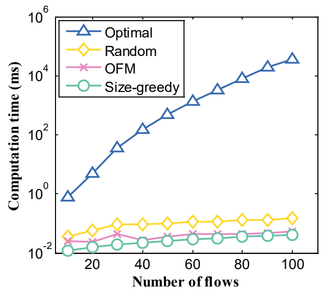
We evaluate `OFM` with the following goals.

- Demonstrate the relationship between flow migration time and the number of flows to migrate. This justifies `OFM`'s estimation of flow migration latency.
- Demonstrate that the `OFM` scaling out algorithm can find an optimized migration plan that effectively alleviates the hot spot within limited calculation time.
- Demonstrate that the `OFM` scaling in algorithm can find an optimal migration plan that brings the maximum migration benefit within acceptable calculation time.
- Demonstrate `OFM` load balancing algorithm's capability to effectively mitigate the load imbalance situation within limited calculation time.

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 [30], Bro [31], and IPtables [32]. 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



(a) Effect of NF scaling out



(b) Computation time

Fig. 8: Evaluation of the `OFM` NF scaling out algorithm

results are presented in Fig. 6, which reveals a linear positive correlation between the migration time and the number of flows to 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.

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. We vary the number of flows from 10 to 100 and place them on one NF instance. In order to simulate NF overload situations and evaluate the optimization effect, we assume that the NF load l reaches 80% of the entire capacity, and set the th_{safe} as 75%, 70%, 65%, and 60%, respectively, indicating that 5%, 10%, 15%, and 20% flows (size-wise) need to be migrated to alleviate the NF overload situation. 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 $[0.5 \times (SFMT + la_{processing}), 1.5 \times (SFMT + la_{processing})]$. Above SLA configuration could ensure that some flow SLAs are violated during migration, and some are not.

We compare the `OFM` scaling out algorithm with three strawman solutions, including a *random* algorithm, a *size-greedy* algorithm, and an *optimal* solution. The random algorithm randomly picks flows one by one until the total size of selected flows reaches the migration percentage. The size-greedy solution always picks the flow with the largest size and checks if both constraints (2) and (3) in Eq.13 are satisfied. If constraint (3) is satisfied (source instance is no longer overloaded) but constraint (2) is not (migration creates a new hotspot), it abandons the current selected flow and picks the next largest flow. The optimal solution exhaustively calculates the total size and migration cost of all possible flow combinations, and finds the combination, which covers enough flow size for migration with the minimal cost.

As shown in Fig. 8(a), the `OFM` scaling out algorithm could reduce the migration cost to a large extent compared with random and size-greedy algorithms, while suffering slightly higher cost compared with the optimal solution. However, Fig. 8(b) shows that the `OFM` algorithm consumes less than 0.1 ms computation time, while the optimal solution could run for around 10 seconds for 100 flows. The flow selection in `OFM` occupies only a tiny portion of the entire migration time, which demonstrates the efficiency of the algorithm. Note that the random algorithm takes a longer calculation time than the `OFM` algorithm, as we run the random algorithm 10 times and select the result with the minimum migration cost.

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 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}. We present the results in Fig. 9 and 10. As shown in Fig. 9, the `OFM` solution could achieve a linear increase in the

