HYPER: A Hybrid High-Performance Framework for Network Function Virtualization

Chen Sun, Jun Bi, *Senior Member, IEEE*, Zhilong Zheng, and Hongxin Hu, *Member, IEEE*

***Abstract*— Network function virtualization (NFV) offers the potential for both enhancing service delivery flexibility and reducing overall costs by virtualizing network functions that are traditionally implemented in dedicated hardware. However, the flexibility of NFV comes with considerable compromises since virtual machine carried functions could introduce significant performance overhead. In this paper, we present a novel high-performance framework called HYPER, which combines programmable hardware infrastructure and traditional software infrastructure in NFV to achieve both high performance and flexibility for supporting virtualized network functions (VNFs). In HYPER, we design a mediator layer to hide underlying infrastructure heterogeneity from the NFV orchestrator to simplify VNF management. In addition, we design a SLA-aware service chaining algorithm in HYPER to leverage the benefits of the hybrid infrastructure to fulfill both functional and performance requirements from service subscribers (or tenants). To optimize resource utilization efficiency, we also introduce a performance-aware VNF placement algorithm in HYPER, which accommodates both resource and performance requirements in placing VNFs. We implement HYPER in a testbed based on OpenStack and ONetCard. Experimental results show that HYPER reduces the forwarding latency of a service chain by 40% to 67% compared with data plane development kit - based implementation, while maintaining the flexibility of VNF management.**

***Index Terms*— Network function virtualization, hybrid infrastructure, high performance, SLA-aware service chaining.**

I. Introduction

IN TRADITIONAL networks, enabling new network services often needs to add new proprietary middleboxes. However, finding the space and power to accommodate these middleboxes is becoming increasingly difficult, along with the increasing costs of energy and capital investment. There is no general hardware framework for middleboxes, and they are hard to scale out at peak load.

Network Function Virtualization (NFV) was recently introduced to address the limitations of dedicated middleboxes and offers the potential for both enhancing service delivery flexibility and reducing overall costs [1]. Considerable efforts have been devoted to exploiting the flexibility of virtualization to provide more scalable services [2], [3]. Rajagopalan *et al.* [2] presented a state-centric and system-level abstraction for elastic middleboxes called Split/Merge, based on which Gember-Jacobson *et al.* [3] proposed OpenNF to help NFV network operators accurately monitor and manipulate network states. Current research on NFV mainly targets on software-based Virtualized Network Functions (VNFs), which could provide high flexibility and high scalability with low costs. However, above benefits of NFV come with considerable compromises. Especially, packet processing in software-based VNFs could introduce significant performance overhead (e.g., Ananta SMux at 100 Kpps can add from 200μs to 1ms of forwarding latency), which may be unacceptable for some network applications, such as algorithmic stock trading and high performance distributed memory caches that demand ultra-low (a few microseconds) latency [4]. Advanced technologies such as Data Plane Development Kit (DPDK) [5] can decrease the forwarding latency and improve throughput to a large extent. However, according to our experiments, DPDK's processing latency of a stateful firewall instance is still 6.4× of hardware-based processing latency (in average), and its latency jitter of 64 byte packets could vary from 8μs to 430μs. Such long latency and significant jitter are still unacceptable for many network applications.

To solve the performance problem of NFV, some recent work proposed to use programmable hardware accelerators (e.g., FPGA) to support NFV [6], [7]. Ge *et al.* [6] revealed the gap between software-based middleboxes and commodity hardware, and proposed to integrate elastic FPGA into OpenStack-based NFV to improve NFV performance. Kachris *et al.* [7] recognized FPGA as an ideal platform for supporting NFV, since FPGA can provide both the flexibility of virtualization and the high performance of specialized hardware. Indeed, contemporary FPGA devices can perform network processing functions at up to 400 Gbit/sec line rates. Thus, such programmable and reconfigurable hardware can achieve almost equal performance to dedicated ASIC devices [8]. However, those existing solutions only utilize high performance hardware to support NFV, without leveraging the

Manuscript received April 1, 2017; revised September 12, 2017; accepted September 25, 2017. Date of publication October 6, 2017; date of current version December 1, 2017. This work was supported in part by the National Key R&D Program of China under Grant 2017YFB0801701 and in part by the National Science Foundation of China under Grant 61472213. *(Corresponding author: Jun Bi.)*

C. Sun, J. Bi, and Z. Zheng are with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China, also with the Department of Computer Science, Tsinghua University, Beijing 100084, China, and also with the Tsinghua National Laboratory for Information Science and Technology, Beijing 100084, China (e-mail: c-sun14@mails.tsinghua.edu.cn; junbi@tsinghua.edu.cn; zhengzl15@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.2017.2760438

benefits and features of commodity servers that could provide *abundant* resource and *high* flexibility.[1]

To address above challenges and limitations, in this paper, we present HYPER, a HYbrid high-PERformance framework, which leverages programmable hardware to establish a high performance infrastructure and seamlessly integrates it with traditional software infrastructure in NFV, to achieve both high performance and flexibility for supporting VNFs. HYPER can accommodate state of the art hardware and software platforms as long as they expose the unified control interfaces. We design a mediator layer in HYPER for hardware and software VNF management to hide the heterogeneity of underlying infrastructure from the NFV orchestrator. The orchestrator simply needs to describe its requirements for new VNF instances, while the mediator can automatically identify the most suitable infrastructure for placing the instances. In addition, in virtue of the hybrid infrastructure in HYPER, service providers are capable of provisioning *various* Service Level Agreements (SLAs). Therefore, to differentiate from traditional function-aware service chaining [12]–[15], we propose a SLA-aware service chaining algorithm in HYPER leveraging high performance of hardware and flexibility of software to satisfy both performance and functional requirements from tenants. We also introduce a performance-aware VNF placement algorithm to improve resource utilization efficiency and satisfy performance requirements in placing VNFs.

We summarize our contributions as follows:

- We propose a novel high-performance NFV framework called HYPER, which leverages hybrid infrastructure combining hardware and software features to achieve both high performance and flexibility for supporting VNFs.
- We design a mediator in HYPER to hide infrastructure heterogeneity from the NFV orchestrator and relieve the orchestrator from infrastructure-related VNF management.
- We introduce a SLA-aware service chaining algorithm to meet both performance and functional requirements from tenants. We also design a performance-aware VNF placement algorithm to optimize resource utilization efficiency in placing VNFs.
- We implement a prototype of HYPER based on OpenStack and ONetCard [16] to demonstrate its feasibility. Experimental results show that a hybrid service chain built in HYPER can achieve up to 67% latency decrease compared with a pure software-based service chain.

The rest of the paper is organized as follows. We elaborate challenges in Section II. The design of HYPER framework is articulated in Section III. We present the SLA-aware service chaining algorithm in Section IV and the performance-aware VNF placement algorithm in Section V. We introduce the implementation in Section VI, and present our evaluation in Section VII. We summarize the related work in Section VIII, and conclude this paper in Section X.

