MicroNF: An Efficient Framework for Enabling Modularized Service Chains in NFV

Zili Meng, Jun Bi, *Senior Member, IEEE*, Haiping Wang, Chen Sun, and Hongxin Hu, *Member, IEEE*

***Abstract*— The modularization of service function chains (SFCs) in network function virtualization (NFV) could introduce significant performance overhead and resource efficiency degradation due to introducing frequent packet transfer and consuming much more hardware resources. In response, we exploit the *reusability*, *lightweightness*, and *individual scalability* features of *elements* in modularized SFCs (MSFCs) and propose MicroNF, an efficient framework for MSFC in NFV. MicroNF addresses the performance overhead and resource efficiency problems in three ways. First, MicroNF graph constructor reuses the processing results of elements from different NFs and reconstructs the MSFC after modularization to shorten the chain latency. Second, optimized placer pays attention to the problem of which elements to consolidate and provides a performance-aware placement algorithm to place MSFCs compactly and optimize the global packet transfer cost. Third, MicroNF individual scaler innovatively introduces a *push-aside scaling up* strategy to avoid degrading performance and taking up new CPU cores. To support MSFC reusing and consolidation, MicroNF also designs a high-performance infrastructure to efficiently forwarding packets with consistency ensured and to automatically scheduling elements with fairness ensured when the elements are consolidated on the CPU core. Our evaluation results show that MicroNF achieves significant performance improvement and efficient resource utilization on several metrics.**

***Index Terms*— NFV, service function chain, modularization.**

I. Introduction

NETWORK Function Virtualization (NFV) [2] was recently introduced by replacing traditional hardware-based dedicated middleboxes with virtualized Network Functions (vNFs). Compared to the legacy network, NFV brings benefits of easy development, high elasticity, and dynamic management. Meanwhile, in scenarios such as datacenters or mobile networks, network operators often require traffic to pass through multiple vNFs in a particular sequence (e.g. Firewall⇒NAT⇒Load Balancer), which is commonly referred to as a Service Function Chain (SFC) [3].

To fasten the development of vNFs, many recent research efforts [4]–[7] proposed to break monolithic Network Functions (NFs) into processing *elements*, which could form a Modularized Service Function Chain (MSFC).[1] For example, an Intrusion Detection System (IDS) can be broken into a Packet Parser element and a Signature Detector element [8]. In this way, new vNFs could be built based on a library of elements, which could significantly reduce human development hours and also bring benefits on runtime management [6].

However, introducing modularization to NFV brings two major drawbacks. First, in NFV networks, each vNF is usually deployed in the form of Virtual Machine (VM) with separated CPU cores and isolated memory resource [9]. When traversing a MSFC, packtes have to be queued and transferred between VMs, which could introduce communication latency [10]. As MSFCs require more times of packet transmission between elements than their corresponding SFCs, their performance would be further degraded. Second, due to modularization, to deploy an MSFC, we need to consume much more (possible $2\times$ or more) hardware resources to accommodate all processing elements compared to a SFC, which compromises resource efficiency.

Some research efforts have been devoted to addressing the problems above. OpenBox [6] improved the performance of MSFCs by merging and eliminating redundant modules and shortening processing paths of packets. However, OpenBox's tree-based reusing algorithm simply copied modules to different branches of the processing tree and multiplicatively increased the number of modules.[2] NFVnice [11] addressed the resource efficiency problem by consolidating several NFs onto a CPU core with containers. However, it was designed at NF-level and ignorant of the new problems of modularization such as frequent inter-VM packet transfer. Also, it did not consider the placement problem of *which elements to consolidate*,

Manuscript received January 28, 2019; revised June 27, 2019; accepted June 28, 2019. Date of publication July 5, 2019; date of current version August 6, 2019. 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 61625203, Grant 61832013, and Grant 61872426. This paper was presented the IEEE International Conference on Communications, Kansas City, MO, USA, May 22, 2018. *(Corresponding author: Chen Sun.)*

Z. Meng and H. Wang are with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China, and also with the Beijing National Research Center for Information Science and Technology, Beijing 100084, China (e-mail: zilim@ieee.org; whp18@mails.tsinghua.edu.cn).

J. Bi, deceased, was with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China, and also with the Beijing National Research Center for Information Science and Technology, Beijing 100084, China.

C. Sun is 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).

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.2019.2927069

[1] Sometimes MSFC are referred to as modularized service graph [6] as there may exist branches between elements. In this paper, we use MSFC and modularized service graph interchangeably.

[2] For graph $\mathcal{G}_1$ and $\mathcal{G}_2$ with branches B_1 and B_2, the total branches in the reused graph will be B_1 B_2 and the number of modules on the branches will accordingly multiplicatively increase.

which is also significant to improve performance and resource efficiency. Inappropriate consolidation may worsen performance by transferring packets repeatedly.

At the same time, a closer look into the modularization technique reveals some features of modularization that could benefit both the performance and resource efficiency of MSFCs. Modularization introduces processing elements that are *lightweight*, *individually scalable* and *reusable* [12], [13]. Therefore, if two NFs have elements with the same functionality (*e.g.* both Firewall and IDS have packet header classifier), we could reuse the elements by combining their inside processing logics and deliver the processing results for different purposes with one elements. Meanwhile, even if elements are different and unreuseable, we could consolidate them on the same VM to reduce hardware resource consumption and improve resource efficiency. By considering which elements to consolidate, performance can be improved by reducing inter-VM packet transfer inside MSFC. Further, in the situation where an element is overloaded during runtime, we can only scale out the overloaded lightweight element itself individually instead of its corresponding monolithic NF, which could significantly reduce the scaling cost [14]. The scaled out replica can also be consolidated onto an already working VM without consuming an extra CPU core to further save resource.

Therefore, based on the above observations, we propose MicroNF, an efficient framework to improve the performance and resource utilization efficiency for MSFC in NFV by enabling element reusing and consolidation. To the best of our knowledge, MicroNF is the first framework to enable the *optimized consolidation and reusing strategies at the same time* for MSFC in NFV. The key idea of MicroNF is to reduce inter-element packet transfer at the orchestration level and fully utilize the processing power of CPU. Specifically, the design goal of MicroNF is to reuse elements from different NFs if they are of the same functionality, or otherwise optimize the element consolidation solutions. However, we encounter three main challenges in our design:

- For MSFC reusing, reusing all elements with the same functionality will violate the inter-element dependency. Thus we are challenged to analyze the inter-element dependencies and determine which elements are reusable.
- For MSFC placement, to optimize the performance of MSFCs, we are challenged to carefully analyze the cost of inter-VM packet transfer via a virtual switch (vSwitch) [15]. Moreover, we are challenged to design a performance-aware placement algorithm to consolidate appropriate elements together.
- For MSFC elasticity control, careless placing the scaled out replica may introduce additional packet transfer between VMs and frequent state synchronization among different replicas of the element, which may degrade the performance significantly (possibly up to tens of *ms* [14]). We are challenged to avoid performance degradation.

To address the above challenges, we design the MicroNF framework for consistent element reusing, performance-aware consolidation placement, and elasticity control. Specifically, to enable and optimize element reusing, we propose two-step dependency analysis mechanism and graph reconstruction mechanism. To optimize the MSFC consolidation placement solution, we carefully analyze and model the inter-VM transfer cost and propose a 0-1 quadratic programming-based optimal placement solution. To enable MSFC elasticity control, MicroNF proposes an innovative *push-aside element scaling up* strategy, where the MicroNF controller will push the *border* elements aside to release resources for *overloaded* elements. Finally, we design MicroNF infrastructure to support consistent packet forwarding with element that schedules CPU resources based on the processing speed of each element. We present the workflow of MicroNF in detail in Section III.

In this paper, we make the following contributions:

- We introduce the problem of which elements to consolidate and model the performance cost of inter-VM packet transfer. We design a *Graph Constructor* to enable element reusing with consistency ensured and an *Optimized Placer* in MicroNF controller to achieve optimal performance of MSFCs using 0-1 quadratic programming.
- We design an *Individual Scaler* in MicroNF controller for individual scaling of elements. We propose an innovative push-aside scaling up strategy as well as a greedy scaling out method to alleviate the hot spot with little performance and resource overhead.
- We introduce the MicroNF infrastructure and implement MicroNF on our testbed (Section VII). Evaluation results of MicroNF framework show that MicroNF could improve both performance and resource efficiency.

A. Roadmap

The related research efforts and the contributions of MicroNF over them are introduced in Section II. We present the overview of MicroNF framework and relationships between different components in Section III. We then respectively introduce in Section IV, V, VI and VII. The implementation details and evaluation results are presented in Section VIII. Finally, we discuss several future work in Section IX and conclude the paper in Section X.

II. Related Work

In this section, we summarize some related work and compare them with MicroNF.

A. Modularization

Click [16] proposed the idea of modularization and applies it to routers. Recently, Slick [5] and OpenBox [6] were proposed to detailedly discuss modularized NFs and decouple control plane and data plane of modularized NFs for easy management. Besides, OpenBox focused on reusing elements to shorten the processing path length. However, above works mainly focus on orchestration-level module management and are orthogonal to our optimizations on performance-aware placement and dynamically scaling.

B. Consolidation

CoMb [7] designed a detailed mechanism to consolidate middleboxes together to reduce provisioning cost. Furthermore, Flurries [9] and NFVnice [11] were proposed to share

CPU cores among different NFs with the technique of Docker Container [17]. By modifying Linux scheduling methods, they achieved almost no loss in NF sharing. However, they operated on monolithic NF level and did not consider the problem of which elements (NFs) to consolidate. Nonetheless, their development details and infrastructure designs could complement our work as the low-level implementation.

C. Placement

Researches on NF placement in NFV, such as [18]–[20], mainly focus on the trade-off between traffic load, service quality, forwarding latencies, and link capacities. However, they are mainly designed for monolithic SFCs spreading over multiple physical servers, which are different from the application scenario of MicroNF. Slick [5] considered placement at element-level. However, all of the work above addressed how to place *middleboxes (vNFs)* onto different *servers* considering complicated and limited physical links. In contrast, MicroNF pays attention to the placement problem of *how to consolidate elements* onto different *VMs* to minimize the inter-VM transmission. We also implement a modified version of Slick in Section VIII-C, which is outperformed by MicroNF.

