Xu TN, Sun HF, Zhang D *et al.* NfvInsight: A framework for automatically deploying and benchmarking VNF chains. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 37(3): 680–698 May 2022. DOI 10.1007/s11390-020-0434-1

NfvInsight: A Framework for Automatically Deploying and Benchmarking VNF Chains

Tian-Ni Xu[1,2] (徐天妮), Hai-Feng Sun[1,2] (孙海锋), Di Zhang[1,2] (张 笛), Xiao-Ming Zhou[1,2] (周小明)
Xiu-Feng Sui[3] (隋秀峰), Sa Wang[1,2,4] (王 卅), *Member, CCF, ACM*
Qun Huang[5] (黄 群), *Member, CCF, ACM, IEEE*, and
Yun-Gang Bao[1,2,4,*] (包云岗), *Senior Member, CCF, Member, ACM, IEEE*

[1] *State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences Beijing 100190, China*

[2] *University of Chinese Academy of Sciences, Beijing 100049, China*

[3] *School of Information and Electronics, Beijing Institute of Technology, Beijing 100081, China*

[4] *Peng Cheng Laboratory, Shenzhen 518055, China*

[5] *Department of Computer Science and Technology, Peking University, Beijing 100871, China*

E-mail: xutianni@ict.ac.cn; sunhaifeng@stu.pku.edu.cn; zhangdi@ict.ac.cn; zhouxiaoming@ict.ac.cn
suixiufeng@bit.edu.cn; wangsa@ict.ac.cn; huangqun@pku.edu.cn; baoyg@ict.ac.cn

Received March 10, 2020; accepted October 15, 2020.

Abstract With the advent of virtualization techniques and software-defined networking (SDN), network function virtualization (NFV) shifts network functions (NFs) from hardware implementations to software appliances, between which exists a performance gap. How to narrow the gap is an essential issue of current NFV research. However, the cumbersomeness of deployment, the water pipe effect of virtual network function (VNF) chains, and the complexity of the system software stack together make it tough to figure out the cause of low performance in the NFV system. To pinpoint the NFV system performance issues, we propose NfvInsight, a framework for automatic deployment and benchmarking VNF chains. Our framework tackles the challenges in NFV performance analysis. The framework components include chain graph generation, automatic deployment, and fine granularity measurement. The design and implementation of each component have their advantages. To the best of our knowledge, we make the first attempt to collect rules forming a knowledge base for generating reasonable chain graphs. NfvInsight deploys the generated chain graphs automatically, which frees the network operators from executing at least 391 lines of bash commands for a single test. To diagnose the performance bottleneck, NfvInsight collects metrics from multiple layers of the software stack. Specifically, we collect the network stack latency distribution ingeniously, introducing only less than 2.2% overhead. We showcase the convenience and usability of NfvInsight in finding bottlenecks for both VNF chains and the underlying system. Leveraging our framework, we find several design flaws of the network stack, which are unsuitable for packet forwarding inside one single server under the NFV circumstance. Our optimization for these flaws gains at most 3x performance improvement.

Keywords network function virtualization (NFV), service chain, performance bottleneck, network stack latency

1 Introduction

Modern enterprises or data centers rely on a variety of data plane appliances to enhance their networking infrastructures. These appliances, also referred to as network functions (NFs), offer valuable benefits, ranging from improving security and boosting performance to reducing resource usage. Traditionally, each type of

Regular Paper

This work was supported by the National Key Research and Development Program of China under Grant No. 2019YFB1802600, the National Natural Science Foundation of China under Grant Nos. 61420106013, 61702480, 61672499, and 61802365, the Youth Innovation Promotion Association of Chinese Academy of Sciences under Grant Nos. 2013073 and 2020105, and the Guangdong Province Key Laboratory of Popular High Performance Computers under Grant No. 2017B030314073.

*Corresponding Author

©Institute of Computing Technology, Chinese Academy of Sciences 2022

NFs is built in proprietary middleboxes with specialized hardware, which provides great performance but falls short in flexibility and also incurs high capital and management costs. The emergence of network function virtualization (NFV) breaks down this barrier by shifting the NFs to the off-the-shelf hardware. NFV encapsulates NFs into virtualized appliances (e.g., virtual machines or containers) forming virtualized network functions (VNFs), and runs them atop commodity hardware. Thus, NFV significantly alleviates the investment in hardware and brings great flexibility in both developing and deploying.

However, NFV inevitably suffers extra performance penalty (e.g., performance interference and extra I/O overhead) brought by virtualization techniques. The state-of-art work puts a lot of effort into improving the NFV performance. NetVM [1] and OpenNetVM [2] provide an optimized hypervisor layer utilizing Intel DPDK for NFV traffic steering, which bypasses the kernel network stack to boost performance. OpenNF [3], OpenBox [4], Metron [5] and NFP [6] provide new programming models of VNFs to shorten user-level packet processing paths. Other studies leverage GPU [7,8], FPGA [9] and SmartNIC [8] to accelerate VNFs. Despite all these studies, how to explore and pinpoint the performance bottleneck or even the root cause of a performance issue in an NFV system remains an open and challenging question. It is tough to figure out the cause of low performance. Several reasons are making it effort-taken.

1) The deployment of the NFV system is cumbersome. It involves configurations and setups on multiple layers, which includes hypervisor, network, kernel, and VNF specified parameters. When doing performance debugging, iterative tests and repeated changing of configurations are necessary but tedious, and cumbersome without high automation.

2) The "water pipe" effect of the forwarding path makes it even difficult. The forwarding path, which includes each VNF, virtual switches and the critical functions of the network stack, forms a water pipe. This effect is particularly prominent when it comes to TCP, because of its ACK-based flow control mechanism. The end-to-end bandwidth is determined by the narrowest point of the path. Thus, it is impossible to locate the narrowest point if the measurement results contain only the end-to-end performance. Even the throughput or latency of each VNF is still not enough. The former cannot indicate the bottleneck because of the water pipe effect, and the latter ignores the impact of virtual switches and the network stack.

3) The shared system is hard to analyze. Under the light-weighted virtualization environment, system modules are shared among all virtualized instances. Though many system modules are not on the packet forwarding path, they still influence the performance. The performance factors include, but are not limited to, softirq, locks in kernel, and CPU scheduling mechanism.

To efficiently discover the NFV performance bottleneck, we propose a distributed framework named NfvInsight. It takes the VNFs specified by the network operator as input and enumerates all possible chain topologies. The framework then automatically deploys VNFs and steers the network traffic flowing through the chain. Meanwhile, the framework collects measurement metrics in fine granularity, covering the whole packet forwarding path and multiple levels of the system.

To address the performance issues of NFV chains, NfvInsight takes a comprehensive benchmarking methodoloy. The performance issues are all reflected in the actual tests and measurement results. Moreover, the comprehensiveness reflects in two ways. One is the actual testing of all possible chain sequences. The other one is the fine granularity measurement which covers multiple metrics for detecting the performance bottlenecks.