[1]TCAM resources on hardware devices are actually rather limited and expensive to support complex and resource-intensive VNFs such as NAT and DPI (e.g., top grade switches from most manufacturers include a TCAM of 2-4K OpenFlow entries [11]).

II. Design Challenges and Contributions

In this section, we reveal three major challenges in introducing hybrid infrastructure in HYPER.

A. Management of Heterogeneous Infrastructure

HYPER aims to accommodate cutting-edge programmable hardware platforms, such as FPGA, P4 [9], and RMT [10], and software virtualization platforms, such as ClickOS [17], Split/Merge [2], and OpenNF [3], to provide both high performance and flexibility for NFV. However, above techniques could vary in external control interfaces and internal processing strategies. This creates the non-trivial task of managing VNFs on heterogeneous infrastructure. To relieve the orchestrator from infrastructure-related management, we design a mediator layer to automatically identify the infrastructure type and location to deploy VNF instances. Besides, we extend platforms integrated in HYPER with *unified* control interfaces and expose them to the mediator, which could hide the interface difference of heterogeneous platforms and simplify VNF management. We demonstrate its feasibility by extending the VNF Manager of both ONetCard based hardware VNFs and commodity servers carried VNFs.

B. SLA-Aware Service Chaining

Hybrid infrastructure of HYPER provides the possibility of provisioning services that meet various SLAs including forwarding latency, throughput, and resource reservation. Therefore, it is challenging for the orchestrator to perform service chaining with respect to both functional and SLA requirements from tenants. Existing research efforts [12]–[15] can only chain required functionalities. However, we design a novel SLA-aware service chaining algorithm that can take both functional and SLA requirements as inputs. The algorithm can leverage high performance of hardware, and rich resource and flexibility of software, to *quickly* construct a qualified *hybrid* service chain.

C. Performance-Aware VNF Placement

HYPER tenants might raise performance requirements in placing VNFs. Also, the hybrid infrastructure integrated in HYPER could vary in performance and capabilities. In traditional VM-based NFV, the strawman placement mechanism is to satisfy resource requirements of VNF instances. In contrast, we introduce a *fast* and *general* VNF placement algorithm that can quickly decide the infrastructure type and location to place new instances with respect to resource and performance requirements, and accommodate those instances with minimal device costs.

III. HYPER Framework

As shown in Fig. 1, we abstract a NFV-based network into four layers in HYPER. The orchestrator layer makes decisions of VNF deployment, destruction and migration, and service chaining. The mediator layer translates decisions from the orchestrator into policies and delivers them to corresponding modules including the VNF managers that control the

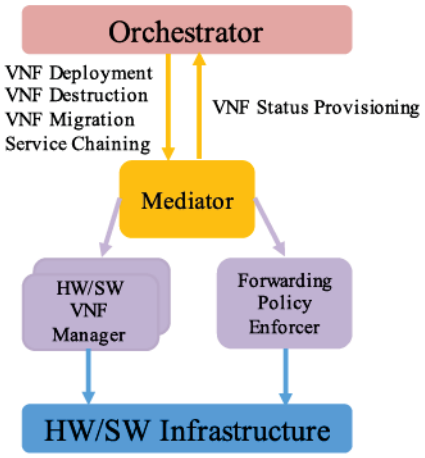


Fig. 1. HYPER framework overview.

life cycle of hardware or software VNFs, and a forwarding policy enforcer that could be implemented based on a SDN controller to issue forwarding policies to steer traffic among VNF instances. In the bottom, we combine the programmable hardware infrastructure with the software infrastructure to construct a hybrid infrastructure layer.

A. Unified Control Interfaces for Hybrid Infrastructure

VNFs in HYPER framework can come from service providers, third-party vendors, managers of programmable hardware platforms such as P4 and RMT, and software virtualization platforms such as ClickOS, Split/Merge and OpenNF. To accommodate hybrid infrastructure in HYPER, similar to [1], each VNF carried by hardware or software owns its corresponding VNF Manager (VNFM), which can be integrated into HYPER under the management of the mediator.

Various types of infrastructure creates the challenge of managing VNFs carried by different platforms. Instead of burdening the mediator to manage heterogeneous VNFMs with different interfaces and internal logics, we design a unified control interface set in HYPER for VNFMs of all platforms. We require that VNFMs should implement the recommendations of ETSI standards [18]. Specifically, we focus on a minimal set of control interfaces including:

- `deploy_instance(`**`instance`**`)`
- `destruct_instance(`**`instance_ID`**`)`
- `configure_instance(`**`instance_ID, configuration`**`)`
- `migrate_instance(`**`old_ID, new_ID, flows`**`)`
- `get_instance_state(`**`instance_ID`**`)`

An **instance** entity for hardware is represented as processing logic and configurations. For example, P4 [9] programs are first converted into an intermediate table dependency graph representation, then mapped onto the hardware platform's specific logics. For the software infrastructure, deploying a new instance usually means installing a system image of the VNF into a newly instantiated VM.

VNF instances can be deployed and destructed through `deploy_instance` and `destruct_instance` interfaces. The mediator can configure all instances through the `configure_instance` interface. As the instance configuration formats vary in different VNFMs, we propose that each VNFM formats their configuration into *formal languages* such as XML. In this way, the configuration of all VNFMs can be transformed into byte strings and transmitted to VNFs. The mediator is also capable of instructing a specific VNFM to migrate some flows from the old one to the new one through the `migrate_instance` interface. During runtime, the orchestrator also needs to query instance states, including resource utilization, workload, and current performance through `get_instance_state` interface, for network monitoring, service chaining, and VNF management.

Above interfaces can be easily implemented (about 300 LoC in ONetCard based VNFMs). They could hide the underlying heterogeneity and simplify VNF management for the mediator.

B. Mediator

1) Why Mediator: Both software VNFM (S-VNFM) and hardware VNFM (H-VNFM) can be accommodated into the HYPER framework. Therefore, both software and hardware implementations of the same VNF, such as stateful firewall, IDS, may co-exist in HYPER networks. However, software and hardware implementations may vary in performance and resource capacity. Therefore, the orchestrator is burdened to pick the most suitable infrastructure to support a VNF and select the best location to place it. For instance, if the orchestrator intends to deploy a new IDS instance in hybrid infrastructure, it must first decide what kind of infrastructure, i.e. software or hardware, is used to build the new instance. Then, the orchestrator should find the most suitable location to place it. Finally, the orchestrator needs to inform the related VNFM to deploy the instance. Coupling orchestration and placement tasks together could decrease the efficiency of orchestration and make HYPER difficult to integrate new platforms.