D. Virtualization Infrastructure

Many researches also focus on designing high-performance infrastructure for NFV. ClickOS [21] implemented different high-performance and lightweight virtual machines for network functions. NetVM [10] and OpenNetVM [22] improved the packet delivery between NFs when they are placed together. E2 [23] proposed a scalable scheduling mechanism to further support network dynamics. On the contrary, MicroNF focuses on the placement and scaling *strategies* rather than *mechanisms* for service chains. MicroNF are orthogonal to the work above and could work with different underlying virtualization infrastructures.

Finally, compared to the earlier version of this paper [1], we have made substantive enhancements in this manuscript. Beyond element consolidation, we make a key observation that SFC latency can be further reduced by *reusing* elements with the same functionality from different NF. We then design the MicroNF *Graph Constructor* to reconstruct the element graph before optimizing consolidation. An element reusing mechanism and a dependency analysis method to optimize the element reusing solutions are proposed in the MicroNF *Graph Constructor*. The element reusing mechanism could complement the consolidation strategies proposed in [1] at different levels. After that, we re-design the MicroNF infrastructure to support both consistent inter-element forwarding between different machines and efficient element scheduling on the same machine. Finally, we have updated the evaluation to demonstrate that the element reusing mechanism could achieve promising performance improvement.

III. MicroNF Design Overview

To address the above challenges, we design the MicroNF framework to enable modularized service chains in NFV. Components and workflows of the MicroNF framework are shown in Fig 1. MSFC deployment workflow (dark orange arrows in Fig. 1) illustrates how to deploy a new MSFC in MicroNF at the initial stage. MSFC control workflow (blue arrows in Fig. 1) depicts how to dynamically scale and schedule elements against traffic fluctuations during runtime. We respectively introduce two workflows in Fig. 1 as below:

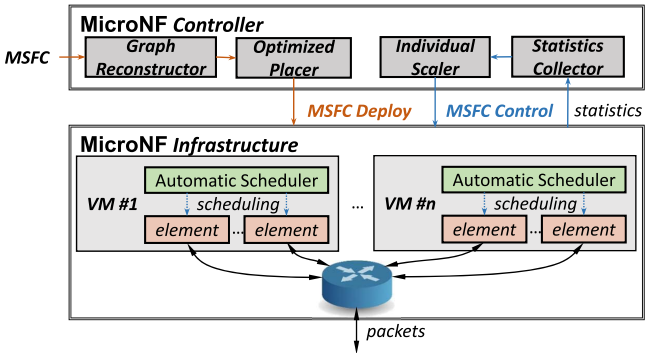


Fig. 1. MicroNF framework overview.

For MSFC deployment workflow, network operators first define the MSFC by describing interconnection relationships between elements [6]. The MSFC will first be processed by *Graph Reconstructor*, where inter-element dependency will be identified and redundant elements will be reused. *Optimized Placer* will then calculate the optimal placement solutions of the reorganized MSFC to optimize the inter-element latency considering the transmission over virtual switches. We respectively introduce *Graph Reconstructor* and *Optimized Placer* in Section IV and Section V.

For MSFC runtime control workflow, *Statistics Collector* continuously collects element statistics (*e.g.* processing rate). *Individual Scaler* then makes decisions on whether and how to scale and migrate elements with minimal inter-element latency. The *Infrastructure* finally executes the control actions from *Individual Scaler* and also ensures packet processing consistency. We respectively introduce *Individual Scaler* and *Infrastructure* in Section VI and Section VII.

Note that MicroNF Controller is designed to optimize the modularization of SFCs on different devices, specifically the latency among different element instances over virtual switches such as Open vSwitch [15]. For simplicity and better illustrations, we discuss the transmissions between VMs in this paper. MicroNF could also be applied over other virtualization techniques supporting virtual switches (*e.g.* containers) since the transmissions over virtual switch between containers are similar to that between VMs [15].

IV. Element Reusing-Aware Graph Reconstruction

In this section, we first introduce the categories of elements we consider in MicroNF in Section IV-A. We then respectively introduce how to reuse elements and reconstruct the element graph with both intra-element consistency (the reused element can produce correct results) and inter-element consistency (the reconstructed element graph are correct) ensured in Section IV-B and IV-C.

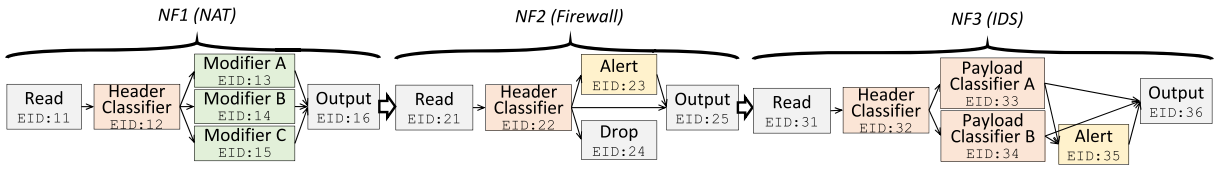


Fig. 2. Element graph of service function chain NAT⇒FW⇒IDS.

TABLE I
SEVERAL TYPES OF MicroNF ELEMENTS

Element Type		Feature	Example
Brancher	Header Classifier	• Attach the following processing path to packets. • Processing logic can be abstracted as a MAT.	• Flow Classifier. • Protocol Analyzer.
	Payload Classifier	• Attach the following processing path to packets. • Processing logic *cannot* be abstracted as a MAT.	• Stateful Payload Inspector.
Nonbrancher	Monitor	• Stateful flow monitoring and statistics collecting. • Modify neither packets nor flow features.	• Flow Counter. • Logger.
	Modifier	• Write on certain packet fields based on internal logics.	• IP Address Modifier
	Reorganizer	• Add or remove specific packet fields.	• Layer 3 Header Encapsulator
	Shaper	• Modify flow features but not packets.	• Leaky Bucket Shaper [24]

A. Element Unit Design

Instead of simply classifying processing units based on their functions [6], we *design* different element units with the purpose of ensuring intra-element consistency during reusing and dependency analysis.

An element is a piece of high-coherent processing logic broken from its original monolithic NF. For example, a Firewall can be split into a Classifier element, an Alert element and a Drop element. Operators can further decompose a SFC into a *element graph*. Fig. 2 is an example of a MSFC that is modularized from a datacenter service chain NAT⇒FW⇒IDS. Monolithic vNFs are modularized into elements as presented in Fig. 2.

To enable element reusing, we summarize common processing units of NFs and design different elements in Table I. We first classify elements into branchers and nonbranchers according to whether their downstream have multiple elements or not.[3]

1) Brancher: Branchers represent elements classifying packets and specifying the processing paths for them, such as the Header Classifier in Firewall. Since most NFs have branching logics, brancher is a significant type of element to consider during element reusing. Thus we further categorize branchers into Header Classifiers and Payload Classifiers according to their inspection levels.

Header Classifier is the brancher that classifies packets solely based on packet headers. Since the structure of packet headers usually falls into several fixed types [25], the internal processing logic of packet headers can be abstracted as a Match-Action-Table (MAT). As shown in Fig. 3, modifications on packets and internal states can be abstracted as different *actions* following respective matching rules. We use *destination Element ID* (`dEID`) to indicate the following processing path of packet in the MAT in Fig. 3.

[3]Terminal elements such as Drop are just intuitive descriptions of actions. They can be executed together with other elements.

IP Address Classifier (EID=22)

Matching rules	Actions
`SrcIP==10.0.0.*`	`dEID+=25;break;`
`SrcIP==10.0.1.*&count<10`	`dEID+=24;count++;break;`
`SrcIP==10.0.1.*&count==10`	`dEID+=23,25;count=0;break;`

IP Address Classifier (EID=32)

Matching rules	Actions
`SrcIP==10.0.0.*`	`dEID+=33;break;`
`SrcIP==10.0.1.*&flag`	`dEID+=33;flag=0;break;`
`SrcIP==10.0.1.*&(!flag)`	`dEID+=34;flag=1;break;`

Fig. 3. Two examples of header classifiers in Fig. 2.

Payload classifier is the brancher that further inspects the payload of packets to make the decisions for those packets. Since Payload Classifiers need to analyze the payload of packets, parsing packet headers only cannot fully represent their behaviors. Thus they cannot be abstracted as MATs and are considered separately in MicroNF.

2) Nonbrancher: In contrast to branchers, nonbrancher represents the element that processes packets only and does not have influence on the future processing path of packets. According to how they process packets, we further categorize them into monitors, modifiers, reorganizers and shapers. We introduce them in detail respectively as below:

Monitor is the element that only reads packets and does not perform any action on packets. Packets before and after the process of monitors are strictly the same.

Modifier modifies certain packet header fields. In our design, read and write (R/W) operations are separated to enable further element reusing among different elements, which will be introduced in Section IV-B.

Reorganizer changes the packet structure. For example, the encapsulator extracted from Virtual Private Network (VPN) constructs a new packet header out of the current packet.

Shaper modifies the flow-level features. They perform no actions on packets directly. Instead, they employs buffer to shape the statistical features such as traffic distribution [24].

Mbit	Matching rules	Actions
11	SrcIP==10.0.0.*	dEID+=33;**break**;
11	SrcIP==10.0.1.*&count_22<10	dEID+=24;count_22++;**break**;
10	SrcIP==10.0.1.*&count_22==10	dEID+=23,32;count_22=0;**goto** 01;
01	SrcIP==10.0.1.*&flag_32	dEID+=33;flag_32=0;**break**;
01	SrcIP==10.0.1.*&(!flag_32)	dEID+=34;flag_32=1;**break**;

Fig. 4. Reusing two header classifiers in Fig. 3

Compared to other analysis on NF dependency [26], MicroNF provides a more thorough categorization (*e.g.* dealing with brancher and nonbrancher respectively) specialized for modularized elements. Moreover, or user-defined customized elements, we can analyze their actions on packets and flow features and classify them into one of the types above according to their actions. For more complicated elements (*e.g.* modifying packet payloads), we will not reuse them to conservatively ensure packet processing consistency.