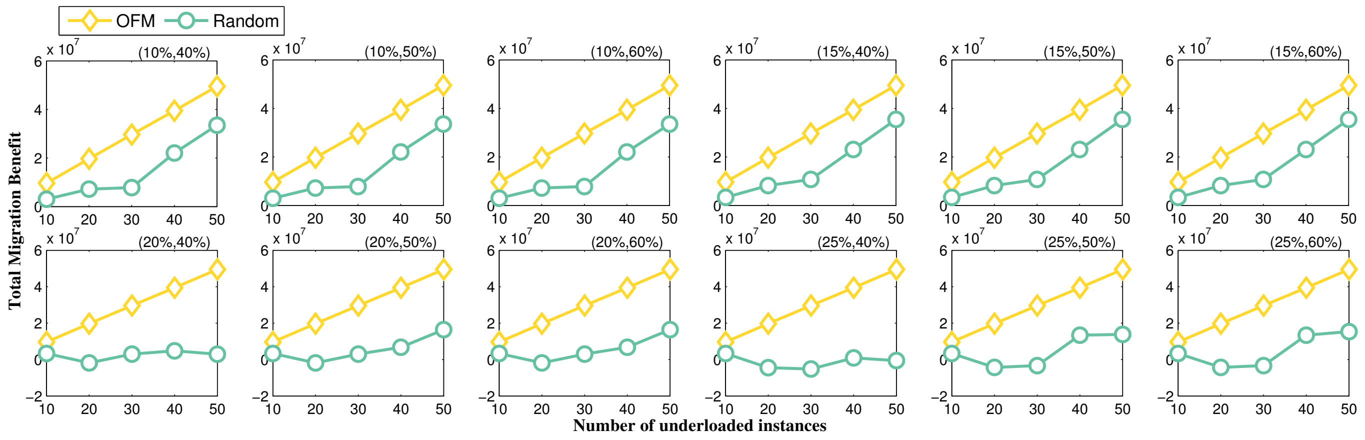


Fig. 9: Effect of `OFM` NF scaling in algorithm. We mark the (th_{bottom}, th_{safe}) on the top-right corner of each figure.

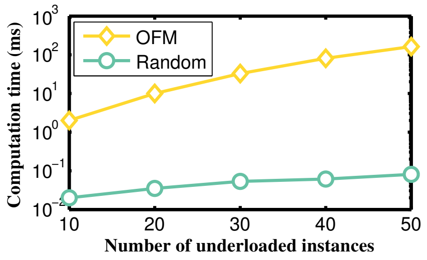
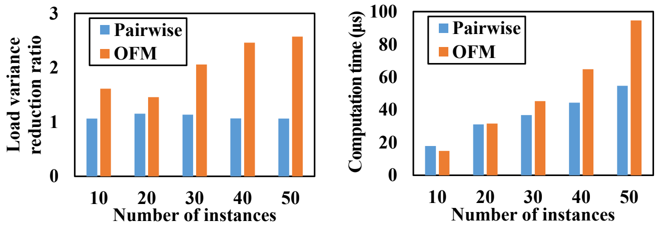


Fig. 10: Computation time of `OFM` NF scaling in algorithm



(a) Effect of NF LB (b) Computation time

Fig. 11: Evaluation of `OFM` NF load balancing algorithm

migration benefit with the increase of the underloaded instance number and always outperforms the random solution largely in different cases. According to Fig. 10, the computation time of the `OFM` NF scaling in the algorithm is below 200 ms when handling 10 to 50 underloaded instances, which is acceptable for real-world networks.

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. 11(a), 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. Besides, as illustrated in Fig. 11(b), the computation time of the `OFM` algorithm is well below 100 μs for balancing 50 NF instances, which could quickly balance NF loads.

V. Related Work

Some research efforts [5], [12], [13], [14], [15] 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] perform state and packet transfer directly among NF instances to improve scalability and performance. Above efforts mainly focus on *safe state* migration in NFV. In contrast, `OFM` addresses the challenge of *optimized flow* migration for NFV elasticity control. Murad et al. [33] 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]. In comparison, `OFM` carefully considers the SLA requirements of flows and selects appropriate flows to migrate to reduce costs, which could support optimized NFV elasticity control.

A strawman solution for NFV elasticity control proposed in E2 [8] adopts a strategy of *migration avoidance*. Existing flows are still processed by the previously assigned NF instance, while new flows are differentially handled. In this way, no flow migration occurs for NFV elasticity control. For NF scaling out, we could 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 during runtime. 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 [2]. 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.

VI. Conclusion and Future Work

In this paper, 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 in detail, identified control goals and challenges for different situations, and carefully designed algorithms to address each situation. We have implemented the `OFM Controller` on top of NFV and SDN environments and evaluated its effectiveness and efficiency. As our future work, we will implement more NFs in `OFM` and integrate `OFM` into popular open-source NFV platforms to further demonstrate its effectiveness and efficiency. Furthermore, we will evaluate the optimality of the algorithms from a theoretical point of view.