To address the above problem, we design a mediator module in HYPER framework. The purpose of designing the mediator is to take service chaining and VNF management policies from the orchestrator and deliver the policies to the most suitable modules through infrastructure-related calculation. The mediator also collects VNF status, including performance and resource utilization, from VNFMs and provisions the status to the orchestrator. Therefore, to deploy a VNF, the orchestrator could simply describe its requirements on the new instance. Then, the mediator can automatically pick the best infrastructure and location to accommodate the new instance. This decoupling of orchestration and placement tasks could relieve the orchestrator from infrastructure-related calculation and automate VNF placement. Besides, when integrating a new type of infrastructure into HYPER, we could only extend the mediator to recognize the new infrastructure without any changes of the orchestration process.

2) Mediator Design: The mediator takes service chaining policies or VNF management policies from the orchestrator and delivers tasks to corresponding modules. During runtime, the mediator collects VNF status from VNFMs and provisions

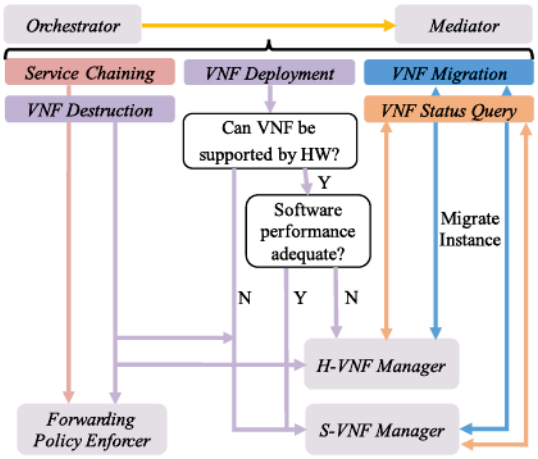


Fig. 2. Decision tree of the mediator.

the statistics to the orchestrator for managing service chaining. We consider four major kinds of tasks provided by the orchestrator including service chaining, VNF lifecycle management, VNF migration, and VNF status query. We build a *Decision Tree* for the mediator as shown in Fig. 2. For each type of policies, the mediator processes them as follows.

3) Service Chaining: For service chaining policies, the mediator first calculates the forwarding path of the flow according to the order of required instances. Then, it delivers the hop-by-hop path to the forwarding policy enforcer.

4) VNF Lifecycle Management: For VNF destruction policies, the mediator will steer traffic away and inform the corresponding VNFM to destroy the instance. For VNF deployment policies, the orchestrator will inform the mediator of the type, performance and resource requirements of the new instance. The mediator will proceed through the following steps. The first step is to decide the infrastructure type. If software performance can satisfy all requirements, the mediator will inform the S-VNFM to deploy a software instance for this VNF, as we assume *software resources are relatively abundant and cheap* compared with hardware [19]. Otherwise, the mediator will calculate the most suitable hardware device. Detailed algorithm will be elaborated in Section V.

5) VNF Migration: The mediator will deploy a new instance, inform the corresponding VNFM to migrate some flows from the old instance to the new one, and destruct the old instance. Finally, the mediator will inform the forwarding policy enforcer to steer affected flows to the new instance.

6) VNF Status Query: The orchestrator will query the resource usage, workload, and performance of each VNF instance for SLA-aware service chaining. The mediator will gather status from VNFMs and report it to the orchestrator.

C. Orchestrator

The orchestrator performs dynamic service chaining for tenants. Service chaining in HYPER is more complex due to various capabilities of the hybrid infrastructure and massive SLA requirements. The orchestrator runs a SLA-aware service chaining algorithm to meet both functional and performance requirements, which will be introduced in Section IV. It then issues the chaining decisions to the mediator, which automatically decide and perform VNF placement. In HYPER, the orchestrator is totally *unaware* of such a process, and is relieved of infrastructure-related placement.

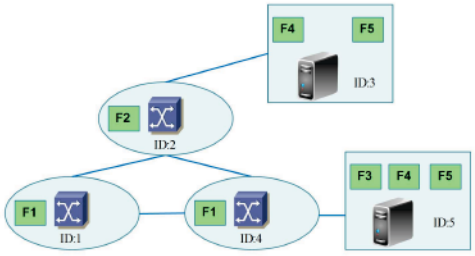


Fig. 3. Example topology for service chaining.

IV. SLA-Aware Service Chaining

Benefiting from the hybrid infrastructure, HYPER is capable of provisioning services of different performances or SLAs. Therefore, we design a SLA-aware service chaining algorithm in HYPER to satisfy both functional and SLA requirements from different tenants. As there exist several instances of the same VNF in the network, this algorithm addresses the challenge of quickly choosing the path that contains required VNFs and satisfies forwarding latency, throughput, and resource reservation (capacity) requirements.

A. Performance and Resource Model for VNFs

First, we establish a performance and resource model for hardware and software based instances. We mainly consider TCAM as resource for hardware, and CPU as resource for software. The model is denoted as follows:

```
hardware_instance          software_instance
| latency,                 | latency,
| throughput_total,        | throughput_total,
| throughput_used,         | throughput_used,
| tcam_total,              | cpu_total,
| tcam_used                | cpu_used
```

The orchestrator periodically queries resource usage and performance of VNF instances from the mediator as constraints for SLA-aware service chaining.

B. Service Chaining Algorithm

Based on the above resource model, we introduce the SLA-aware service chaining algorithm. The algorithm can be divided into three steps. Fig.3 depicts an example topology.

Step 1 (Functionality Fulfillment): Some required VNFs might not have been deployed in the network. The algorithm first goes over required functionalities, and instantiates undeployed VNFs according to performance and capacity requirements to make sure that at least one instance of each required VNF exists in the network.

Step 2 (Latency Fulfillment): We satisfy the forwarding latency requirement first, as it is relatively more complex compared

with other factors. Assume that a service chain should contain $Function\ F1,\ F4,\ F2$ in sequence. As shown in Fig.3, $F1$ instances reside in nodes 1, 4, $F4$ in nodes 3, 5 and $F2$ in node 2. Possible paths of this chain can be expressed with node IDs as:

$$S \rightarrow Stage\ 1(1,4) \rightarrow Stage\ 2(3,5) \rightarrow Stage\ 3(2) \rightarrow$$

We take the above expression as three *stages*, each containing nodes that support one required function. We use Dynamic Scheduling Algorithm to find the path with the lowest *latency*, denoted as P. We denote the starting and ending mark of P as S, E. Here S, E are simply two symbols and do not refer to any nodes. For each node $N_{x,i}$ in stage i, the shortest path passing $N_{x,i}$ is denoted as $short(S \xrightarrow{N_{x,i}} E)$. We have:

$$short(S \xrightarrow{N_{x,i}} E) = short(S \rightarrow N_{x,i}) + short(N_{x,i} \rightarrow E)$$

And:

$$P = \arg\min_{N_{x,i}}(short(S \xrightarrow{N_{x,i}} E))\ \ foreach\ N_{x,i}\ in\ stage\ i$$

We iterate each stage from S to E until we find the path with the lowest latency. We record the shortest delay from S to node x in stage i as $s[x][i]$, and store the previous node on this path in stage $i-1$ as $pre[x][i]$ to finally generate the entire path. We denote the forwarding latency between node x and y as $latency\_t[x][y]$, and the processing latency of node x as $latency\_p[x]$. The algorithm is articulated in Algorithm 1. Finally, we generate a path with the shortest forwarding latency. If this latency cannot satisfy tenant requirements, we iterate through each node in the path, request VNF placement with more strict latency constraint, and re-run the algorithm. If all VNFs are implemented in hardware and the latency still exceeds tenant requirement, HYPER will not be able to form a qualified service chain under this latency constraint.