B. Element Reusing Mechanisms

When reusing elements, we aim at achieving *intra-element consistency*, *i.e.* the reused elements must handle packets and maintain states the same as the unreused individual elements. Based on the above element units, we intend to ensure *consistent processing logic* when reusing the same type of elements originally belonging to different vNFs together similar to OpenBox [6]. We respectively design reusing mechanisms for different types of elements.

Header Classifier: The key of reusing Header Classifiers is reusing their respective MATs. When designing the structure of reused header classifiers, we have three goals:

① Matching rules should be visited in the same sequence as individual classifiers to avoid inconsistency caused by violating inter-element dependencies.
② Packet must visit the entries that it should match from all elements only once.
③ Packets should match all entries they should match as soon as possible.

To achieve the goals above, MicroNF introduces *Match Bit* (`MBit`) for each entry to indicate where it originates from. When reusing two MAT Classifiers, MicroNF reuses the actions of entries with the same matching rules. Reused entries are labeled with `MBit` 11 and entries from element #1 are with 10. In response to goal ①, `MBit` with discontinuous 1s, such as 101, will never appear during reusing because it violates the sequence of #1⇒#2⇒#3. In response to goal ②, we elaborately design respective actions for entries after packet processing. When the lowest bit of `MBit` of the entry is 1, which indicates that the packet has gone through all elements (goal ①), the process of looking up MAT terminates. The final action in this case will be `break`, as shown in Fig. 4. If the lowest bit of MBit is 0, which indicates that there is at least one element the packet has not gone through, the packet will continue looking up the MAT The final action will be `goto` the first entry reused from all elements that the packet has not gone through yet. In Fig. 4, the final action of entries with `MBit==10` is `goto 01`. To achieve goal ③, we reconstruct the MAT by `MBit` in decreasing order. Packets will match the reused entries first, which will shorten the looking up time on average. An example of the sequence of looking up forwarding tables when reusing 3 elements is presented in Fig. 5.

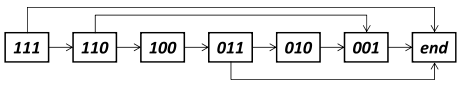


Fig. 5. `MBit` forwarding mechanisms for 3 bits.

Note that the total number of entries in the reconstructed MAT is always decreased because we reuse the repeated entries but do not create new entries.[4] Sometimes packets may be dropped between two classifiers (e.g. Drop in Fig. 2), then it should not match the downstream classifiers. To ensure robustness, we set the lower bits in the related `MBit` to 1.

Monitor and Modifier: MicroNF element unit separates the R/W operations of monitors and modifiers from their processing logics. To save the read-write time and queuing time of packets among elements, MicroNF inputs packet once, sequentially executes respective processing logics and transfers the pointer of packet among them after reused. For modifiers, MicroNF further reuses write actions of different modifiers and writes back all modifications together when packet processing finishes. If modifications are conflicting, result from the last modifier will be kept.

Shaper and Reorganizer: Similar to modifiers, if there are several shapers in the processing graph, only the last shaping rule will take effect. When reusing shapers, we can directly take the result from the last shaper. However, reorganizers modify the packet structure. If network operators reuse two reorganizers, other elements between those two reorganizers will face significant problems since packets cannot be correctly parsed. Thus MicroNF does not allow reorganizers to reuse.

Payload Classifier: For Payload Classifiers, special analysis on the internal processing logic and state space should be taken, which is out of our scope. We can easily adopt related research on analysis of complicated firewall state spaces [27] into MicroNF after modularized.

C. Dependency Analyzing

Elements in the same chain may have read-write dependencies on each other. Because element reusing changes the processing sequence inside the element graph, carelessly reusing may lead to consistency problems. To ensure *inter-element consistency*, we must be aware of dependencies between elements when reusing element graphs. We propose a *result consistency* principle: *A group of elements are reusable, if and only if the reused and reconstructed graph results in the same processed packets and internal states as the original one.*

Before introducing the analysis mechanisms in detail, we introduce the design of Element Interconnection Specification (EIS) to describe an element graph. For each element, EIS has three required parameters including:

EID: is the ID of element to distinguish the elements.

[4]In contrast, naively reusing classifiers [6] by doing Cartesian product on matching rules will lead to explosively increased number of rules.

TABLE II

CONSISTENCY CHECKING TABLE. *Green Blocks Denote Reusable, Orange Blocks Denote Further Checking, Gray Blocks Denote Not Reusable*

Mid \ End	Monitor & Classifier	Modifier	Reorganizer	Shaper
Monitor & Classifier				
Modifier				
Reorganizer				
Shaper				

Downstream: describes the interconnection relationship of elements inside a graph. For terminal elements, such as Drop or Output, `Downstream` is left to empty.

Reusability: indicates whether this element can be reused. If operators want to isolate the element, they can set `Reusability` to `false` to prevent it from potential reusing. Default value of `Reusability` is `true`.

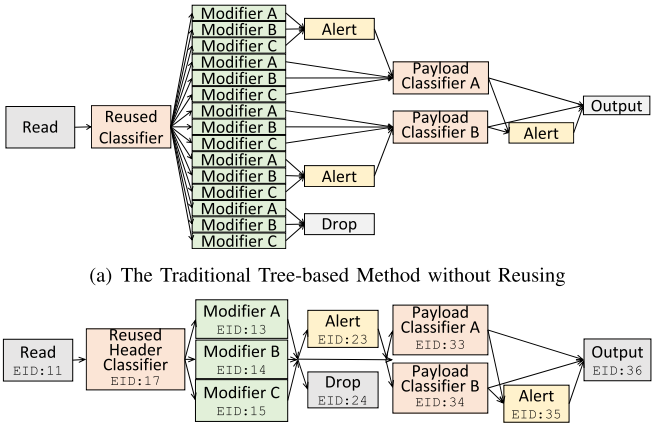
With EIS, we propose two mechanisms to analyze the inter-element dependency. Type-by-type analysis is relatively concise and suitable for most cases, while field-by-field analysis is designed for those tough situations that type-by-type analysis cannot deal with.

1) Type-by-Type Analysis: Type-by-type analysis checks the types of elements and directly determines the reuseability. Based on the result consistency principle, we summarize the reusable and unreusable situations in Table II. *End* represents the element to reuse at both ends, while *Mid* refers to the element between those two end elements. Green blocks represent the reusable situations. Orange blocks represent situations that we cannot justify merely by element types. Those situations need further field by field checking. Gray blocks represent unreusable situations. For example, for an MSFC $ms1 \Rightarrow ms2 \Rightarrow ms3$, where $ms1$ and $ms3$ are the same type of element, if all of $ms1, ms2$ and $ms3$ are monitors and do not modify packets, $ms1$ and $ms3$ are certainly interchangeable thus reusable. But if $ms2$ modifies the packet, we need to further check whether the modified fields will be read by $ms3$, otherwise the result consistency principle may be violated.

For cases with more than one mid elements, which is more common in the real processing graphs, MicroNF checks each one of them and takes the conservative result, i.e. two elements are reusable *if and only if* all mid elements are under reusable situations.

According to [26], 75% of currently implemented NFs are read-only, such as Firewall, Load Balancer, etc, which will result in classifiers or monitors after modularized. Thus the most frequently hit block is the top-left green (reusable) block in Table II. NFs with definitely unreusable elements such as VPN are less than 7%. MicroNF is suitable for most cases in the real world.

2) Field-by-Field Analysis: For situations that cannot be justified by Table II, we propose field-by-field analysis to further check read-write dependencies and avoid inconsistency. We first define *read field* and *write field* of element, which are the fields on *packets* that element will read or modify. For example, the read field and write field of an IP Address Classifier are $srcIP \cup dstIP$ and $\emptyset$ because it classifies packets by $srcIP$ and $dstIP$ but does not modify packets.



(a) The Traditional Tree-based Method without Reusing

(b) Reusing Elements on Different Branches with MicroNF. The convergence and divergence of arrows are simplified illustrations for multiplicative paths.

Fig. 6. Reused element graph of Fig. 2.

We should ensure that fields read by end elements should not be modified by mid elements and vice versa. We denote the read field and write field of element i as R_i and W_i. Thus the checking result, denoted as M, is:

$$M = \left[\left(\bigcup_{i \in Mid} R_i\right) \cap W_{End}\right] \cap \left[\left(\bigcup_{i \in Mid} W_i\right) \cap R_{End}\right] \quad (1)$$

where *Mid* refers to the element between those two end elements, R_{End} and W_{End} are the read field and write field of the end element. If both constraints are satisfied, M will be empty and field-by-field analysis returns `reusable`. Otherwise, the read fields and write fields of elements have overlaps and the elements cannot be reused to ensure result consistency.

3) Reusing Multiple Elements: When reusing multiple elements, we can simplify the dependency analysis by the property of inter-element dependencies. We denote the reuseability of the same type of element $ms1$ and $ms2$ in graph $\mathcal{G}$ as $\langle ms1, ms2\rangle_{\mathcal{G}}$. $ms1 \succ ms2$ means $ms1$ is in the upstream of $ms2$. For $ms1, ms2$ and $ms3$, if $ms1 \succ ms2 \succ ms3$, we have:

$$\langle ms1, ms3\rangle_{\mathcal{G}} = \langle ms1, ms2\rangle_{\mathcal{G}} \;\&\&\; \langle ms2, ms3\rangle_{\mathcal{G}} \quad (2)$$

Eq. 2 holds because the read field $R_{ms1} = R_{ms2} = R_{ms3}$ and write field $W_{ms1} = W_{ms2} = W_{ms3}$. If elements between $ms1$ and $ms2$ do not interfere the reusing of $ms1$ and $ms2$, they will not interfere the reusing of $ms1$ and $ms3$ either. Thus we can reuse $ms1, ms2$ and $ms3$ together. With Eq. 2, MicroNF can easily reuse three or more elements to improve performance. Also, we do not need to go through every pair of elements in the processing graph but just check the reuseability of all 'adjacent' elements.