Leveraging this framework, we analyze five typical VNFs and the chains they form. We find that there are differences in the performance of VNFs themselves, and their scale-out performance is distinct too. For the typical scenario of the NFV system, in which multiple VNFs communicate inside a physical machine, we analyze the fine-grained measurement results obtained by NfvInsight and find that several designs in the network stack are not highly efficient for internal packet transferring. 1) SR-IOV does not have a NIC bypass mechanism. 2) UDP slicing is time-consuming and unnecessary. 3) Softirq is CPU-consuming and not balanced on cores. We make simple optimizations of the above three problems. Each modification involves less than 200 lines of code, and we obtain at most three times performance gaining.

To summarize, we make contributions in the following three aspects.

• We implement a prototype that automatically generates chain graphs and configurations, deploys VNFs and evaluates performance with little human involvement, exactly, one-click deployment and benchmarking.

• We perform case studies to showcase the convenience and usability of NfvInsight in finding bottlenecks for both VNF chains and the system.

• NfvInsight helps find several design flaws inside the network stack for packets forwarding inside one single server in the NFV circumstance. Our optimization of the network stack gains at most 3x performance improvement.

2 Challenges

Network operators chain VNFs into a directed acyclic graph (DAG), which is referred to as a "service chain". Nodes in the DAG represent VNFs, and the directional edges represent the data path between two VNFs. The network traffic is steered to travel along the DAG and processed by the VNFs in the path based on their specific purposes. There are at least three challenges to build a framework for automatically deploying and benchmarking VNF service chains.

1) *Chain Graph Generation.* The sequences of the chain mostly are determined based on experiences. There are not any must-obey rules, but only case suggestions in white paper documents [10]. However, according to our knowledge, the sequence influences the chain performance in at least two ways. One intuitive case is that some VNFs change the traffic volume of flows [11]. For example, the WAN Optimizer reduces traffic volume, while the BCH encoder increases it. Thus, the VNFs' position influences the traffic volume processed by the subsequent VNFs, and further influences the performance of the entire chain. Another non-intuitive case is presented in Probius [12]. Though irrelative to traffic volume, locating the slowest VNF at the beginning controls the I/O contentions of the subsequent VNFs, and it results in the highest performance among all possible chain graphs. Thus, possible sequences of VNFs in a chain should all be considered.

The sequence of different types of VNF needs to be taken into account. Counting multiple software implementations, naively enumerating all possible chains encounters the problem of graph explosion.

2) *Automatic Deployment.* The deployment of VNFs in enterprise networks ranges from tens to hundreds in scale [13]. Setting up an experimental environment at this scale is not easy. Our survey of top conference authors who have done NFV research work, shows that: despite the differences in proficiency, the deployment work requires at least two weeks of labor, and one month on average. According to our own practice, 391 lines of bash commands have to be executed to deploy and benchmark a chain, which is composed of five VNFs, on a distributed cluster having three servers. Even for an experienced network operator, we suppose it time-consuming and error-prone. Though there are industrial open projects for NFV management and orchestration①②, they lack the flexibility to do iterative measurements, which includes changing network configurations, allocating different resources for each VNF and trying multiple VNF configurations. Thus, flexible configuration and automation are needed.

However, automation does not mean merely putting all the bash commands together, and several issues need to be solved, which include transferring chain topology representation to deployment commands, defining practical configuration interfaces, and enabling iterative measurement of multiple graphs and multiple configurations.

3) *Fine Granularity Measurement.* NFV is proposed to reduce costs by utilizing commodity hardware. Therefore, it is common to consolidate multiple heterogeneous VNFs on one single physical machine for resource efficiency. It leads to a typical scenario that packets transferring inside a server. Only a fine-grained tracking of the packet forwarding path achieves in-depth analysis. Existing profiling tools are too general to track a packet's latency on the forwarding path, and show the disadvantage of unacceptable overhead delay. Moreover, multi-dimensional metrics of each layer of the system are also important to profile the system shared by multiple VNFs.

Collecting as many metrics as possible is the basic requirement, and on this basis, achieving both fine granularity and low overhead is challenging.

3 NfvInsight Framework

To efficiently discover the performance bottleneck of an NFV system, we propose the framework NfvInsight. The working process of NfvInsight is depicted in Fig.1. NfvInsight has quite a simple interface for network operators. The network operator only needs to specify the VNFs to be tested, and then check and modify the configuration file. With a one-click running, the specified chain will run and performance reports will be generated. There are three main phases of NfvInsight. Each

①https://www.opnfv.org/, Aug. 2020.

②https://osm.etsi.org/, Aug. 2020.

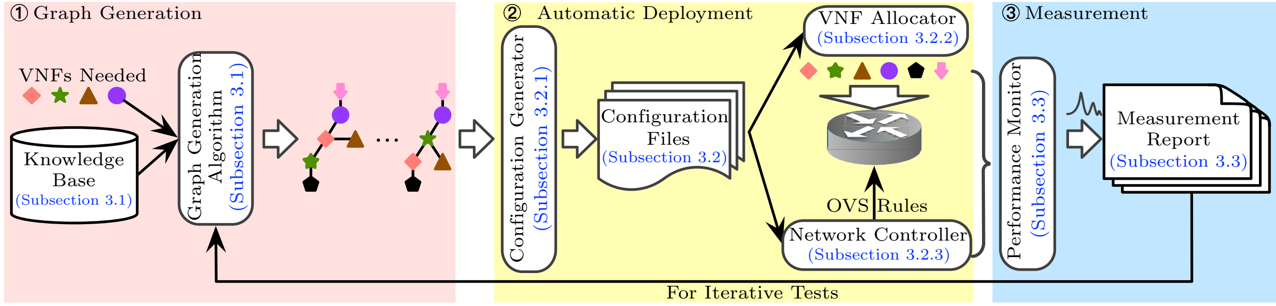


Fig.1. NfvInsight working process.

of them tackles one challenge elaborated in Section 2.

1) *Graph Generation.* In this phase, the graph generation algorithm enumerates service chains composed of the VNFs demanded by network operators. The knowledge base contains external information that can be used to optimize the enumeration. (Subsection 3.1)

2) *Automatic Deployment.* The configuration generator converts the chain graphs expressions into configuration files, which are utilized by the VNF allocator and network controller to automatically deploy the service chains. (Subsection 3.2)

3) *Fine Granularity Measurement.* The performance monitor collects multiple performance metrics and forms measurement report as output. The benchmark results are returned to network operators for iterative tests. Network operators examine the results to diagnose performance bottlenecks or further extend for their own purposes. (Subsection 3.3)

3.1 Graph Generation

Recall that a service chain is a directed acyclic graph, where a node is a VNF and an edge indicates a connection directed from one VNF to another. The graph generation algorithm is to construct all valid graphs composed of VNFs demanded by network operators. The algorithm outputs a set of expressions which indicate different sequences the given VNFs can form.

Logically, the algorithm is composed of two parts: one for generating graphs and the other for filtering invalid ones. However, the naive generate-and-filter design will cause the graph explosion problem, which means that the huge number of candidate graphs exhausts computational resources. This motivates us to integrate the two logical parts.

Graph Explosion Problem. We examine the graph explosion problem with an example of four VNFs. We first consider the number of possible topologies regardless of the mapping from VNFs to nodes. Fig.2 shows that there are four non-isomorphic topologies. Then, if we associate specific VNFs with the nodes, the resulting number further increases. Specifically, there are 4!=24 possible mappings in Figs.2(a) and 2(b). Fig.2(c) has 12 possible mappings, while Fig.2(d) has 4. Thus, the total number is 64.