The major computational complexity of the algorithm comes from the loop expressions. We denote the node number in stage i as $s[i]$. The total loop number is $\sum_{i=1}^{n-1} s[i] \times s[i+1] < (n-1) \times (max(s[i])\ for\ i \in [1,\ n])^2$. Therefore, if we denote the maximum node number in each stage as S_{max}, the computational complexity is $O(n * S_{max}^2)$.

Step 3 (Throughput and Capacity Fulfillment): The algorithm checks whether all nodes alongside the path can provide sufficient throughput and capacity. If so, a qualified path is generated and the algorithm ends. If any nodes fail the check, the nodes' latency should be marked as infinite to avoid picking this off-grade node in future, or deploy a new instance with more strict throughput or capacity constraints. Then the algorithm goes back to Step 2 and continues the cycle until a qualified path is produced or no possible paths exists.

A possible design choice for addressing throughput or capacity insufficiency is to separate a tenant's flow into two parts and use two instances of the same hardware or software VNF to process them in parallel. However, we argue against this solution. Since many VNFs such as stateful firewalls [20] need to maintain flow states, it is more appropriate to process all the traffic of *one flow* in *one instance*.

Algorithm 1: Dynamic Scheduling for Latency Fulfillment

```
Input: Number of stages: n, s[x][i] initiated as INFINITE.
Output: Nodes along the shortest path P: node[n].
/* Iterate each stage from 1 to n and calculate s[x][i] */
foreach i ∈ 1 to n do
    foreach Node x ∈ Stage i do
        foreach Node y ∈ Stage i − 1 do
            /* Calculate s[x][i] through node y in stage i − 1. */
            if s[y][i − 1] + latency[x][y] < s[x][i] then
                s[x][i] = s[y][i − 1] + latency_t[x][y] + latency_p[x];
                pre[x][i] = y;
/* Last node in P is the node with the shortest latency in stage n. */
node[n] = arg min_{N_{x,n}} s[x][n];
/* Iterate backwards to find previous nodes. */
foreach i ∈ n − 1 to 1 do
    node[i] = pre[node[i + 1]][i];
```

V. Performance-Aware VNF Placement

For VNF placement, the orchestrator describes requirements of a new instance including functionality, performance and resource reservations. The mediator is therefore challenged to quickly find a suitable infrastructure type and location to satisfy all requirements. Therefore, we present a performance-aware VNF placement algorithm to address the following two major challenges.

First, the mediator should decide whether to place the VNF on hardware or software. Assume the new instance can be support by both hardware and software, i.e. both hardware and software versions of images of the same NF have been integrated into HYPER. As mentioned above, if software performance cannot fulfill SLA requirements, the mediator should pick hardware. Otherwise, we assume that software resources are relatively abundant and cheap compared with hardware [19] and place the instance on software.

Second, network dynamics can be frequent since network traffic volumes could vary significantly and affect VNF workloads. In HYPER, we perform the VNF placement in two major scenarios, including relatively *slow offline initialization* where could happen only occasionally, and *fast online VNF deployment*, which could be triggered frequently.

A. Offline Initialization

During network initialization, service providers need to distribute all VNF instances into hardware and software infrastructure. To reduce operational cost, we try to support all VNF instances with the minimal costs through the offline placement.

We pre-build a traffic model of each VNF instance, which has been well illustrated [21], [22] and is out of the scope of this paper. We use the model to estimate latency, throughput, and peak TCAM or CPU resource consumption of the instances, and decide whether to support them with hardware or software, as mentioned above. Placement for hardware and software are similar due to the nearly identical resource models. Thus, we mainly introduce the algorithm for hardware.

We formulate a bin packing problem [23] to minimize the overall costs to support all required VNF instances while

TABLE I
NOTATIONS USED IN VNF PLACEMENT ALGORITHM

H	set of hardware devices
M	set of VNF instances to place
λ_{mh}	=1 if device h can support instance m
α_m	required TCAM resource of instance $m \epsilon M$
ε_h	TCAM capacity of device $h \epsilon H$
ε_{hf}	free TCAM resource amount of device $h \epsilon H$
β_m	latency requirement of instance $m \epsilon M$
ζ_h	processing latency of device $h \epsilon H$
γ_m	throughput requirement of instance $m \epsilon M$
η_h	total throughput of device $h \epsilon H$
η_{hf}	remaining throughput of device $h \epsilon H$
c_h	overall costs (such as power) of device $h \epsilon H$
g_{mh}	Binary variable to designate VNF instance m was assigned to device h
ϕ_h	Binary value to designate hardware device h was occupied

meeting TCAM resource constraints and performance requirements of H-VNF instances (Table I presents the notations).

$$Min \sum_{h \epsilon H} c_h \cdot \phi_h \tag{1}$$

subject to

$$g_{mh} \leq \lambda_{mh}, \quad \forall m \epsilon M, \ h \epsilon H \tag{2}$$

$$\sum_{h \epsilon H} g_{mh} = 1, \quad \forall m \epsilon M \tag{3}$$

$$\sum_{m \epsilon M} g_{mh} \cdot \alpha_m \leq \varepsilon_h, \quad \forall h \epsilon H \tag{4}$$

$$\beta_m \geq \zeta_h, if g_{mh} = 1 \tag{5}$$

$$\sum_{m \epsilon M} g_{mh} \cdot \gamma_m \leq \eta_h, \quad \forall h \epsilon H \tag{6}$$

1) Objective Function: We want to minimize the costs to support all required VNF instances. Let the variable ϕ_h be set to denote device h is occupied where $h \epsilon H$. The objective function is described in Eq.1 and minimize the overall costs. *Cost* here refers to a weighed sum of all consumptions including power, CAPEX and OPEX. The weight on each factor can be assigned by the network operator.

2) Assignment Constraints: We set $\lambda_{mh} = 1$ if instance m from set M can be carried by device h (in terms of functionality). Let g_{mh} denote whether instance m is placed at device h. Eq.2 ensures that any module m is placed at a device that can support it. Eq.3 ensures that a given module m is placed once.