After finally reconstructing the element graph, the reused element graph is shown in Fig. 6(b). Compared to the traditional tree-based orchestration in Fig. 6(a), MicroNF has two main advantages. First, by reusing elements on different branches, MicroNF reduces the number of element significantly and improves resource efficiency. Second, for stateful elements, MicroNF avoids complicated state synchronizations

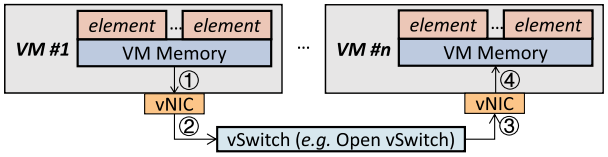
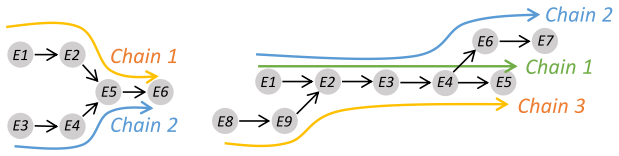


Fig. 7. Inter-VM packet transfer deep dive.



(a) Topology 1 (b) Topology 2

Fig. 8. Two Examples of Processing Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$.

among several replicas on different branches, which are expensive and error-prone [14].

V. Performance-Aware MSFC Placement

In this section, we present the MicroNF elements placement algorithm inside Optimized Scaler of MicroNF Controller for the initial deployment of an MSFC. We have the following goal in mind in our design: *We prefer to consolidate adjacent elements in an MSFC on the same VM and place the MSFC compactly to reduce inter-VM packet transfer cost.*

In the following, we first analyze the one-hop inter-VM packet transfer cost due to vSwitch-based forwarding. We then find the relationship between CPU utilization and processing speed for an element. These two analyses serve as the fundamentals of placement algorithm of MSFC, which usually contains multiple hops and multiple elements.

A. Packet Transfer Cost Analysis

In NFV implementation, elements are usually implemented as VMs separately with dedicated CPU cores [9]. To simplify the resource constraint analysis, we assume that each VM is implemented on one CPU core, which could easily be extended to situations where a VM is allocated with multiple CPU cores (Section IX). When packets are consolidated on the same VM with Docker Container [17], intra-VM packet transferring is simple. With shared memory technique provided by Docker, we can directly deliver the pointer on memory of packet from one element to another with negligible latency (about 3 μs under our implementation). However, when packets are transferred between VMs, they must go through four steps, as shown in Fig. 7. First, packets are copied from memory to the virtual NIC (vNIC) on the source VM (Step ①). Next vNIC transfers packets to vSwitch (Step ②). And then packets are delivered reversely from vSwitch to the vNIC of destination VM (Step ③) and finally from vNIC to memory (Step ④). The total transfer delay (about 1 *ms* in our evaluation) degrades the performance of MSFC significantly.

We use *Delayed Bytes (DB)* to represent the packet transfer cost. Theoretically, DB is constrained by the minimum of element throughput, memory copy rate (Step ① and ④), and packet transmission rate (Step ② and ③). However, memory copy rate and packet transmission rate ($\sim$10 *GB/s* according to [28]) are much greater than the SFC throughput (99% are $<$1 *GB/s* in datacenters [29]). Thus DB is directly constrained by the throughput between elements. We denote the throughput as Θ and the additional four-step transfer delay as t_d. Thus

$$DB = \Theta \cdot t_d \tag{3}$$

In the placement analysis, we will use the total sum of DB as our optimization target of performance overhead.

B. Resource Analysis

Before globally optimizing the total sum of DB, we need to analyze and find the constraints on CPU resource utilization. For a certain element, as the throughput increases, it will consume more CPU resources to process. Thus for each type of element i, we can measure a respective one-to-one mapping function between CPU utilization r and processing speed v:

$$r_i = \phi_i(v_i) \text{ and } v_i = \phi_i^{-1}(r_i) \tag{4}$$

The mapping function $\phi(\cdot)$ could be profiled for each type of element in advance to enable the optimization below. MicroNF provides an automatic tool to efficiently measure the mapping function, the details of which will be introduced in Section VIII-E.

Docker-based consolidation technique is lightweight and takes few resources [9]. Thus, we can directly add up respective CPU utilizations of elements to estimate the total CPU utilization. Also note that Eq. 4 is an upper bound estimation for CPU utilization given throughput. When several elements are consolidated together, the alternate scheduling mechanism by reusing the idle time caused by interrupts [28] can enable a higher total throughput.

C. MSFC Placement Algorithm

We first abstract the packet processing in MSFCs as a directed acyclic processing graph, denoted as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Each node $k \in \mathcal{V}$ represents an element and each edge in $\mathcal{E}$ represents a hop between elements in an MSFC. $k \in \mathcal{I}$ represents the VMs, i.e. CPU cores. Service chains, denoted as $\mathcal{C}$, are defined by tenants. Fig. 8 shows two examples of processing graph. There are two chains and six elements in Fig. 8(a). Chain 1 is E1⇒E2⇒E5⇒E6 and Chain 2 is E3⇒E4⇒E5⇒E6. To consolidate compactly, we assume that the processing speed of each elements on the same chain matches the throughput of the entire chain at initial placement. We denote the throughput of chain j as Θ_j. Conservatively, network administrators can estimate Θ_j with its required bandwidth according to Service Level Agreement [18].

$\alpha_i^j \in \{0, 1\}$ indicates whether element i is on chain j. π_i^j represents the upstream element of element i on chain j, which can be realized with a doubly linked list. To ensure robustness, when i is the first element on chain j, we set $\pi_i^j = i$. When $\alpha_i^j = 0$, we set $\pi_i^j = 0$.

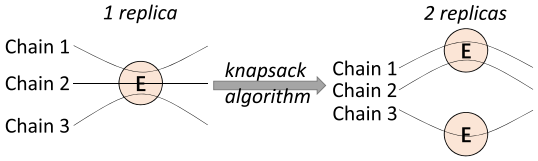


Fig. 9. Knapsack algorithm for one element shared by multiple chains.

MicroNF applies a 0-1 Integer Programming (0-1 IP) algorithm to minimize the inter-VM overhead. $x_{i,k}$ is a binary indicator of whether placing element i onto VM k. For chain j, we analyze the performance overhead between each element i and its upstream element π_i^j on it. From Eq. 3, the hop from π_i^j to i will incur an inter-VM cost of DB_j if and only if they are not placed on the same VM, i.e.

$$x_{i,k}x_{\pi_i^j,k} = 0, \quad \forall k \in \mathcal{I}$$

Thus we can use $\left(1 - \sum_{k\in\mathcal{I}} x_{i,k}x_{\pi_i^j,k}\right) \in \{0,1\}$ to indicate whether element π_i^j and i are consolidated together. Then we add up DB of all inter-VM hops in chain j and further add up DB of different chains as our objective function. MicroNF aims at minimizing the total inter-VM cost to improve performance:

$$\min \sum_{j\in\mathcal{C}}\sum_{i\in\mathcal{V}} \alpha_i^j DB_j \left(1 - \sum_{k\in\mathcal{I}} x_{i,k}x_{\pi_i^j,k}\right) \tag{5}$$

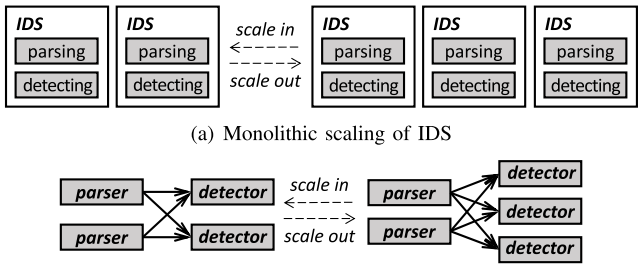
Meanwhile, the following constraints should be satisfied:
(1) $x_{i,k} \in \{0,1\},\ \forall i \in \mathcal{V},\ k \in \mathcal{I}$
//An element is either consolidated on VM k or not.
(2) $\sum_{k\in\mathcal{I}} x_{i,k} = 1,\ \forall i \in \mathcal{V}$
//An element can only be placed onto one VM.
(3) $\sum_{i\in\mathcal{V}} \left[x_{i,k} \cdot \phi_i \left(\sum_{j\in\mathcal{C}} \alpha_i^j \Theta_j\right)\right] \leqslant 1,\ \forall k \in \mathcal{I}$
//Each CPU core cannot be overloaded at initial placement.

Moreover, if the estimated throughput of element i_0 is too high to be placed on one CPU core, i.e.

$$\exists i_0 \in \mathcal{V},\ s.t. \sum_{j\in\mathcal{C}} \alpha_{i_0}^j \Theta_j > \phi_{i_0}^{-1}(1) \tag{6}$$

Constraint (2) and (3) may conflict. This is due to the incorrect orchestration between flows and elements. Actually it rarely happens in the real world and never happens in our evaluation. In this situation, scaling out is needed. For overloaded element i_0, if there is only one chain j_0 containing it, MicroNF scales out $\lceil \frac{\Theta_{j_0}}{\phi_{i_0}^{-1}(1)} \rceil$ replicas and performs load-balancing among them. If there are several chains containing i_0 (*e.g.* E5 in Fig. 8(a)), MicroNF scales it out to multiple replicas. Since splitting traffic inside a service chain might introduce additional management cost [14], to minimize the management cost, MicroNF reallocates those service chains onto newly created replicas with a knapsack algorithm as presented in Fig. 9. Then MicroNF splits the overlapped chains to different replicas and reconstructs processing graph $\mathcal{G}'$ to ensure that Eq. 6 does not hold for $\forall i \in \mathcal{G}'$.