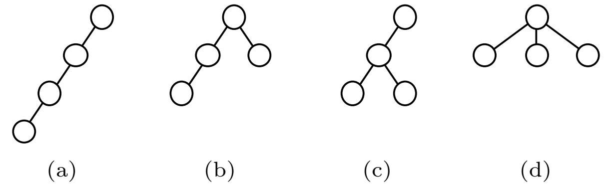


Fig.2. Four non-isomorphic topologies with four nodes.

In general, the number of resulting graphs dramatically increases with the number of VNFs n. To estimate the number, we start with a scenario that allows isomorphism. For the number of topologies, we consider a simplified case in which all graphs are binary trees. In this case, the number of the binary tree can be calculated as Catalan number [14], which is $\frac{(2n)!}{(n+1)!n!}$. Since a node has an arbitrary degree, the actual number of topologies is much larger than $\frac{(2n)!}{(n+1)!n!}$. When n VNFs are associated with the nodes, there are $n!$ possible mappings. Thus, the total number of resulting graphs is at least $\frac{(2n)!}{(n+1)!}$. Even though only non-isomorphism graphs are reserved, the number remains the same magnitude of $\frac{(2n)!}{(n+1)!}$.

Pruning Rules. To deal with graph explosion, we need to prune invalid service chains in the process of graph enumeration. This implies that the pruning rules highly hinge on the enumeration approach. To this end, we construct a graph by incrementally adding VNFs. When a new VNF is added, we examine whether it can be connected to any existing VNFs. Therefore, all pruning rules have the same form in our design. Specifi-

cally, each rule takes two VNFs (denoted by VNF_1 and VNF_2, respectively) and indicates whether the traffic can go from VNF_1 to VNF_2.

Algorithms. Algorithm 1 elaborates on how to integrate the topology enumeration and pruning rules. The algorithm takes a set of VNFs as input and produces all possible graphs denoted by G. It maintains two sets *picked* and *remain*. In particular, *picked* represents the VNFs added to the current graph, and *remain* represents the VNFs left.

Algorithm 1. VNF Chain Graph Generation

```
Input: VNFs in demand
Output: possible graphs G
 1: G = ∅
 2: picked = ∅
 3: remain = a set of all the VNFs
 4: function GRAPHGEN(picked, remain, G)
 5:     if remain == NULL then
 6:         Add this chain graph to G
 7:         return
 8:     for child in remain do
 9:         for parent in picked do
10:             if PARENTJUDGE(parent, child) then
11:                 Add child after parent
12:                 picked := picked + child
13:                 remain := remain − child
14:                 GRAPHGEN(picked, remain, G)
15: function PARENTJUDGE(parent, child)
16:     for rule in ruleList do
17:         if not obey rule then
18:             return False
19:     return True
```

The function GRAPHGEN generates the graph recursively. When *remain* is NULL, the iteration ends. In this case, a new graph is generated and added to the set G (lines 5–7). Otherwise, if *remain* is not NULL, we enumerate all possible connections between VNFs in *picked* and VNFs in *remain* (lines 8 and 9). The function PARENTJUDGE is called to judge whether each pair of VNFs can be connected (line 10). If this pair of VNFs obeys every pruning rule (lines 15–19), the subroutine PARENTJUDGE returns True. In this case, the new VNF in *remain* will be added to the current graph (lines 11–13). Then the next recursion is performed (line 14).

Knowledge Base. The knowledge base provides two types of information for pruning rules. First, it associates each VNF demanded by network operators with a category. Second, for any pair of categories, the knowledge base indicates whether they can be connected. Thus, when we judge whether there can be an edge between two specific VNFs, we first infer their categories and then judge the connection between the two corresponding categories.

We build the knowledge base based on well-recognized existing studies (e.g., [6, 15]) and our practical deployment experiences. Currently, we define four categories. The VNF types of each category and the typical appliances are summarized in Table 1.

Table 1. Commonly Used VNFs

Category	VNF	Appliance
Leave-node	IDS	Snort③
	Monitor	NetFlow④
	DNS	BIND⑤
Network-oriented	NAT	iptables⑥
	Gateway	Cisco MGX⑦
	WAN Opt.	Wanos⑧
Host-oriented	L4 Load Balancer	HAProxy⑨
General	L7 Cache	Squid⑩
	VPN	OpenVPN⑪
	Proxy	envoy⑫
	Compression	Cisco IOS⑬
	Traffic Shaper	Linux TC [16]
	IP Firewall	iptables⑥
	Anti-virus	McAfee⑭

③https://www.snort.org/, Aug. 2020.
④https://www.manageengine.com/products/netflow/, Aug. 2020.
⑤https://www.isc.org/bind/, Aug. 2020.
⑥https://netfilter.org/projects/iptables/, Aug. 2020.
⑦https://www.cisco.com/c/en/us/products/switches/mgx-8800-series-switches/index.html, Aug. 2020.
⑧http://wanos.co/wan-optimization/, Aug. 2020.
⑨https://www.haproxy.org/, Aug. 2020.
⑩http://www.squid-cache.org/, Aug. 2020.
⑪https://openvpn.net/, Aug. 2020.
⑫https://www.envoyproxy.io/, Aug. 2020.
⑬https://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-technologies/index.html, Aug. 2020.
⑭https://www.mcafee.com/en-us/index.html, Aug. 2020.

• Network-oriented VNFs indicate those at the entrance of a data center network (e.g., NAT).

• Host-oriented VNFs are those located right near the host. Taking load balance as an example, to reduce repeated processes, the actions of other VNFs are better done before dispatching requests to end hosts.

• Leave-node VNFs are those without transferring packets to the next hop such as IDS and monitor.

• General VNFs' positions can be changed freely.

Based on the four categories, we define three pruning rules.

1) A network-oriented VNF is the entrance of a graph.

2) A host-oriented VNF should locate the nearest to the server.

3) A leave VNF can be paralleled with other VNFs.

We take five VNFs in Table 1 as an example to elaborate the pruning rules. The result of our graph generation algorithm is shown in Fig.3.

The VNFs are IDS, NAT, L4 Load Balancer, L7 Cache and IP Firewall, which cover all the four categories we defined. To chain these five VNFs, pruning rules 1 and 2 stipulate that NAT locates at the beginning as the entrance of the chain, and L4 Load Balancer is the last in the chain connecting to the server directly. L7 Cache and IP Firewall locate between NAT and L4 Load Balancer, and the sequence of them can be changed. The leave VNF IDS can be connected to any other VNF. Thus, we get eight graphs as a result, instead of $\frac{(2\times5)!}{(5+1)!} = 5\,040$ without pruning rules.

We use the above four categories and pruning rules as default information in the knowledge base. In addition, we leave interfaces to add more sophisticated categories and rules. Enhancing the knowledge base is one of the most important future work of NfvInsight.

3.2 Automatic Deployment

Our framework helps to transmit the cumbersome procedure of NFV deployment and measurement into a one-click action. The architecture of our system is shown in Fig.4. Our design for automatic deployment and measurement on multiple servers is composed of four modules (the shadowed part). We will introduce the function of each module respectively, the configuration generator, VNF allocator and network controller in Subsection 3.2, and the performance monitor in Subsection 3.3.