3) Resource Constraints: Each hardware device h has ε_h TCAM resources. α_m denotes an upper bound of TCAM resources that a instance requires. When multiple instances co-locate at the same device, Eq.4 assures that the device TCAM capacity is respected.

4) Performance Constraints: For each instance m, we pre-estimate the latency and throughput requirements. β_m is the upper bound of tolerated processing latency, and γ_m is the upper bound of required throughput. Eq.5 assures that the processing latency ζ_h can meet the loosest requirement of the instance β_m. Eq.6 assures that VNF instances placed at node h cannot exceed its throughput capacity η_h.

Offline initialization is not frequent compared to online VNF instance deployment. We prefer to endure relatively slow calculation and get the maximum optimization on resource utilization efficiency. We adopt the bin-completion solution [23] and achieve optimal placement with acceptable running time.

B. Online VNF Instance Deployment

For VNF migration or dynamic SLA fulfillment, the mediator needs to deploy a new instance. As mentioned above, we prefer software implementation of the new instance if software performance can fulfill SLAs. If not, we place the new instance on hardware. We seek to distribute the new instance to the most *suitable* infrastructure.

The baseline of traditional runtime VNF placement algorithms are Least Used Host placement algorithm and Least Busy Host algorithms [24]. In comparison, we define *suitability* by introducing a *greedy strategy*. A device is the most suitable if its performance is satisfactory and remaining resource *merely* meets the *peak* requirements of the new instance to minimize resource over-provisioning. Besides, our algorithm should quickly adapt to VNF deployment dynamics. Due to the similarity of hardware and software infrastructure, we only introduce the algorithm for hardware infrastructure. We denote the instance as mn, and define $fitness$ of each hardware node h as:

$$fitness(h) = (\alpha_{mn}/\varepsilon_{hf}) * (\gamma_m/\eta_{hf}), \quad \forall h \epsilon H$$

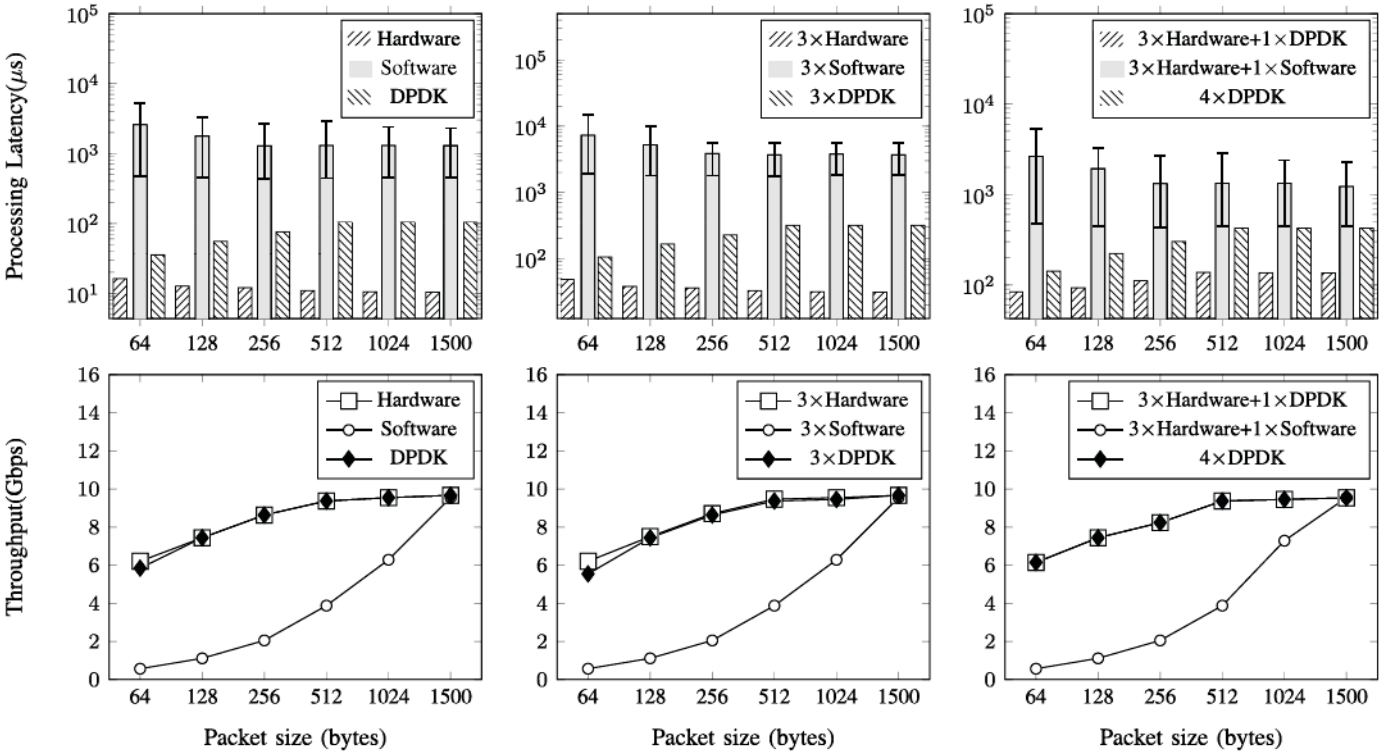
The resource-greedy algorithm has two major steps. First, we select hardware devices with adequate performance and resource as candidate set H_c for the new instance. A qualified device h in H_c should meet the following formulations.

$$\beta_{mn} \geq \zeta_h, \quad \alpha_{mn} \geq \varepsilon_h, \quad \gamma_{mn} \geq \eta_h, \quad \forall h \epsilon H_c$$

Then, we find the device from step 1 with the largest $fitness$ value to place the new VNF instance. In this way, the most suitable device is selected, to which the new instance will be deployed. The resource-greedy algorithm is relatively fast since it only has to iterate all nodes once to select the best fit node for the new instance. Suppose there are n nodes. The calculation time complexity would be $O(n)$, which is optimal and could adapt to quick network dynamics.

VI. IMPLEMENTATION

We selected ONetCard [16], a programmable hardware platform based on NetFPGA, as hardware resource and commodity servers as software resource. ONetCard is an acceleration card supporting two 10G network interfaces based on PCI Express. Its center is the FPGA device Kintex7 (XC7K325T-2), which connects network sub-system, storage sub-system, CPU connection sub-system, and inter-board sub-system. As the programmable center of the entire ONetCard developing board, the Xilinx Kintex7-325T FPGA provides



(a): Single stateful firewall instance (b): Hardware v.s. software chain (c): Hybrid v.s. software chain

Fig. 4. Performance of HYPER compared with pure software implementation with/without DPDK on forwarding latency, jitter and throughput. The marked line on each latency bar represents the maximum and minimum latency in that condition.

over 326 thousand logic cells. The TCAM resource on the board is simulated by Look Up Tables (LUTs) based on RAM on the ONetCard platform. ONetCard is equipped with richer resource than NetFPGA [25], which motivates us to use ONetCard as our hardware platform. On the ONetCard hardware platform, we implemented Stateful Data Plane Abstraction (SDPA) [20], an enhanced OpenFlow data plane that is simple yet powerful enough to support a variety of network functions. We extended SDPA to expose control interfaces to the HYPER mediator.