From the analysis above, we find that Eq. 5 is a 0-1 Quadratic Programming problem and can be solved within limited time and space [30]. By solving the above formulations, we can get the performance-aware optimized placement solution. We evaluate this algorithm in Section VIII-C.



(a) Monolithic scaling of IDS

(b) Individual scaling of modularized IDS

Fig. 10. Individually scalability.

VI. Optimized Individual Scaling

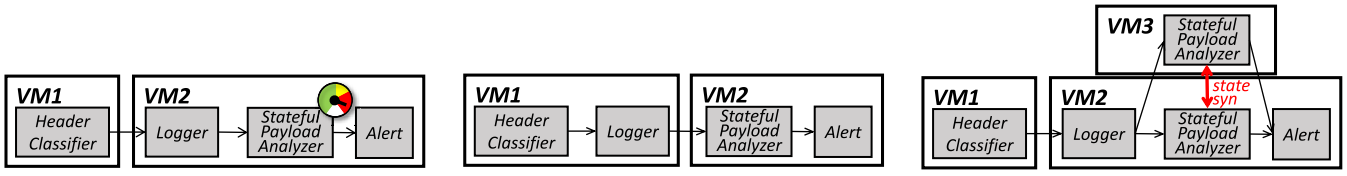
For monolithic NFs, all components must be scaled out at the same time when overloaded, which will take up more resources than needed. Also, continuously synchronizing numerous internal states will introduce significant overhead [14]. After modularization, when traffic increases, only the overloaded elements need to scale out. For example, when the *detector* element of IDS is overloaded, instead of scaling out the whole IDS (Fig. 10(a)), we can only scale out the detector element itself (Fig. 10(b)).

However, deciding where to place the scaled out replica is also important. Careless placement will degrade performance and resource efficiency. In this section, MicroNF Individual Scaler provides two innovative scaling strategy including performance-aware push-aside scaling up, and resource-aware greedy scaling out, to efficiently alleviate the overload situation. Push-aside scaling up can avoid the performance degradation caused by additional inter-VM packet transfer. Greedy scaling out can achieve resource efficiency by placing replicas on existing VMs.

A. Push-Aside Scaling Up

When scaling out, the traditional NF-level scaling method taken by [14], [31] simply starts a new VM with taking a new CPU core and scales out the overloaded element to the new VM. For example, when the Stateful Payload Analyzer in VM2 is overloaded (Fig. 11(a)), traditional method starts VM3 and copies the element to it (Fig. 11(c)). However, this will introduce additional latency overhead due to inter-VM packet transfer. For example, in Fig. 11(c), a part of packets will suffer 3 inter-VM hops to go through the total MSFC (VM1⇒VM2⇒VM3⇒VM2). Also, frequent state synchronization between replicas will also degrade the performance.

However, when an element in an MSFC is overloaded, the VMs that its upstream or downstream elements placed on may be underloaded. Enabled by the lightweight feature of element, we can re-balance the placement of elements on the two VMs. Thus, the key idea of push-aside scaling up is that the overloaded element can *push* its downstream/upstream element *aside* to its downstream/upstream VM and *scale* itself



(a) MSFC before scaling (b) Push-aside scaling up (c) Scaling out with traditional method

Fig. 11. Comparison of push-aside scaling up and traditional scaling out.

up to alleviate the overload. As for Fig. 11(a), we can *migrate* Logger to VM1 and release its CPU resource (Fig. 11(b)). By allocating the newly released resource to Stateful Payload Analyzer and scaling it up, the overload can be alleviated.

Push-aside scaling up has two advantages. First, compared to the traditional method, it does not create new inter-VM hops, thus there is no additional packet transfer cost. Second, it does not create a new replica but allocates more resources to overloaded element. Thus push-aside scaling up does not suffer the state share and synchronization problems [14].

The algorithm includes the following four steps.

Step 1: Check the practicability. If the estimated throughput $\Theta_{i_0}^{exp}$ for element i_0 is so large that the overload on i_0 cannot be alleviated with one CPU core, i.e. $\Theta_{i_0}^{exp} > \phi_{i_0}^{-1}(1)$, push-aside scaling up will not work. This happens only when an extremely large burst comes. In this situation, algorithm terminates and MicroNF goes to greedy scaling out.

Step 2: Find border elements. For element i_0 in chain j_0, MicroNF first finds out its upstream and downstream border elements $up\_border_{i_0}^{j_0}$ and $down\_border_{i_0}^{j_0}$. Upstream border element refers to the element that $up\_border_i^j$ and i are placed in the same VM but $up\_border_i^j$ and $\pi^j_{up\_border_i^j}$ are placed separately. With doubly linked list, MicroNF goes through elements hop by hop to find out border elements and composes them into a set $\mathcal{B}_{i_0}$. If i_0 is contained by several chains, MicroNF checks each chain and composes the results into $\mathcal{B}_{i_0}$.

Step 3: Check whether it can be migrated. After finding out the border elements, MicroNF checks whether they can be migrated to the adjacent VM. Suppose both $b_0 \in \mathcal{B}_{i_0}$ and i_0 are on chain j_0. We denote the adjacent VM of b_0 as $k_{b_0}^{adj}$. If

$$\phi_{b_0}(\Theta_{j_0}) + \sum_{i \in \mathcal{V}} x_{i,k_{b_0}^{adj}} \cdot \phi_i \left(\sum_{j \in \mathcal{C}} \alpha_i^j \Theta_j \right) < 1 \qquad (7)$$

which means there is available resource for b_0 on $k_{b_0}^{adj}$, we can migrate b_0 to $k_{b_0}^{adj}$ and release its resource. Similarly, we can check all $b \in \mathcal{B}_{i_0}$ and its respective k_b^{adj}. If none can be migrated, MicroNF goes to the greedy scaling out strategy. This happens only when all of the adjacent VMs of i_0 do not have enough resource, which in practice rarely happens. If some of them can be migrated, MicroNF composes them into $\mathcal{B}'_{i_0} \subset \mathcal{B}_{i_0}$.

Step 4: Check whether overload can be alleviated. At last, MicroNF checks if migrating all elements in $\mathcal{B}'_{i_0}$ will make enough room for i_0 to scale up to alleviate the overload. Otherwise the migration will be useless. MicroNF calculates the needed resource $r_{i_0}^* = \phi_{i_0}(\Theta_{i_0}^{exp}) - \phi_{i_0}(\Theta_{i_0}^{cur})$, where $\Theta_{i_0}^{cur}$ is the current processing speed of i_0. The CPU utilization of element $b' \in \mathcal{B}'_{i_0}$ satisfies $r_{b'} = \phi_{b'}\left(\sum_{j \in \mathcal{C}} \alpha_{b'}^j \Theta_j\right)$. If $\sum_{b' \in \mathcal{B}'_{i_0}} r_{b'} < r_{i_0}^*$, which means migration cannot release enough resource, the algorithm terminates and MicroNF goes to greedy scaling out. Else, push-aside scaling up can be applied. Also, aware that migrating elements from one VM to another has performance cost and controller overhead [14], MicroNF tries to minimizes the number of elements to migrate. MicroNF finds a subset $\mathcal{B}''_{i_0} \subset \mathcal{B}'_{i_0}$ with minimal number of elements that satisfies $\sum_{b'' \in \mathcal{B}''_{i_0}} r_{b''} \geqslant r^*$.

Moreover, to avoid potential frequently scaling up among elements, network administrators can set a timeout between each time of scaling up. In this way, we can alleviate the overload with minimum elements to migrate.

B. Greedy Scaling Out

If the overloaded element can push none of border elements aside to other VMs, Individual Scaler have to scale it out to somewhere else. In this situation, performance degradation caused by scaling out is unavoidable. Even so, we can still save resource by placing the new replica to an already working VM.

MicroNF decides the VM to place the replica based on a greedy algorithm. First, it calculates the remained resource of each VM and sorts them in increasing order. Next, MicroNF greedily compares it with the needed resource $r_{i_0}^*$. When the remained resource of any VM, i.e. CPU core, is larger than $r_{i_0}^*$, MicroNF places the replica there. If none of the VMs have available CPU resource, MicroNF will call up new VMs and specify new CPU cores, just as the traditional method does.

VII. Infrastructure

In this section, we introduce two designs of MicroNF infrastructure to support our previous designs:

- When reusing elements from different NFs, we need to reconstruct the element graph, as introduced in Section IV. In this case, the forwarding sequences of packets are now totally different from previous service chains: different packets have different forwarding paths. How to ensure the consistency is challenging (Section VII-A).
- For runtime MSFC management, when consolidating multiple elements on the same CPU core, we need to ensure fairness when scheduling CPU resources among

elements. However, traditional approach [11] requires *manual* configuration of element priorities, which is time-consuming and lacks scalability. We are challenged to design an automatic scheduler (Section VII-B).

A. Forwarding Mechanism Design

When forwarding packets among elements, traditional methods [6], [10] either configure the interconnection relationship of ports and output packets to different ports, or forward packets centrally at vSwitch. However, the first method is unsuitable when reusing elements because it can just indicate only next one hop to packets. Thus its tree-based algorithm simply reuses the graph by copying elements onto different branches, as shown in Fig. 6(a). In contrast, element reusing needs multi-hop processing paths indicated by branchers. As for the second method, as the total number of flows inside a network for a period can be extremely large (possibly up to 10k [32]), additionally looking up a table containing tens of thousands of rules will introduce significant performance overhead [33].

MicroNF ensures the consistency of processing sequence by attaching `dMID` to packets to indicate the following processing path, which is referred to as dMID-based forwarding mechanism. The dMID-based forwarding mechanism has three steps: First, when a packet has been processed by a brancher, *`dMID` Attacher* module will attach the following processing path to it by modifying `dMID` fields in *metadata*. Second, when the packet comes to vSwitch, it will be forwarded to one of elements according to the `dMID` attached on that packet. Finally, *`dMID` Updater* module removes the out-of-date `dMID` and the packet will be processed by the element. After processing finishes, it will be outputted to the vSwitch again.