The framework runs upon a particular infrastructure. To leverage the container's lightweight isolation, fast deploying feature, and its convenience of packing runtime environment, VNFs are wrapped in dockers and managed by the hypervisor. The hypervisor takes charge of managing hardware resources of the whole cluster (e.g., computation, storage and network) and does docker allocation. The data path is set up on the designated networking. NfvInsight supports three kinds of networking configurations, which are OVS, Linux bridge, and SR-IOV.

3.2.1 *Configuration Generator*

The chain graphs generated by the graph generation algorithm are taken as this module's input. The generator transforms the graphs into configuration file expressions, which not only indicate the VNF sequences but also contain necessary parameters for VNF deploying and running. Fig.5 shows an example snippet.

First of all, the configuration file generator changes the option *parent* (line 12), which indicates the chain graph topology. The expression "parent = root" means that this VNF is the first node of the chain. For other subsequent VNFs, the *parent* option should be the name of its previous VNF. This information of sequences is used by the module VNF allocator for network allocation and the module network controller for traffic steering respectively.

Apart from the *parent* option, other options all have default values and can be customized for each round of iterative measurements. The content of the configura-

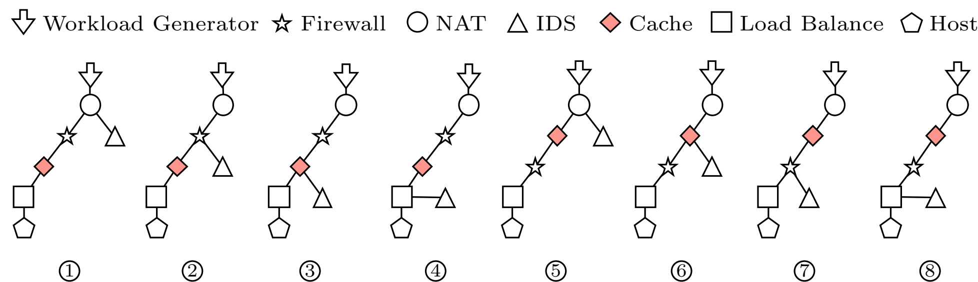


Fig.3. Eight chain graphs.

tion file can be divided into two parts (above and below the dash line in Fig.5). The above includes VNF specific parameters, and the below includes infrastructure-related parameters. The VNF specific parameters are used for VNF launching. NfvInsight provides these options as interfaces for customers to finely tune VNFs. Taking the configuration snippet of weighttp (an HTTP request generator) as an example, the numbers of total requests, TCP connections, working threads, and repeat rate requests are all configurable in our configuration file (lines 2–5 in Fig.5). The infrastructure-related parameters are used by the VNF allocator module for VNF deployment.

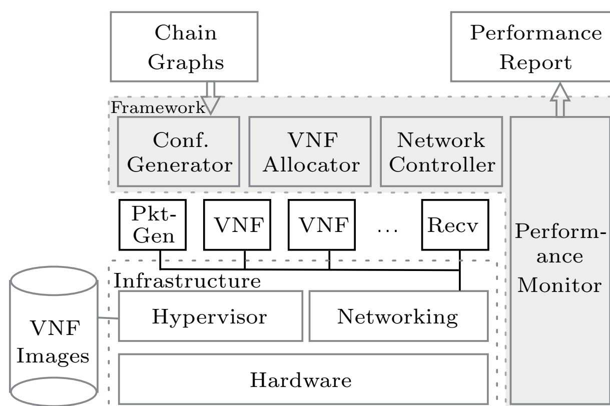


Fig.4. Framework architecture.

```
 1 [weighttp]
 2 requests = 10000
 3 connections = 10
 4 threads = 5
 5 repetitive_rate = 20%
 6 cpu = 1000
 7 memory = 1024
 8 node = k8s01
 9 instances = 1
10 image = ni/weithttp:new
11 parent = root
12 perf_events = default
```

Fig.5. Example of the configuration file format.

3.2.2 *VNF Allocator*

VNF allocation involves hardware resource allocation and VNF launching. In the phase of hardware resource allocation, NfvInsight reads in the infrastructure-related parameters (lines 6–10 in Fig.5) and sends them to the hypervisor when calling the APIs for docker deployment.

These parameters include resources reserved for docker instances (e.g., CPU, memory) and the location information of each VNF. The parameter of the running node and the instance number can be specified and modified in iterative tests. Also, the image of VNF can be specified, and this leaves flexibility to measure different appliances for each type of VNF.

3.2.3 *Network Controller*

This module is responsible for network resource allocation, network adaptation, and traffic stirring among multiple servers.

When doing network resource allocation, NfvInsight arranges IP address for each docker container, and records the usage of IP resource of each round of measurement. For VNFs such as NAT, the chain has to be divided into two LANs. The network controller allocates IPs for two LANs according to VNF sequence information indicated in the configuration file.

NfvInsight currently supports three kinds of network configurations, which are Linux bridge, OVS and SR-IOV. NfvInsight switches among the three according to user demands. For Linux bridge and OVS, dockers are connected to the bridge through Veth pairs, and the physical NIC is also connected to the bridge forming an overlay network. For SR-IOV, virtual functions (VFs), which are identical instantiations of the physical functions (PFs), are directly plugged into docker containers. VF provides line rate network bandwidth through hardware virtualization bypassing the software stack.

When setting up the forwarding path for Linux bridge and SR-IOV, the subsequent VNF IP is specified in the configuration of the previous VNF. For OVS, the communication channels are established by installing OpenFlow rules on the virtual switches. OVS rules are installed by the centralized OpenFlow controller. To provide it with the information needed, this module also records the mapping of (IP, MAC, $ofport$) of each docker and *dpid* of the virtual switch.

3.3 Fine Granularity Measurement

The performance monitor module collects metrics of multiple levels on each server, and synthesizes them on the master to generate final reports for further analysis. In this subsection, we will introduce the metrics NfvInsight supports, how we obtain the fine granularity forwarding the path latency in a low overhead way, and the measurement overhead.

3.3.1 *Multiple-Layer Metrics*

The multiple dimensions of performance metrics are listed in Table 2. NfvInsight currently supports 13 metrics on five layers. The five layers are composed

of application, virtualization, virtual switch, OS, and hardware. The application latency and the application bandwidth are obtained from the reports of packet generators. The bandwidth recorded at the virtualization layer indicates more information than the end-to-end application performance reports. The two metrics of virtual switch reflects its utilization and can be used to judge whether the virtual switch is the bottleneck. Metrics on the OS layer offers a possibility to understand what is happening in the shared system and on the forwarding path in the network stack. The hardware metrics reflect not only CPU distribution, but also the utilization of each core. The architecture-related hardware metrics indicate the information of micro architecture.

Table 2. Performance Metrics

Layer	Feature
Application	Request latency
	Request bandwidth
	CPU utilization
	Memory usage
Virtualization	Virtual device bandwidth
Virtual switch	Number of flow entries
	OVS CPU utilization
OS	Number of syscalls
	Process context switch
	Network stack latency distribution
Hardware	CPU distribution
	Instructions per cycle
	Cache miss

Performance Report. Our framework refines the raw data of measurement results and summarizes them into several entries of the reports. The results are divided into three categories according to the methodology of obtaining the data. The first includes bandwidth, latency or other information reported by workload generators or VNFs. The second is composed of the data from system tools such as docker API and top. The third entry contains metrics retrieved by profiling tools (e.g., perf and ftrace), which bring larger performance degradation than the system tools. The first three categories of report entries are generated by default, while entries of the last two categories can be turned on and off in the configuration file (line 11 in Fig.5). We will showcase how to utilize the reports in Section 5.