We implemented both hardware and software versions of several VNFs including NAT, stateful firewall, and elephant flow detection, and a software version of DPI. NAT monitors flow states to allocate an entry when a new connection starts and to deallocate it when it ends. A stateful firewall keeps tracking the states of network connections and determines packet handling according to associated state information [26]. We issued 100 static flow filtering rules to the stateful firewall. Elephant flow detection preserves counters for each flow, and alerts to the VNFM when any counter exceeds a certain threshold. For DPI, we assigned 100 regular expressions as patterns to match on packet payloads. The DPI alerts the VNFM when any specific pattern is matched.

We implemented VNFMs, HYPER mediator, and orchestrator based on OpenStack *(Newton version)* and Floodlight [27]. OpenStack serves as S-VNFM since it can create, start or terminate S-VNFs, and monitor resource usage of each instance. We extended Floodlight with an H-VNFM module. HYPER mediator is written as a module application in Floodlight (about 2.3K LoC), including the performance-aware VNF placement algorithm. We implemented the orchestrator as another module application in Floodlight and implemented an SLA-aware service chaining algorithm inside the orchestrator (about 1.5K LoC). For packet switching between hardware and software devices, we used Open vSwitch (OVS) [28]. We configured the forwarding rules in OVS to ensure the reachability between VNFs.

VII. Evaluation

To evaluate HYPER, we built a general hardware infrastructure based on ONetCard with two 10G NICs. We implemented software VNFs in Dell PowerEdge R730 servers with 40 Intel Xeon CPU (3.00GHZ), 256G memory, and two 10 Gigabit/s NIC managed by OpenStack. We constructed HYPER infrastructure based on two pieces of ONetCard boards and two commodity servers. We used Ixia tester to generate packets and measure the latency, jitter, and throughput.

A. Performance of HYPER service chain

We evaluated HYPER performance of a single VNF, a pure hardware service chain, and a hybrid service chain compared with the traditional software implementation based on VMs with or without DPDK enhancement.

1) Hardware Implementation v.s. Software Implementation of the Same VNF: To compare the performance of software and hardware implementations of the same VNF, we run a hardware stateful firewall instance and a software one. We used

Ixia tester to generate TCP packets of fixed sizes, and sent the same traffic to the two instances. Results in Fig. 4(a) shows that the processing latency of unoptimized software could be 119× to 154× higher than that of hardware, while DPDK enhanced software still suffers an average of 5.4× higher latency than hardware. The throughput of unoptimized software is 91% lower than hardware, while DPDK enhanced software could achieve as high throughput as hardware. The latency jitter of DPDK is lower than bare software but is still high compared with stable hardware. The huge performance gap demonstrates the benefit of integrating hardware into the NFV to achieve high performance.

2) Hardware Chain v.s. Software Chain: We implemented stateful firewall, NAT, and elephant flow detection on hardware, and compared the performance of this hardware chain with a software chain composed of the same sequence of VNFs with or without DPDK enhancement. As shown in Fig. 4(b), the hardware chain outperforms the software chain without DPDK on forwarding latency, latency jitter and throughput. The DPDK enhanced software service chain can achieve almost the same throughput performance as hardware. However, its latency is still much higher than hardware and latency jitter remains significant, which could be unacceptable for some latency-sensitive applications.

3) Hybrid Chain v.s. Software Chain: We further evaluated HYPER through forming a hybrid chain composed of both hardware and software VNF instances. We implemented stateful firewall, NAT, and elephant flow detection on hardware, and DPI on software with or without DPDK, and compare their performance with a pure DPDK implementation. Experimental results are shown in Fig. 4(c). A hybrid service chain composed of hardware and DPDK enhanced software outperforms the pure DPDK solution on both processing latency (with a decrease of 40% to 67%) and latency jitter. The results further demonstrate the forwarding latency performance is boosted by integrating hardware into the NFV network.

B. Efficiency of H-VNF Deployment

To evaluate the efficiency of VNF deployment, we deployed a hardware instance of stateful firewall VNF. During initialization, the mediator installs the processing logic and relevant table entries into the hardware instance through `deploy_instance` and `configure_instance` control interfaces. We designed 16 table entries that records the state transition rules and actions under different TCP states. According to the statistics, a hardware instance of our stateful firewall can be quickly deployed within $6.1ms$ by installing the configurations (i.e. 16 table entries) into the hardware, demonstrating the flexibility and efficiency of the deployment of hardware instances, which could happen frequently in dynamic NFV networks. This further proves the feasibility of integrating hardware into NFV while maintaining flexibility.

C. Performance of H-VNF Migration

The integration of hardware infrastructure in HYPER creates the necessity of H-VNF migration, which is implemented by transferring all table entries from the old instance to the new one in SDPA. To evaluate the migration performance, we deployed two hardware stateful firewall instances and issued commands to the mediator to migrate states in one instance to the other one. Here states refer to flow-level TCP states in stateful firewalls. Results show the mediator can migrate approximately $9,000$ flows per second. Therefore, regardless of the time for new instance deployment, a instance containing 30,000 state records can be migrated to the new instance in $3.3s$, which is fast and can adapt to network dynamics.

VNF migration is a relatively common phenomenon in a NFV-based network. Thus, we measured the extra traffic introduced by VNF migration to prove that HYPER can meet such frequent dynamics. For a stateful firewall, it takes 31 bytes to store a flow and 1 byte to store the TCP state of this flow. Thus, it takes totally 32 bytes to represent a flow and its state in the table. To migrate 100 thousand table entries, it introduces a total of 3.2 MB extra traffic to the network, which is rather trivial and acceptable.

D. Performance of VNF Placement Algorithm

In order to demonstrate the benefit of performance-aware VNF placement algorithm, we compared it with a strawman method that uses First-Fit Decrease Algorithm (FFD). FFD is a straightforward greedy approximation algorithm. For each instance, we enhance FFD to place the instance in the first node that can accommodate the instance, with respect to both performance and resource, which already is an extension compared with traditional resource-only placement. If no node is found, FFD opens a new node and puts the instance within the new node. As different hardware devices vary in capacity, performance and cost, we assume that all hardware devices share the same parameters. We compared the two algorithms using the following metric: the number of occupied nodes when accommodating the same set of VNF instances. As shown in Figure. 5, our algorithm saves approximately 5% devices compared with the FFD algorithm. In a large NFV network that runs hundreds of instances [29], the performance-aware VNF placement algorithm could save a large number of devices, and as a result save CAPEX.