We realize the `dMID` design by utilizing the metadata in `mbuf` structure with Intel DPDK [34]. The 64-bit metadata is divided into three following fields in MicroNF:

FID (14 bits): Flow ID (`FID`) identifies different flows inside the element graph for SFC dynamic management.

dMID (36 bits): `dMID` is an array containing six 6-bit MIDs to indicate the following several hops in its processing path. According to our preliminary attempts to break monolithic NFs, most of NFs can be broken into less than 8 elements. Also, the number of NFs in most SFCs is usually less than 5 [3], [35]. Thus up to 64 elements provided by 6-bit MID is enough inside a graph. Meanwhile, the control range of a brancher is less than the length of SFC. Hence, 6 hops for one brancher can handle most of situations in the real world.

Tags (14 bits): Tags are used to transfer messages between elements. For example, the Alert element may need the result from its upstream brancher to alert messages.

B. Automatic Consolidation Scheduling

When consolidating several elements onto one CPU core, MicroNF uses one Docker [17] for each element. At this time, we need a scheduling algorithm to enable fair resource allocation. However, as we discussed above, traditional rate-proportional scheduling methods [36] and priority-aware scheduling methods [11] are not scalable due to massive manual configurations.

In response, we design a novel scheduling algorithm to match processing speed of elements with its throughput to automatically achieve both fairness and efficiency. Here, we take CPU resource as the allocation variable since CPU is more likely to become a bottleneck resource than memory [11], especially when elements are densely consolidated in MSFC. Note that MicroNF provides the optimized strategy, which could be applied over previous research efforts on NF or element scheduling mechanisms in the user space [11], [37].

MicroNF takes an incrementally adaptive adjustment scheduling algorithm. The algorithm tries to match the processing speed of each element with its packet arrival rate. It incrementally adjusts the CPU utilization of the next scheduling period based on the statistics of current scheduling period. The detailed algorithm is introduced below.

In consolidation, CPU resources are scheduled among elements by periodically allocating *time slices* with CGroup [38]. Suppose scheduling period is T. For element i on a VM, we can get its CPU utilization proportion r_i by counting the number of time slices. Note that r_i is a proportion thus $\sum_i r_i = 1$. Also, we can get current buffer size B_i and last time buffer size B_i'. From Eq. 4, we can know the processing speed v_i satisfies $v_i = \phi_i^{-1}(r_i)$. We denote B_i^*, v_i^* and r_i^* as the *predicted* buffer size, processing speed and CPU utilization at the next scheduling period.

Our scheduling algorithm is based on the matching principle: *For all elements, their buffer variations should be proportional to respective processing speeds*, i.e.

$$\frac{B_i^* - B_i}{v_i^* T} = C,\ \forall i \in \{1, \cdots, n\} \tag{8}$$

By modeling in this way, we try to ensure fairness among elements and effectively allocate resources. The key idea is to match the processing speed with its respective packet arrival rates. In this way, the element with a lower processing speed and smaller flows can also attain an appropriate proportion of CPU. However, when several elements are consolidated together, downstream element may directly read the packet that are already loaded into memory by its upstream element. Thus we cannot get the actual packet arrival rate by simply measuring at the last switch. Naively adding statistics measuring module on the top of Docker will introduce unnecessary overhead. Instead, we can infer the arrival rate v_{ai} from the variation of buffer size, i.e.

$$v_{ai} = \frac{B_i - B_i'}{T} + v_i \tag{9}$$

As traffic usually does not vary sharply [39], we can assume that in a scheduling period (usually at millisecond level), the packet arrival rate keeps invariant, i.e. $v_{ai} = v_{ai}^*$. By substituting B_i^* in Eq. 8 with Eq. 9, we can get the CPU proportion for element i in the next scheduling period:

$$r_i^* = \phi_i(v_i^*) = \phi_i\left(\frac{\frac{B_i - B_i'}{T} + v_i}{C+1}\right) \tag{10}$$

The sum of CPU utilization needs to be normalized, thus C subjects to

$$\sum_i \phi_i \left(\frac{\frac{B_i - B_i'}{T} + v_i}{C+1} \right) = 1 \tag{11}$$

Although we cannot get an explicit expression of C, we can first randomly specify an initial value for C and then normalize r_i^*. Comparing to the millisecond level scheduling period, the solving time in this way is negligible.

Another important function of consolidation scheduler is to tell the individual scaler when to scale out. A direct indicator is the buffer size of an element. If buffer is overflowed, packet loss incurs and the element is definitely overloaded. At this time, scheduler can do nothing but execute the scaling up or scaling out methods, as introduced in Section VI.

VIII. Evaluation

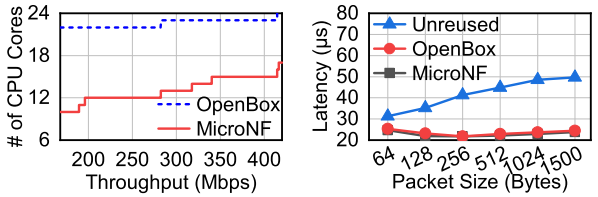
In this section, we first introduce our implementation details (Section VIII-A) and evaluate MicroNF by answering the following questions:

- Are the element reusing mechanisms in MicroNF *Graph Reconstructor* consistent, resource-efficient and high-performance? Our evaluation shows that MicroNF element reusing can improve resource efficiency by 29.1% to 54.5% with consistency ensured and performance maintained (Section VIII-B).
- Can MicroNF *Optimized Placer* provide high-performance placement solutions? Evaluation shows that the 0-1 programming could outperform strawman solutions by 1.21 times to 2.46 times on different topologies with negligible computation overhead (Section VIII-C).
- Can MicroNF *Individual Scaler* improve the performance and resource efficiency? Experiment results demonstrate that with push-aside scaling up mechanism, the latency of MSFC could be reduced by 1.8 times while the resource efficiency could be improved by 1.5 times (Section VIII-D).
- How can network operators deploy MicroNF in practice? We introduce a convenient measurement method for the parameters MicroNF needs and present a simplified method to the optimization procedure (Section VIII-E).

A. Implementation Details

1) Experimental Environment: We build MicroNF with Docker [17] to consolidate elements on VMs, and enable inter-VM packet forwarding with Open vSwitch [15]. We deploy the low-level dynamical element migration mechanism from OpenNF[5] [14]. We evaluate MicroNF based on a testbed with one server equipped with two Intel® Xeon® E5-2690 v2 CPUs (3.00GHz, 8 physical cores), 256GB RAM, and two 10Gbps NICs. The server runs Linux kernel 4.4.0-31. We use DPDK [34] on another server that directly connect to the server above to generate packets according to LBNL/ICSI enterprise traces [32].

[5]http://opennf.cs.wisc.edu/code.



(a) Resource efficiency. (b) Processing latency.

Fig. 12. Comparing MicroNF reusing mechanism against other baselines.

2) Tested Elements: We implement all the elements presented in the MSFC in Fig. 2 and 6. To make a fair comparison, the elements evaluated in our experiments are largely inherited from those in OpenBox[6] [6]. The Header Classifier composes a ruleset of 4560 firewall rules. The Payload Classifier is constructed from Snort IDS. The Modifier is built from a Layer 3 Network Address Translator to modify the output ports and destination IP addresses.

B. Element Reusing

1) Consistency Validation: We validate the consistency guarantee of MicroNF by comparing it with the no consistency guarantee reusing in OpenBox [6]. We evaluate the processing graph in Fig. 2. For Firewall rules, we randomly select 15% of flows based on the srcIP-dstIP pair from our evaluation traces to be *dropped* while *allowing* the rest to pass. We repeat the experiment for 1000 times to eliminate the randomness.

According to the analysis above, the classifiers from NAT and Firewall are unreusebale because the write field of modifiers from NAT ($srcIP \cup dstIP$) will be read by the classifier from Firewall. However, naively reusing the processing graph, just as the traditional method does, will lead to inconsistency. Experiment results show that without consistency guarantee in TranSys, the precision of Firewall will be decreased by 14.79%. Also, additional 2.31% of packets are recalled due to the incorrect classification. In contrast, MicroNF guarantees the consistency with the precision and recall the same as two unreused elements, i.e. the reused element with MicroNF is faithful towards the original one.

2) Resource Efficiency Improvement: To evaluate the resource efficiency of MicroNF, we compare the resource taken by the processing graph reused by MicroNF with the result of the traditional method in OpenBox, as shown in Fig. 6(a) and 6(b). We allocate one CPU core for each element and gradually increase the throughput. We use one CPU core to manage the elements. We scale the element out with another CPU core if overloaded (which means CPU usage exceeds 80%). The number of CPU cores taken by the two methods at different throughputs is shown in Fig. 12(a).

As shown in Fig. 6(a), there are 21 elements to implement[7] when reusing with the traditional method. Thus it takes at least 22 CPU cores (including a manager) to implement the processing graph. In contrast, MicroNF only needs to

[6]https://github.com/OpenBoxProject

[7]Drop element is a terminal element that can be performed with other elements. Thus it does not need to be implemented separately.

implement 9 elements. MicroNF achieves a resource efficiency improvement from 29.1% to 54.5% at different throughputs up to 420 *MB/s*.

3) Performance Benefit Maintenance: To evaluate the performance, we compare the processing latency of the element graph reused by MicroNF (Fig. 6(b)) with the unreused MSFC (Fig. 2) and the element graph reused by OpenBox (Fig. 6(a)). We simulate with the traffic from LBNL/ICSI enterprise traces [32] and calculate the average latency. The results are shown in Fig. 12(b).