3.3.2 Network Stack Latency Distribution

Network stack latency distribution helps to understand the entire life cycle of packets in a server, which is critical for diagnosing and optimizing the network stack. To obtain this information, two difficulties need to be solved. The first one is tracking a unique packet through the network stack, and the second one is to do instrumentation and timing in a low-overhead way.

Packet Tracing. Tracing a single packet throughout the network stack is the first step to get the fine granularity network stack latency. Profiling tools such as Perf can obtain the proportion of the function execution time by sampling the kernel call stack. This method cannot get an accurate function latency. It can only point out the relatively hot function, which is coarse-grained. Additionally, in the scenario of NFV, packets go in and out of the network stack several times. The call stack sampling results superimpose the repeated calls. Thus, only by tracing the packets through the network stack functions can we get the latency of the tortuous forwarding path.

Identifying a unique packet inside a server is challenging. There are mainly two ways to solve the problem. The first one is to add a unique token field in the IP packet header. But it needs to modify the network protocol, making this method customized for a controlled network environment. The second one is to leverage existing fields in *sk_buff*, for example the fields *tstamp* and *rxhash*[17], the gray blocks in Fig.6. However, along with the kernel updating, these two variables have different usages. *tstamp* has been occupied by some special network interface cards filling in a time stamp. *rxhash* has been used to mark different namespaces, and even been deleted since the kernel version 3.15.

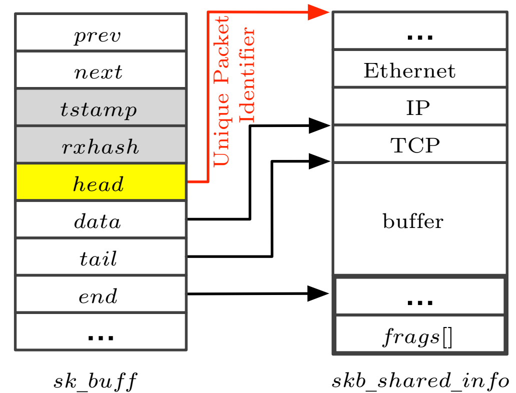


Fig.6. *sk_buff* structure in Linux kernel.

In order to solve the problem of packet unique identification, we innovatively adopt a third way: using the *head* address of *sk_buff* as a unique identifier.

The structure *sk_buff*, as shown in Fig.6, is formed once a packet enters the kernel. The variables in the

first column are metadata of a packet, and among them, the pointers (*head*, *data*, *tail* and *end*) point to the memory location of the real packet. This method has three benefits. 1) The kernel address, which *head* points to, is unique for each packet inside one server. 2) The value of this address is hardly changed, only when memory copy happens. Modifying the packets' metadata does not affect identifying the packet. 3) We only read the address value without any modification of the kernel code.

Instrumentation. To calculate the network stack latency distribution, we need timing at the beginning of each function for each packet. Solving the problem of identifying each packet, the next problem is to insert the "magic code" into kernel functions in a low overhead way.

The traditional instrumentation and profiling tool SystemTap uses Kprobe[15] to perform customized instrumentation in the kernel, but the method is expensive. According to our experiences, the overhead is at least 20%, even probing only one frequently-used kernel function. The root cause is that kprobe first registers the handler function of the instrumentation, and then it replaces the instruction at the instrumentation point to *int3*. When the program executes *int3*, it will break into an interruption and jump to the *int3* handler. If the interrupt handler detects that the source of the interruption is from kprobe, it will look up the previously registered kprobe handler for the next instruction. The *int3*-based mechanism makes kprobe powerful and flexible, but it changes the code execution to the single-step mode, which is slow and expensive.

Our framework uses the mcount mechanism provided by ftrace[16] to do instrumentation. When the Linux kernel is compiled with ftrace on, the compiler adds a *call mcount* instruction at the first line of each kernel function, as illustrated in Fig.7. To prevent the inserted instruction from bringing overhead, ftrace sets the *call mcount* to *NOP*, which is a multi-byte non operation instruction. Thus, when there is no ftrace instrumentation, the performance overhead of *NOP* can be ignored. When the instrumentation is needed, we only need to modify the *NOP* of interested functions into a *CALL*, which invokes the "magic code" to record time stamps. Moreover, because the *CALL* instruction is the first instruction of the probed function, all parameters of this function can be directly accessed by the "magic code", including the unique identifier.

Compared with kprobe, although the call mcount mechanism is functionally weaker, since it can only inject statistical code at the beginning of each function, not anywhere in the code, the overhead is much lower. It satisfies the requirement of our framework's network stack latency measurement.

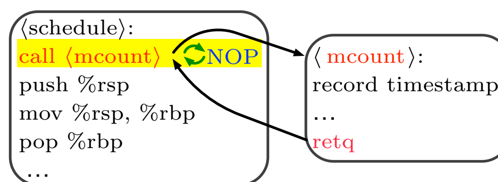


Fig.7. *mcount* mechanism illustration.

3.3.3 Low Overhead Measurement

In order to further reduce the overhead of collecting the network stack latency, we add a sampling method, which means only an appointed ratio of packets is marked to be tracked. The filtering happens at the first network stack function so that the concerned packets are still tracked throughout the whole stack. The overhead of the network stack latency tracking is shown in Fig.8. Along with the decreasing of the sampling ratio, the performance degradation drops from 28% to 15.6% for UDP, and 41.6% to 2.2% for TCP. There is no need to track every packet, and according to our experiences, 1% sampling is enough for getting a sufficient number of packets and observing the trend of latency. Meanwhile, the overhead of 1% is acceptable, especially compared with at least 20% overhead of SystemTap.

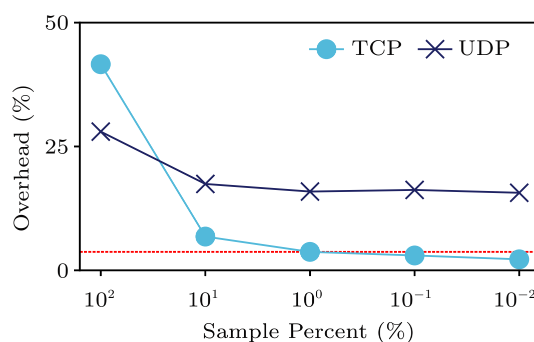


Fig.8. Overhead of the network stack latency tracking tool.

Besides the overhead of the network stack tracing tool, we also evaluate the overhead generated by collecting other performance metrics. The user-level metrics are collected from the reports of the packet generator,

[15] https://www.kernel.org/doc/Documentation/kprobes.txt, Aug. 2020.

[16] https://www.kernel.org/doc/Documentation/trace/ftrace.txt, Aug. 2020.

which does not influence the chain performance. The system tools our framework uses, for example docker API and top, only bring in 0.6% performance degradation for the chains.