E. Performance of Service Chaining Algorithm

We implemented 20 VNF instances in the network. Since currently there are no service chaining algorithms designed to meet tenant performance requirements, we implemented a strawman method of only taking into account tenants' function requirements and device resource constraints, like an algorithm introduced in Stratos [13]. Note that to measure the performance of the algorithm, we assume the VNF can be deployed immediately when the algorithm issues the deployment requests. We generated 10,000 random SLAs of functions, forwarding latency and throughput. The average calculation time of SLA-aware algorithm is $1.2ms$, while the strawman solution achieves the average time of $1.0ms$. The increase of calculation latency is incurred by fulfilling performance requirements. However, the increased calculation

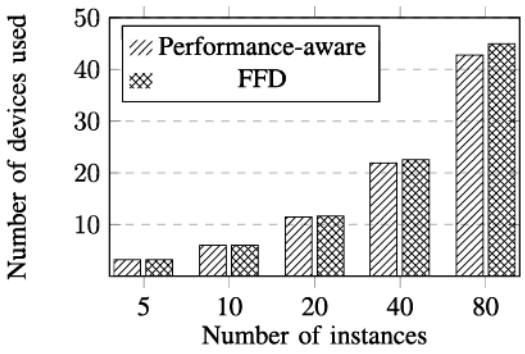


Fig. 5. Number of devices to accommodate all instances in performance-aware VNF placement algorithm.

time of $0.2ms$ is acceptable and the algorithm runs quickly enough to fulfill massive SLAs.

VIII. Related Work

A. Hybrid Infrastructure

Some recent work recognized the benefits of combining hardware and software to achieve both high performance and flexibility of NFV [4], [30]–[32]. Gandhi *et al.* [4] proposed DUET, which used existing switch hardware and a small deployment of software to build high performance load balancers. Nevertheless, DUET was restricted to the load balancing VNF. Moens and De Turck [30] proposed VNF-P to support VNFs on dedicated hardware and general software. However, dedicated hardware is still limited in flexibility for VNF management. Putnam *et al.* [31] and Byma *et al.* [32] proposed to extend servers in data centers with FPGAs to enhance the performance of NFV. In contrast, HYPER can support a variety of VNFs based on state of the art programmable hardware and software platforms, and introduces a SLA-aware service chaining algorithm to provide better quality of service.

B. VNF Placement

Some recent work also proposed VNF placement algorithms [29], [30], [33], [34]. CoMB [29] proposed to consolidate software-based middleboxes for easier service chain management. The objective of CoMB placement algorithm was to minimize the maximum load of each middlebox, as in most of other works. However, HYPER integrates hardware and software to improve overall performance, while the VNF placement algorithm maximizes resource utilization efficiency with respect to both performance and resource requirements of VNFs instances. These two algorithms can optimize VNF placement in orthogonal directions. Moreover, extended CoMB with unified interfaces can be integrated in HYPER. Luizelli *et al.* [34] also solved the VNF placement problem using ILP. However, HYPER performs VNF placement on the hybrid infrastructure and builds performance and resource models for hardware and software NFs to guide the placement of NFs.

Service Chaining: Halpernand Pignataro [12], Gember *et al.* [13], Fayazbakhsh *et al.* [14], and Qazi *et al.* [15] proposed service chaining algorithms that supported dynamic ordering of service functions. However, they only chained network functions and considered load balancing between instances, while we propose an SLA-aware service chaining algorithm that also considers tenants' performance requirements. Sahhaf *et al.* [35] and Luizelli *et al.* [34] proposed a service chaining mechanism that covers the performance requirements from tenants. However, it focuses on minimizing the service decomposition mapping cost, while HYPER targets on the chaining of network functions on a hybrid infrastructure.

IX. Discussion

HYPER integrates both software and hardware infrastructures to achieve both high performance and flexibility. Therefore, if a VNF can be both implemented on hardware and software, HYPER is challenged to pick the most appropriate infrastructure type to support the VNF. We have demonstrated that hardware-based VNFs outperforms software-based VNFs in both latency and throughput. However, commodity servers, compared with advanced programmable hardware devices such as ONetCard, are relatively cheaper and more flexible [19]. Therefore, to decide where to place an VNF, we follow the following two principles, as introduced in this paper. First, we consider satisfying SLA requirements of tenants as our first class objective. Second, if software could provide adequate performance, we select software to accommodate the VNF. Otherwise, we choose the hardware implementation.

With the development and popularity of programmable hardware, such as Barefoot Tofino [36], hardware devices might grow cheaper and more flexible. However, our insight on programmable hardware is that it has to trade a certain amount of programmability and flexiblity for performance. Complex VNFs still need to be implemented based on software, which leads to a hybrid infrastructure situation, as described in HYPER. However, with the improvement of both software and hardware techniques, we may need to adjust infrastructure type selection principles and workload distribution among them.

X. Conclusion

In this paper, we have proposed a novel hybrid high-performance framework, HYPER, for NFV, where network functions can be effectively supported through the integration of hardware and software infrastructures. We have designed unified control interfaces to support more flexible and efficient VNF management. We have also introduced a performance-aware VNF placement algorithm for the management of infrastructure resources and a SLA-aware service chaining algorithm to fulfill tenant SLAs. Our experimental results have demonstrated that our solution can significantly improve service chain performance while maintaining the flexibility of NFV. In the future, we will integrate more platforms such as P4 into HYPER to demonstrate its feasibility and efficiency.

REFERENCES

[1] R. Guerzoni *et al.*, "Network functions virtualisation: An introduction, benefits, enablers, challenges and call for action, introductory white paper," in *Proc. SDN OpenFlow World Congr.*, Jun. 2012, pp. 5–7.

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

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

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

[5] DIntel. *Data Plane Development Kit*. Accessed: Sep. 12, 2017. [Online]. Available: http://dpdk.org

[6] X. Ge *et al.*, "OpenANFV: Accelerating network function virtualization with a consolidated framework in openstack," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 353–354, 2014.

[7] C. Kachris, G. Sirakoulis, and D. Soudris. (2014). "Network function virtualization based on FPGAs: A framework for all-programmable network devices." [Online]. Available: https://arxiv.org/abs/1406.0309

[8] G. Brebner, "Softly defined networking," in *Proc. 8th ACM/IEEE Symp. Archit. Netw. Commun. Syst.*, 2012, pp. 1–2.

[9] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014.

[10] P. Bosshart *et al.*, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," in *Proc. ACM SIGCOMM Conf. (SIGCOMM)*, 2013, pp. 99–110.

[11] Q. Maqbool, S. Ayub, J. Zulfiqar, and A. Shafi, "Virtual TCAM for data center switches," in *Proc. IEEE Conf. Netw. Function Vis. Softw. Defined Netw. (NFV-SDN)*, 2015, pp. 61–66.