VII. Acknowledgement

This research is supported by National Key R&D Program of China (2017YFB0801701), the National Science Foundation of China (No.61472213) and Tsinghua University Initiative Scientific Research Program. Jun Bi (junbi@tsinghua.edu.cn) is the corresponding author.

References

[1] R. Guerzoni *et al.*, "Network functions virtualisation: an introduction, benefits, enablers, challenges and call for action, introductory white paper," in *SDN and OpenFlow World Congress*, 2012.

[2] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM, 2010, pp. 267–280.

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

[4] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, V. Sekar, and A. Akella, "Stratos: A network-aware orchestration layer for virtual middleboxes in clouds," *arXiv preprint arXiv:1305.0209*, 2013.

[5] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "Opennf: Enabling innovation in network function control," in *Proceedings of the 2014 ACM conference on SIGCOMM*. ACM, 2014, pp. 163–174.

[6] V. Mathew, R. K. Sitaraman, and P. Shenoy, "Energy-aware load balancing in content delivery networks," in *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2012, pp. 954–962.

[7] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "SIMPLE-fying middlebox policy enforcement using SDN," in *Proceedings of the ACM SIGCOMM 2013 conference (SIGCOMM'13)*. ACM, 2013.

[8] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: a framework for nfv applications," in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 121–136.

[9] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[10] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "Openstate: programming platform-independent stateful openflow applications inside the switch," *ACM SIGCOMM Computer Communication Review*, 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 *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'14)*, 2014.

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

[13] B. Kothandaraman, M. Du, and P. Sköldström, "Centrally controlled distributed vnf state management," in *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. ACM, 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 *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 227–240.

[15] Y. Wang, G. Xie, Z. Li, P. He, and K. Salamatian, "Transparent flow migration for nfv," in *Network Protocols (ICNP), 2016 IEEE 24th International Conference on*. IEEE, 2016, pp. 1–10.

[16] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang, "Duet: cloud scale load balancing with hardware and software," in *Proceedings of the 2014 ACM conference on SIGCOMM*. ACM, 2014, pp. 27–38.

[17] 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 Computer Communication Review*, vol. 42, no. 4, pp. 13–24, 2012.

[18] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14). Seattle, WA: USENIX Association*, 2014, pp. 459–473.

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

[20] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 254–265.

[21] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: Fine grained traffic engineering for data centers," in *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*. ACM, 2011, p. 8.

[22] O. N. Foundation, "Openflow switch specification 1.4.0," 2013.

[23] M. Alhamad, T. Dillon, and E. Chang, "Conceptual sla framework for cloud computing," in *Digital Ecosystems and Technologies (DEST), 2010 4th IEEE International Conference on*. IEEE, 2010, pp. 606–610.

[24] L. Wu, S. K. Garg, and R. Buyya, "Sla-based resource allocation for software as a service provider (saas) in cloud computing environments," in *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*. IEEE, 2011, pp. 195–204.

[25] A. Schrijver, *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[26] Project floodlight. [Online]. Available: http://www.projectfloodlight.org/floodlight/

[27] M. Berkelaar, J. Dirks, K. Eikland, P. Notebaert, and J. Ebert, "lpsolve: A mixed integer linear programming (milp) solver," *URL http://sourceforge. net/projects/lpsolve*, 2007.

[28] Open vswitch. [Online]. Available: http://openvswitch.org/

[29] Lbnl/icsi enterprise tracing project. [Online]. Available: http://www.icir.org/enterprise-tracing

[30] Passive real-time asset detection system. [Online]. Available: http://prads.projects.linpro.no.

[31] V. Paxson, S. Campbell, J. Lee *et al.*, "Bro intrusion detection system," Lawrence Berkeley National Laboratory, Tech. Rep., 2006.

[32] netfilter/iptables project. [Online]. Available: http://www.netfilter.org/

[33] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless network functions: Breaking the tight coupling of state and processing," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), Boston, MA*, 2017, pp. 97–112.