4 Implementation

We implement the prototype system NfvInsight in Python3. The prototype uses the rules described in the knowledge base as the system's default pruning rules. These rules are embedded into the code as an implementation of the knowledge base. To realize automation in the testbed module, NfvInsight generates a series of scripts to allocate VNFs, starts tests, and does performance profiling.

NfvInsight leverages Kubernetes (v1.11) as the hypervisor to manage the physical cluster and docker (v17.03), and do resource allocation and docker placement. To integrate NfvInsight's data plane with Kubernetes' network management, our cluster has two sets of networks: one for cluster management, and the other one for data path. The management network is under the control of Kubernetes, which connects each docker with a Linux bridge via 1 Gb ethernet. The data path of VNFs is set on 40 Gb ethernet. Docker stats is used to collect performance data, as well as Perf. Our ftrace-based network latency tracking tool requires the kernel version 4.19.

5 Case Study

In this section, cases are conducted to illustrate the convenience and usability of NfvInsight in finding bottlenecks for both VNF chains and the system. As summarized in Fig.9, the first case shows that NfvInsight has the ability to identify the bottleneck VNF in a specified chain through a series of iterative tests. The second case shows that several NfvInsight designs in the network stack are not highly efficient for internal packet transferring. We further provide solutions to optimize the system and gain performance improvement. Before elaborating on the cases in detail, the network functions, workloads and the hardware platform used to do the experiments are introduced.

Network Functions. Five typical NFs commonly used in enterprise networks or data centers are picked to accomplish the case study.

• *IDS.* Snort[17] is used for intrusion prevention, real-time traffic analysis, and packet logging. To feed traffic to Snort, OVS rules are installed to duplicate and redirect out coming flows of the previous VNF.

• *NAT.* iptables[18] is set with particular rules as DNAT and SNAT. Two virtual network interfaces are plugged into the docker of NAT to simulate two LANs.

• *FW.* The firewall is also implemented by iptables[18]. Carefully-crafted rules are configured to filter the traffic.

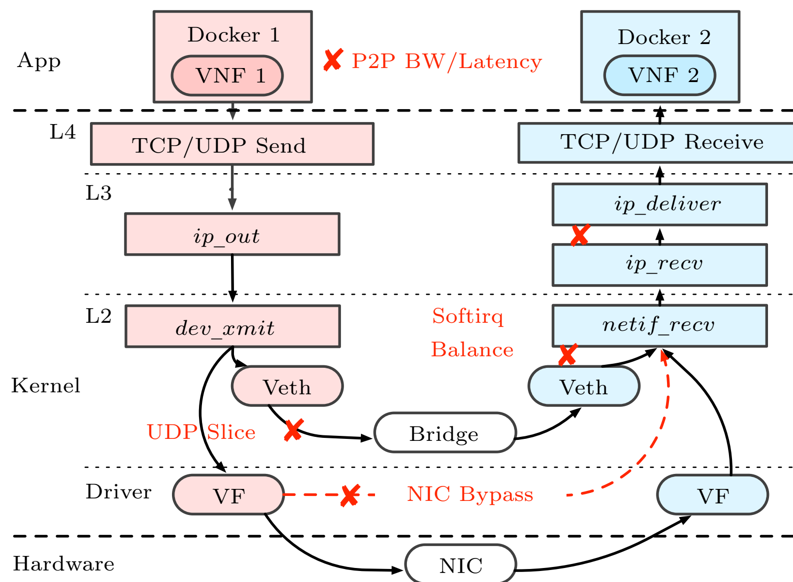


Fig.9. Summary of the case study.

[17] https://www.snort.org/, Aug. 2020.
[18] https://netfilter.org/projects/iptables/, Aug. 2020.

• *LB*. HAProxy[19] performs as an L4 load balancer. Two sets of connections are maintained for the front-ends and back-ends respectively. The load balanced mappings between connections are recorded by the proxy.

• *Cache*. Squid[20] is an L7 content caching. It is configured to work as a reverse proxy, which reduces response time by caching and reusing frequently-visited web pages.

Workloads. To feed the chain with packet flows, three types of workloads are used. The first two are TCP and UDP traffic generated by netperf[21], and the third one is HTTP requests generated by weighttp[22]. Compared with HTTP requests, TCP and UDP traffic do not have the procedure of user layer packets processing. In the experiments, netperf is run for two minutes, which is long enough to obtain the stable network bandwidth. Meanwhile, HTTP requests are more appropriate to simulate enterprise or data center workloads. The client weighttp gets static pages from the Apache server. Additionally, weighttp is modified to send requests with a given repetitive rate. The proportion of repeated requests reflects the locality of HTTP workloads.

When doing measurements, both the workload generator and the workload receiver should work under maximum performance. Our testbed prototype emulates the workloads from enterprise networks accessing to data center servers. The series of VNFs on this path in front of the servers brings down the bandwidth or extend latency. To do pressure tests, the workload generators should guarantee the maximum performance when there is no VNF on the path. To achieve the line rate as the baseline bandwidth in TCP workload tests, multiple pairs of netperf clients and servers are used. For HTTP workloads, experiments show that four threads and 64 connections (eight connections per thread) achieve the maximum bandwidth. We also find that one single Apache server cannot fully fill the bandwidth of a 40 Gb NIC, because the utilization of the CPU resource limits the performance. Thus, four Apache servers are used. For each test, we run the experiment five times and calculate the average.

Hardware Platform. Our experiments are conducted on three physical servers with dual sockets Intel Xeon E5-2650 v4 2.2 GHz, 24 cores with SMT turned on, 32 GB DRAM, 40 Gb NIC, and the servers are connected by 40 Gb Mellanox switch. The servers run Ubuntu 16.04 with kernel 4.19 and VNFs are encapsulated in dockers running Ubuntu 14.04. For more realistic simulation, all VNFs run on a single physical server, while load generators and receivers run on the other two servers respectively.

5.1 Bottleneck VNF Identification

This case intends to present the process of NfvInsight identifying the bottleneck VNF which limits the performance of the entire chain. In addition, VNF instance scale-out schemes are utilized to obtain a better performance.

Performance of the Specified VNFs. We first measure the maximum bandwidth of the eight chain graphs generated according to our graph generation algorithm. In Fig.10, the chain index is in consistency with that in Fig.3. For each chain, along with the rising of requests repeat ratio (rrr) of HTTP, the maximum bandwidth rises, and even doubles when all the requests are asking for the same page. The increasing of the repeat ratio only accelerates Cache's processing time. According to system acceleration ratio rules, Cache can be inferred as the bottleneck VNF, though we need more proof.

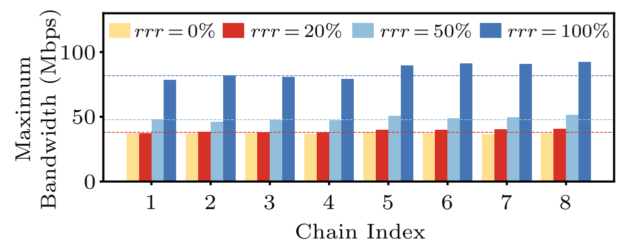


Fig. 10. Bandwidth of eight chain graphs when the workload locality of HTTP requests varies.