[12] J. Halpern and C. Pignataro, *Service Function Chaining (SFC) Architecture*, (Work Progress), document RFC 7665, document draft-ietf-sfc-architecture-07, 2015.

[13] A. Gember *et al.* (2013). "Stratos: A network-aware orchestration layer for virtual middleboxes in clouds." [Online]. Available: https://arxiv.org/abs/1305.0209

[14] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags," in *Proc. USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2014, pp. 533–546.

[15] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "SIMPLE-fying middlebox policy enforcement using SDN," in *Proc. ACM SIGCOMM Conf. (SIGCOMM)*, 2013, pp. 27–38.

[16] *ONetCard*. Accessed: Sep. 12, 2017. [Online]. Available: http://onetswitch.org/onetcard.html

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

[18] NETSI. (2014). *ETSI Network Functions Virtualisation (NFV) Industry Standards (ISG) Group Draft Specifications*. [Online]. Available: http://docbox.etsi.org/ISG/NFV/Open

[19] J. Ellerton *et al.*, "Prospects for software defined networking and network function virtualization in media and broadcast," in *Proc. Annu. Tech. Conf. Exhib.*, Oct. 2015, pp. 1–21.

[20] S. Zhu, J. Bi, C. Sun, C. Wu, and H. Hu, "SDPA: Enhancing stateful forwarding for software-defined networking," in *Proc. 23rd IEEE Int. Conf. Netw. Protocols (ICNP)*, Nov. 2015, pp. 323–333.

[21] C. F. Daganzo, "The cell transmission model, part II: Network traffic," *Transp. Res. B, Methodol.*, vol. 29, no. 2, pp. 79–93, 1995.

[22] R. Jain and S. Routhier, "Packet trains–measurements and a new model for computer network traffic," *IEEE J. Sel. Areas Commun.*, vol. SAC-4, no. 6, pp. 986–995, Sep. 1986.

[23] R. E. Korf, "A new algorithm for optimal bin packing," in *Proc. AAAI/IAAI*, 2002, pp. 731–736.

[24] S. Clayman, E. Maini, A. Galis, A. Manzalini, and N. Mazzocca, "The dynamic placement of virtual network functions," in *Proc. IEEE Netw. Oper. Manage. Symp. (NOMS)*, May 2014, pp. 1–9.

[25] *NetFPGA*. Accessed: Sep. 12, 2017. [Online]. Available: http://netfpga.org/

[26] C. Roeckl and C. M. Director, "Stateful inspection firewalls," Juniper Networks, Sunnyvale, CA, USA, White Paper, 2004.

[27] *Project Floodlight*. Accessed: Sep. 12, 2017. [Online]. Available: http://www.projectfloodlight.org/floodlight/

[28] *Open vSwitch*. Accessed: Sep. 12, 2017. [Online]. Available: http://openvswitch.org/

[29] 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.*, Aug. 2012, p. 24.

[30] H. Moens and F. De Turck, "VNF-P: A model for efficient placement of virtualized network functions," in *Proc. 10th Int. Conf. Netw. Service Manage. (CNSM)*, Nov. 2014, pp. 418–423.

[31] A. Putnam *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proc. ACM/IEEE 41st Int. Symp. Comput. Archit. (ISCA)*, Jun. 2014, pp. 13–24.

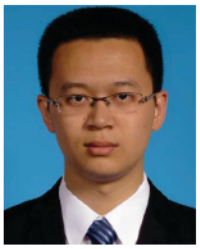
[32] S. Byma, J. G. Steffan, H. Bannazadeh, A. Leon-Garcia, and P. Chow, "FPGAs in the cloud: Booting virtualized hardware accelerators with OpenStack," in *Proc. IEEE 22nd Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, May 2014, pp. 109–116.

[33] X. Li and C. Qian, "A survey of network function placement," in *Proc. 13th IEEE Annu. Consum. Commun. Netw. Conf. (CCNC)*, Jan. 2016, pp. 948–953.

[34] 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.

[35] S. Sahhaf, W. Tavernier, D. Colle, and M. Pickavet, "Network service chaining with efficient network function mapping based on service decompositions," in *Proc. 1st IEEE Conf. Netw. Softw. (NetSoft)*, Apr. 2015, pp. 1–5.

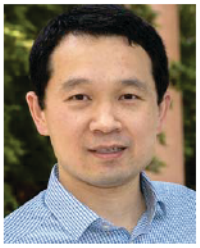
[36] H. Stubbe, "P4 compiler & interpreter: A survey," in *Proc. Future Internet (FI) Innov. Internet Technol. Mobile Commun. (IITM)*, vol. 47. 2017, pp. 1–72.



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

Jun Bi (S'98–A'99–M'00–SM'14) received the B.S., C.S., and Ph.D. degrees from the Department of Computer Science, Tsinghua University, Beijing, China. He is currently a Changjiang Scholar Distinguished Professor with Tsinghua University and the Director of the Network Architecture Research Division, Institute for Network Sciences and Cyberspace, Tsinghua University. His current research interests include Internet architecture, SDN/NFV, and network security. He is a Distinguished Member of the China Computer Federation. He successfully led tens of research projects, published more than 200 research papers and 20 Internet RFCs or drafts, owned 30 innovation patents, received national science and technology advancement prizes, the IEEE ICCCN Outstanding Leadership Award, and best paper awards. He is the Co-Chair of the AsiaFI (Asia Future Internet Forum) Steering Group and the Chair of the China SDN Experts Committee. He served as the TPC Co-Chair a number of future Internet related conferences or workshops/tracks at INFOCOM, ICNP, and so on. He served on Organization Committee or Technical Program Committees of SIGCOMM, ICNP, INFOCOM, CoNEXT, SOSR, and so on.

Zhilong Zheng received the B.S. degree from the Department of Software Engineering, Chongqing University, Chongqing, China, in 2016. He is currently pursuing the Ph.D. degree with the Institute for Network Sciences and Cyberspace, Tsinghua University. His research interests include software-defined networking and network functions virtualization.



Hongxin Hu (S'10–M'12) received the Ph.D. degree in computer science from Arizona State University, Tempe, AZ, USA, in 2012. He is an Assistant Professor with the Division of Computer Science, School of Computing, Clemson University. He has authored or co-authored over 80-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, security and privacy in social networks, and security in cloud and mobile computing. He was a recipient of the Best Paper Award from ACM CODASPY 2014, and the Best Paper Award Honorable Mentions from ACM SACMAT 2016, the IEEE ICNP 2015, and ACM SACMAT 2011. His research has been funded by National Science Foundation, U.S. Department of Transportation, VMware, Amazon, and Dell. He has served as a Technical Program Committee Member for many conferences, such as ACM Symposium on Information, Computer and Communications Security, the IEEE Conference on Communications and Network Security, and the ACM Symposium on Access Control Models and Technologies.