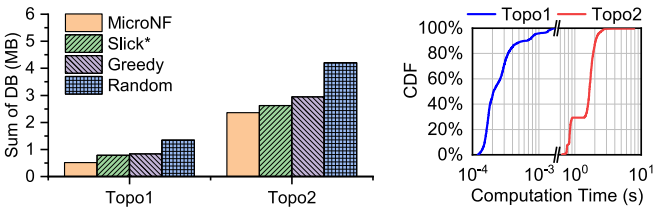
Compared to the unreused MSFC, MicroNF keeps the performance benefit on total processing latency brought by element reusing. Meanwhile, the latency in MicroNF steadily outperforms OpenBox a little by 2.8%. The additional benefit is gained from the elaborately designed element reusing mechanisms. When reusing MAT classifiers, MicroNF decreases the average number of entries to look up while OpenBox multiplicatively increases the entries. As our future work, instead of merely reusing branchers, we can reuse more types of element and achieve a higher performance improvement. We will also carefully analyze the latency benefit of different elements and further reduce the processing latency.

C. Performance-Aware MSFC Placement

As for evaluating the performance of placement algorithm, we evaluate the total *DB* in the processing graph. We use an Optimization Toolbox [40] to solve the 0-1 Quadratic Programming. We implement Topology 1 in Fig. 8(a) and Topology 2 in Fig. 8(b). In the two topologies, all elements are implemented as the Header Classifier as we introduced in Section VIII-A.2 to eliminate the influence from element types and demonstrate the performance MicroNF placement algorithm solely. We randomly select flows from the LBNL/ICSI enterprise trace [32] to different chains and repeat the experiment for 1000 times to eliminate the randomness. We try to place Topology 1 on 2 VMs and Topology 2 on 4 VMs.

Since that there is no ready-made solution on *which elements to consolidate*, we compare MicroNF with three baselines, including a *greedy* mechanism, a *random* mechanism, and a modified Slick mechanism (denoted as Slick*). The greedy mechanism first calculates the capacity for each vNF and then greedily places elements onto VMs chain by chain. This is to minimize the intra-chain transmissions and improve overall performance. The random mechanism naively and randomly selects available VMs to place elements. For Slick*, since Slick is designed to optimize the consolidation of NFs spreading many servers, we implement the key idea to optimally break MSFCs into several sub-chains chain-by-chain and consolidate them onto the same VM.

An important feature of MicroNF placement is performance-aware, which is evaluated as the sum of *DB*. The total *DB* in processing graph of successful placements with different strategies is shown in Fig. 13(a). For Topology 1, MicroNF reduces the total *DB* by at least 1.51× even compared to Slick* strategy and more against other baselines. For Topology 2, even the lengths of chains increase and more inter-VM packet transfers are unavoidable, MicroNF still outperforms the random strategy by 1.78× and greedy strategy by 1.12×. The main reason for the performance improvements of MicroNF against other heuristics is that MicroNF globally optimize the inter-VM latency.



(a) Performance (b) Computation time.

Fig. 13. Experiment results of MicroNF *Optimized Placer* on two topologies.

TABLE III
PLACEMENT FAILURE RATE (OF 1000 TESTS)

	MicroNF	Slick*	Greedy	Random
Topology 1	11.8%	11.8%	16.3%	23.7%
Topology 2	5.6%	6.5	6.5%	7.7%

Another feature of MicroNF is resource efficient by compact placement, which can be interpreted as: *Given limited CPU cores, i.e. VMs, for different traffic, MicroNF has a higher probability to place all of them on successfully.* As shown in Table III, when placing Topology 1, MicroNF improves the failure rate by at least 1.38$times$ against Slick*. Note that the placement of Topology 1 ($\frac{6 \text{ elements}}{2 \text{ VMs}} = 3$) is tighter than Topology 2 ($\frac{9 \text{ elements}}{4 \text{ VMs}} = 2.25$), thus placing Topology 1 has a higher failure rate than Topology 2. Even so, MicroNF improves the failure rate from 7.8% (random) and 6.5% (greedy) to 5.8% for Topology 2.

We finally measure the computation overhead for MicroNF *Optimized Placer*. Among 1000 experiments, MicroNF *Optimized Placer* is able to generate the optimized solutions within 0.01 second for Topology 1 and 10 seconds for Topology 2 on our testbed, as shown in Fig. 13(b). The computation time of Topology 2 is longer than that of Topology 1 since it has more nodes and edges. Note that this algorithm *runs offline only once* for initial placement before deploying elements onto devices. Thus a computation time of 10s is negligible compared to the minute-level time for program compilation and element deployment. Meanwhile, for large-scale and complicated topologies, network operators could efficiently solve the quadratic programming by adopting state-of-the-art commercial solvers (*e.g.* IBM CPLEX [41], Gurobi [42]) to further reduce the computation overhead.

D. Push-Aside Scaling Up

To evaluate the performance of push-aside scaling up, we use the MSFC in Fig. 11. At first, the throughput of MSFC is 100 kpps, with each packet 512 B. At the 10k-th packet, we increase the traffic to 150 kpps, which causes the Stateful Payload Analyzer overloaded. The traditional method taken by OpenNF [14] naively scales out by copying Stateful Payload Analyzer to a newly started VM, as shown in Fig. 11(c).

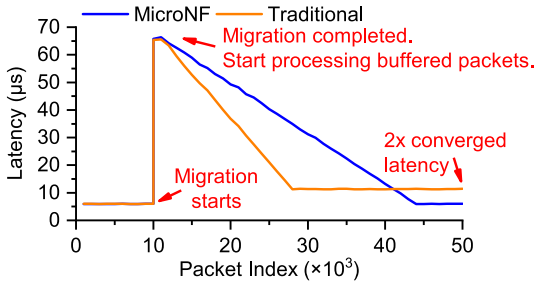


Fig. 14. Real-time latency per packet.

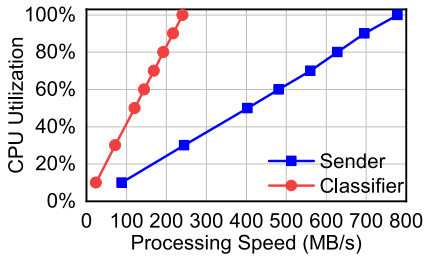


Fig. 15. Throughput-CPU utilization mapping on a single core.

In contrast, MicroNF migrates Logger to VM1 and allocate the released resource to Stateful Payload Analyzer.

For performance, the comparison of two methods on real-time latency of each packet is shown in Fig. 14. The latency at 10k-th packet increases sharply due to the element migration. The convergence time of traditional method ($\frac{28k-10k}{150kpps} = 0.12s$) is faster than that of MicroNF ($\frac{44k-10k}{150kpps} = 0.23s$) since it consumes more resources and has a higher processing speed. Nonetheless, both convergence time are acceptable in real-world network scenarios since elements would not be frequently scaled [43]. However, MicroNF has a lower converged latency of 6 *ms* compared to 11 *ms* of the traditional method. The improvement is achieved by reducing the additional packet transfer and state synchronization latency. For resource efficiency, with traditional method, the scaled MSFC is allocated with more resources (3 VMs in total). In contrast, 2 VMs are enough with MicroNF in this situation (Fig. 11(b)) by push-aside scaling up. MicroNF achieves a higher resource efficiency by 1.5×.

E. Throughput-CPU Utilization Mapping

To measure the throughput-CPU utilization mapping, we constrain the available CPU utilization for the element by fixing the `cpu-period` and changing the `cpu-quota` parameter in Docker. We use a packet sender to test the maximum throughput under the current limited CPU proportion. We provide an efficient and automatic tool to dynamically adjust the resource allocations and measure its throughput capacity. With this method, administrators can get the mapping function $\phi(v)$.

We measure two types of element with different complexity. Packet Sender represents elements with simple processing logic. IP Address-based Header Classifier contains 100 rules and represents relatively complicated elements. The mapping functions are shown in Fig. 15. Surprisingly, a strong linearity correlation can be observed. We present the linear regressions of the two mapping functions:

- **Sender:** $r = -0.022 + 0.0013 \times v,\ R^2 = 0.9997$
- **Classifier:** $r = 0.00048 + 0.0042 \times v,\ R^2 = 0.9999997$

$r \in [0, 1]$ is the CPU utilization and v is the processing speed in *MB/s*. R^2 is a measure of goodness of fit with a value of 1 denoting a perfect fit. Thus in practice, we can further simplify the solving procedure by substituting $\phi(v)$ and $\phi^{-1}(r)$ with their linear approximations.

IX. Discussions

In this section, we discuss how to extend MicroNF and highlight several open issues as future directions.

Multi-core placement analysis: For simplicity, MicroNF assumes that each VM is allocated with one CPU core when optimizing placement to satisfy the general applications. In the large-scale deployment in practice, tenants might allocate multiple CPU cores to a VM. In this case, MicroNF can be easily extended by considering the resource constraint of multiple CPU cores instead of a single core.

Intra-core analysis: MicroNF analyzes the inter-core cost caused by vSwitch-based packet transfer. As our future work, by designing cache replacement policies, we may reduce the miss rate of Layer 1 and 2 Cache and further reduce repeatedly packet loading from memory to cache. Moreover, more designs are needed to ensure isolation between consolidated elements. However, those analyses are infrastructure-dependent and differs on various types of CPU, which is beyond our scope. MicroNF can be easily extended to analyze intra-core situations on a certain type of CPU.

Framework Generality: The design of MicroNF controller (Graph Reconstructor, Optimized Placer and Individual Scaler) builds on the high latency of virtual switch between elements and is agnostic to underlying infrastructure, as we discussed in Section III. For other types of devices (*e.g.* CPU, GPU, SmartNIC, or even programmable switches) since they usually need carefully-crafted resource utilization models [44], MicroNF could not be directly adopted. However, the key ideas (e.g. push-aside migration) could also be applied. We leave the heterogeneous resource optimizations of elements between different types of devices for future work.

X. Conclusion and Future Work

This paper presents MicroNF, a high performance and efficient resource management framework, for providing compact and optimized element consolidation in MSFC. MicroNF addresses the problem of which elements to consolidate in the first place and provides a performance-aware placement algorithm based on 0-1 Quadratic Programming. MicroNF also innovatively proposes a push-aside scaling up strategy to avoid performance degradation in scaling. MicroNF further designs an automatic CPU scheduler aware of the difference of processing speed between elements. Our preliminary evaluation results show that MicroNF could reduce packet transfer cost by up to 2.46× and improve performance at scaling by 45.5% with more efficient resource utilization. As our future work, we will exploit other features of modularization,

such as *customization*. Building microservice-based NFV gives us the potential to easily customize new microservices, which could accelerate the development cycle.