In addition, the sequence of VNFs in the chain results in performance difference. Though being very slight, the last four graphs perform better than the first four. It is because of Cache locating in front of FW and reducing the workload volume.

Single VNF Performance. To find out the performance impact of each VNF, chain graphs with only one VNF are measured. In Fig. 11, the bar "None", as the baseline performance, is measured in the case where there is no VNF between the load generator and

[19] https://www.haproxy.org/, Aug. 2020.

[20] http://www.squid-cache.org/, Aug. 2020.

[21] https://hewlettpackard.github.io/netperf/, Aug. 2020.

[22] https://github.com/lighttpd/weighttp, Aug. 2020.

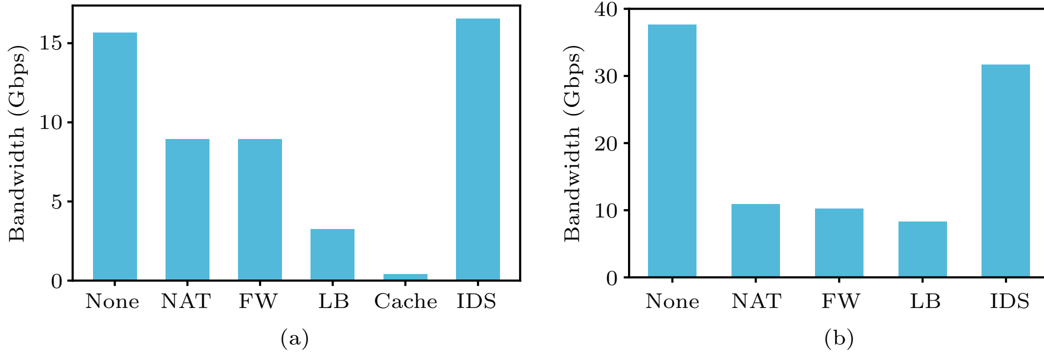


Fig.11. Bandwidth of single VNF. (a) HTTP workload. (b) TCP workload.

the receiver. The bandwidth results show that Cache degrades the performance most by 93.92%. A simple and straightforward idea is to expand the number of instances of each single VNF to understand the upper bound of performance gain through scaling out. Therefore, the next round of measurement is single VNF scale-out.

Single VNF Scale-out. The numbers of instances of NAT, FW, LB and Cache are expanded as shown in Fig.12 driven by the HTTP workload. For further confirmation, TCP workload is also used for this experiment shown in Fig.13, though without Cache. For the TCP workload, when scaling out NAT, FW and LB, it reaches near baseline bandwidth (37.8 Gbps, the same as "None" of Fig.11(b)). For both HTTP and TCP workloads, these three VNFs reach the near baseline bandwidth, roughly 40 Gbps for TCP and 15 Gbps for HTTP. Thus, a conclusion can be made that NAT, FW and LB limit the entire chain bandwidth, but they will not be the bottleneck VNF after scaling-out.

However, when it comes to Cache, even scaling-out cannot make the bandwidth equal to the baseline. In Fig.12(b), the bandwidth stops scaling when the number of Cache instances reaches 8.

Bottleneck Scale-out. Cache has greater performance influence than the other VNFs. Thus, it is chosen to be scaled out in the chain graph in the first place. We take chain graph ① in Fig.3 as an example. As shown in Fig.14, the bandwidth grows until the instance number of Cache reaches 10. The performance with 10 Cache instances is 3.59 times better, compared with only one Cache instance in the chain. When the instance scales to 12, the entire chain suffers sharp degradation and fluctuation. Thus, scaling to 10 reaches the upper bound.

Summary. Experiments in this case reveal that VNFs in a service chain are chained like tubes of different thickness connected up. Since the VNFs used in our case are socket-based, the number of flows a single VNF can process at one time can be analogized to the thickness of the tube. The input workloads are like water flowing through a pipe. The flow of the entire pipe is limited by the thinnest point. The thickening of other none bottleneck points does not bring changes in performance. For the service chain, the entire chain performance is limited by the slowest VNF, which is Cache in this case. Scaling it out at most gains a performance 4x better.

5.2 System Bottleneck

To find internal packet transferring bottlenecks, we perform all the experiments by sending packets directly between the sender and the receiver, and we call

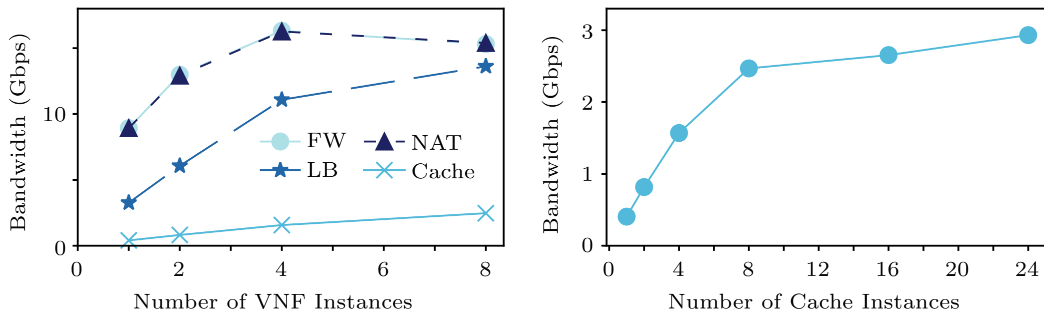


Fig.12. Bandwidth of VNF scale-out under HTTP workloads.

this point-to-point measurement (P2P). We use netperf sending TCP/UDP packets of the same size (64 KB) to simulate the flow inside a physical machine. We bound the sender thread and the receiver thread to two separated CPU cores for two reasons. One is that the migration of threads on CPU cores causes unstable network performance. The other one is that VNFs are usually bound to CPU cores in the production environment.

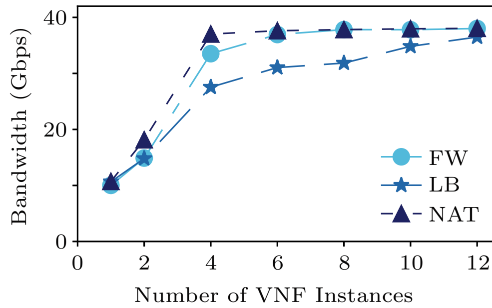


Fig.13. Bandwidth of scaling out each VNF under TCP.

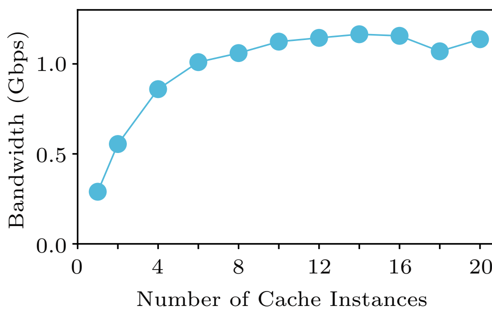


Fig.14. Bandwidth of scale-out cache in chain.

The bandwidth and softirq core distribution under three kinds of network configurations (OVS, Linux bridge (Br), and SR-IOV) driven by two kinds of workloads are showed in Fig.15 and Table 3 respectively. By analyzing and comparing this data, we find three problems in the network stack. We analyze the root cause and optimize them separately.