ACKNOWLEDGMENT

The authors thank for suggestions from anonymous JSAC reviewers. Zili Meng specially thanks for the support from Prof. Mingwei Xu in Tsinghua University. The preliminary version of this paper titled "CoCo: Compact and Optimized Consolidation of Modularized Service Function Chains in NFV" was published in IEEE ICC 2018 [1].

REFERENCES

[1] Z. Meng, J. Bi, C. Sun, H. Wang, and H. Hu, "CoCo: Compact and optimized consolidation of modularized service function chains in NFV," in *Proc. 53rd IEEE Int. Conf. Commun. (ICC)*, May 2018, pp. 1–7.

[2] R. Mijumbi *et al.*, "Network function virtualization: State-of-the-art and research challenges," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 1, pp. 236–262, 1st Quart., 2016.

[3] J. Halpern and C. Pignataro, *Service Function Chaining (SFC) Architecture*, document RFC 7665, IETF, 2015.

[4] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat, "xOMB: Extensible open middleboxes with commodity servers," in *Proc. 8th ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, Oct. 2012, pp. 49–60.

[5] B. Anwer, T. Benson, N. Feamster, and D. Levin, "Programming slick network functions," in *Proc. 1st ACM SIGCOMM Symp. Softw. Defined Netw. Res. (SOSR)*, 2015, Art. no. 14.

[6] A. Bremler-Barr, Y. Harchol, and D. Hay, "OpenBox: A software-defined framework for developing, deploying, and managing network functions," in *Proc. Conf. ACM Special Interest Group Data Commun. (SIGCOMM)*, 2016, pp. 511–524.

[7] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *Proc. 9th USENIX Conf. Netw. Syst. Design Implement. (NSDI)*. Berkeley, CA, USA: USENIX Association, 2012, p. 24.

[8] N. Zhang, H. Li, H. Hu, and Y. Park, "Towards effective virtualization of intrusion detection systems," in *Proc. ACM Int. Workshop Secur. Softw. Defined Netw., Netw. Function Virtualization*, 2017, pp. 47–50.

[9] W. Zhang, J. Hwang, S. Rajagopalan, K. K. Ramakrishnan, and T. Wood, "Flurries: Countless fine-grained NFs for flexible per-flow customization," in *Proc. 12th Int. Conf. Emerg. Netw. Exp. Technol. (CoNEXT)*, 2016, pp. 3–17.

[10] J. Hwang, K. K. Ramakrishnan, and T. Wood, "NetVM: High performance and flexible networking using virtualization on commodity platforms," in *Proc. 11th USENIX Conf. Netw. Syst. Design Implement. (NSDI)*. Berkeley, CA, USA: USENIX Association, 2014, pp. 445–458.

[11] S. G. Kulkarni *et al.*, "NFVnice: Dynamic backpressure and scheduling for NFV service chains," in *Proc. Conf. ACM Special Interest Group Data Commun. (SIGCOMM)*, 2017, pp. 71–84.

[12] H. Khazaei, C. Barna, N. Beigi-Mohammadi, and M. Litoiu, "Efficiency analysis of provisioning microservices," in *Proc. IEEE Int. Conf. Cloud Comput. Technol. Sci. (CloudCom)*, Dec. 2016, pp. 261–268.

[13] S. Newman, *Building Microservices*. Sebastopol, CA, USA: O'Reilly Media, 2015.

[14] A. Gember-Jacobson *et. al.*, "OpenNF: Enabling innovation in network function control," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 163–174, 2014.

[15] B. Pfaff *et al.*, "The design and implementation of open vswitch," in *Proc. 12th USENIX Conf. Netw. Syst. Design Implement. (NSDI)*. Berkeley, CA, USA: USENIX Association, 2015, pp. 117–130.

[16] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, Aug. 2000.

[17] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, p. 2, 2014.

[18] M. C. Luizelli, L. R. Bays, L. S. Buriol, M. P. Barcellos, and L. P. Gaspary, "Piecing together the NFV provisioning puzzle: Efficient placement and chaining of virtual network functions," in *Proc. IFIP/IEEE Int. Symp. Integr. Netw. Manage. (IM)*, May 2015, pp. 98–106.

[19] S. Mehraghdam, M. Keller, and H. Karl, "Specifying and placing chains of virtual network functions," in *Proc. IEEE 3rd Int. Conf. Cloud Netw. (CloudNet)*, Oct. 2014, pp. 7–13.

[20] M. Savi, M. Tornatore, and G. Verticale, "Impact of processing costs on service chain placement in network functions virtualization," in *Proc. IEEE Conf. Netw. Function Virtualization Softw. Defined Netw. (NFV-SDN)*, Nov. 2015, pp. 191–197.

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

[22] W. Zhang *et al.*, "OpenNetVM: A platform for high performance network service chains," in *Proc. ACM SIGCOMM Workshop Hot Topics Middleboxes Netw. Function Virtualization (HotMiddleBox)*, 2016, pp. 26–31.

[23] S. Palkar *et al.*, "E2: A framework for NFV applications," in *Proc. 25th ACM Symp. Oper. Syst. Princ. (SOSP)*, 2015, pp. 121–136.

[24] S. Radhakrishnan, S. V. Raghavan, and A. K. Agrawala, "A flexible traffic shaper for high speed networks: Design and comparative study with leaky bucket," *Comput. Netw. ISDN Syst.*, vol. 28, no. 4, pp. 453–469, 1996.

[25] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.

[26] 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. (SIGCOMM)*, 2017, pp. 43–56.

[27] J. Deng *et al.*, "On the safety and efficiency of virtual firewall elasticity control," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2017, pp. 1–15.

[28] A. S. Tanenbaum, *Modern Operating Systems*. London, U.K.: Pearson, 2009.

[29] 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.

[30] A. Caprara, "Constrained 0–1 quadratic programming: Basic approaches and extensions," *Eur. J. Oper. Res.*, vol. 187, no. 3, pp. 1494–1503, 2008.

[31] 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.

[32] (2005). *LBNL/ICSI Enterprise Tracing Project*. [Online]. Available: http://www.icir.org/enterprise-tracing

[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. (SIGCOMM)*, 2017, pp. 15–28.

[34] I. DPDK. (2010). *Data Plane Development Kit*. [Online]. Available: http://dpdk.org

[35] D. A. Joseph, A. Tavakoli, and I. Stoica, "A policy-aware switching layer for data centers," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 51–62, 2008.

[36] D. Stiliadis and A. Varma, "Rate-proportional servers: A design methodology for fair queueing algorithms," *IEEE/ACM Trans. Netw.*, vol. 6, no. 2, pp. 164–174, Apr. 1998.

[37] Y. Li, L. T. X. Phan, and B. T. Loo, "Network functions virtualization with soft real-time guarantees," in *Proc. 35th Annu. IEEE Int. Conf. Comput. Commun. (INFOCOM)*, Apr. 2016, pp. 1–9.

[38] (2007). *Linux Container*. [Online]. Available: https://linuxcontainers.org/

[39] 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. (CoNEXT)*, 2011, Art. no. 8.

[40] (2015). *MATLAB Optimization Toolbox*. [Online]. Available: https://www.mathworks.com/products/optimization.html

[41] (2018). *CPLEX Optimizer*. [Online]. Available: https://www.ibm.com/analytics/data-science/prescriptive-analytics/cplex%-optimizer

[42] *Gurobi Optimizer—The State-of-the-Art Mathematical Programming Solver*. Accessed: Jan. 28, 2019. [Online]. Available: http://www.gurobi.com/

[43] C. Sun, J. Bi, Z. Meng, T. Yang, X. Zhang, and H. Hu, "Enabling NFV elasticity control with optimized flow migration," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 10, pp. 2288–2303, Oct. 2018.

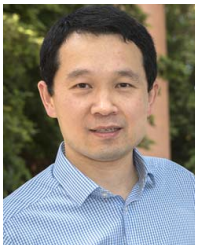
[44] B. Li *et al.*, "ClickNP: Highly flexible and high-performance network processing with reconfigurable hardware," in *Proc. Conf. ACM SIGCOMM Conf.*, 2016, pp. 1–14.

Zili Meng is currently pursuing the bachelor's degree with the Department of Electronic Engineering, Tsinghua University. He has authored or coauthored papers in ACM SIGCOMM, IEEE JSAC, and so on. His research interests include learning-based network systems and network function virtualization. He was the Winner of the Student Research Competition in ACM SIGCOMM 2018.

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 was a Changjiang Scholar Distinguished Professor with Tsinghua University, where he was the Director of the Network Architecture Research Division, Institute for Network Sciences and Cyberspace. He successfully led tens of research projects. He has published more than 200 research papers and 20 Internet RFCs or drafts. He holds 30 innovation patents. His previous research interests include Internet architecture, software-defined networking (SDN)/network function virtualization (NFV), and network security. He received the National Science and Technology Advancement Prizes, the IEEE ICCCN Outstanding Leadership Award, and best paper awards. He was a Distinguished Member of the China Computer Federation (CCF).

Haiping Wang received the B.S. degree from the Department of Software Engineering, Southeast University, Nanjing, China, in 2018. 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 function virtualization.

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 Department of Computer Science and Technology, Tsinghua University. He has published papers in SIGCOMM, JSAC, ToN, IWQoS, *IEEE Communications Magazine*, *IEEE Network Magazine*, and so on. His research interests include Internet architecture, software-defined networking, and network function 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 Associate 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 received the NSF CAREER Award in 2019. He was a recipient of the Best Paper Award from ACM SIGCSE 2018 and ACM CODASPY 2014 and the Best Paper Award Honorable Mention from ACM SACMAT 2016, IEEE ICNP 2015, and ACM SACMAT 2011.