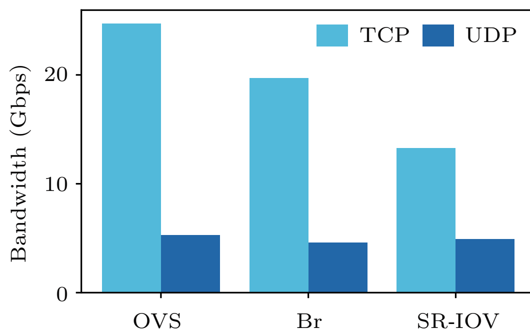


Fig.15. Bandwidth of three network configurations under two workloads.

Table 3. CPU Usage Distribution of Softirq

Network Configuration	Application	Clinet Node (%)	Server Node (%)
Linux bridge	TCP	35.6	34.3
	UDP	63.9	0.0
OVS	TCP	25.5	23.3
	UDP	56.0	0.0
SR-IOV	TCP	0.0	0.0
	UDP	50.3	0.0

5.2.1 *SR-IOV Internal Forwarding*

In Fig.15, for the TCP workload, the forwarding bandwidth of SR-IOV is 32.5% lower than that of Linux bridge, and 46.3% lower than that of OVS. SR-IOV is different from the other two network configurations in that the virtual functions (VF) are created by virtualizing the hardware NIC. Therefore, we conduct an in-depth analysis of SR-IOV's internal forwarding mechanism. The tracked network stack functions are listed in Table 4. The sampling ratio is set to 1%.

Table 4. Tracked Network Stack Functions

Function Name	Index
ip_local_out	F1
__dev_queue_xmit	F2
veth_xmit	F3
__netif_receive_skb	F4
ip_rcv	F5
ip_local_deliver	F6
tcp_transmit_skb	F7
tcp_v4_rcv	F8
i40evf_xmit_frame	F9
napi_gro_receive	F10
udp_send_skb	F11
udp_rcv	F12

The network stack tracking results in Fig.16(a) show that packets have been transferred to the physical NIC between F9 and F10. Receiving and reconstructing packets (between F10 and F4) results in an extremely long latency, because in this process, memory space is allocated for new packets, and memory copy happens to construct skb in kernel, which is time consuming. We further analyze the implementation of SR-IOV device driver, which is Intel i40evf for the NIC card we use. We find out that SR-IOV does not have a mechanism bypassing NIC for intra server packets forwarding.

Thus, we modify the NIC driver by adding a judgment of whether the destination MAC address of the packet is a VF inside the same machine. If so, the packet is forwarded directly back to the network stack,

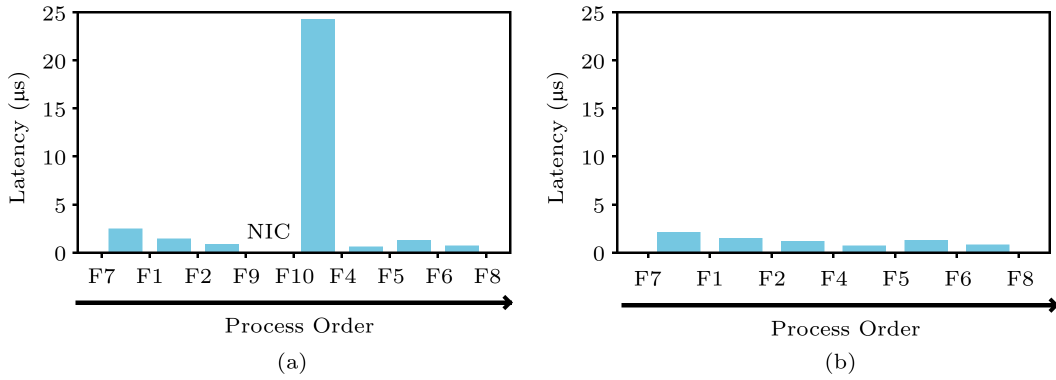


Fig.16. Network stack latency distribution of SR-IOV. (a) Original forwarding. (b) Modified forwarding.

bypassing the NIC. The network stack latency distribution after this optimization is shown in Fig.16(b). The functions of exiting and entering NIC are replaced by forwarding inside the software network stack, and the extremely long latency is reduced. The bandwidth after modification, compared with original SR-IOV, increases by 100% and 37% for TCP and UDP respectively. The performance improves because the forwarding function we use in replacement deals with the skb without any memory operation.

5.2.2 *Softirq Core Affinity Imbalance*

Observing the data in Table 3, CPU usages of softirq (si) of UDP are all concentrated on the sender core (shown in red, meaning si imbalanced), where the client thread locates. But for TCP, it is different. In comparison, the CPU usages of softirq for the TCP traffic are distributed on the sender core and the receiver core (shown in green, meaning si balanced). The softirq CPU usages of the TCP traffic of SR-IOV are exceptional, which are both zero, but they are still si-balanced. When using SR-IOV, softirq is triggered by NIC but not the software stack. Softirq triggered by NIC appears on an arbitrary core, which can be the sender core or the receiver core or any other. We observe the bandwidth fluctuation of SR-IOV TCP when the softirq thread migrates from core to core. Combined with the bandwidth data of UDP workloads, we conclude that softirq threads are CPU-intensive, and interfere the performance of the service. Therefore, we try to change the softirq imbalance of UDP on the Linux bridge.

The original UDP softirq locates on the sender core as shown in Fig. 17(a). We look into the code of Linux bridge and find that softirq appears twice at both the sending and the receiving processes. We modify the code to split the two softirq threads. There are three optimization schemes to allocate the split softirq threads, as shown in Figs.17(b)–17(d). We choose the plan in Fig.17(b) to balance the send softirq on the sender core (S) and the receive softirq on the receiver core (R), because in this case softirq does not occupy any other CPU cores. We suppose that VNFs should not occupy more resources than allocated unless it is authorized.

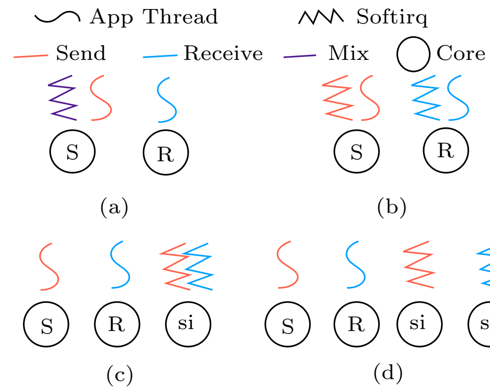


Fig.17. UDP softirq distribution on core.

We write a dynamically loadable kernel module. The module can extract the topology information of the chain from the configuration file of the framework, and determine the core binding scheme before the chain is deployed and run. The method we use avoids checking the owner of a softirq thread and reconfiguring the kernel after the VNF startup. As Fig.17 shows, softirq is balanced by our approach. The CPU usage in Table 5 proves it. The resulting bandwidth is shown in Fig.18 (Br-LB) which increases by 22%. In this situation, the client core's CPU idle is zero, which means the client reaches the CPU resource limitation. Thus, softirq core affinity imbalance limits the internal packet transfer. We also test the bandwidth of plans in Fig.17(c) and Fig.17(d). The bandwidth increases by 61.2% and

135.3% respectively compared with the original plan in Fig. 17(a), so that allocating more CPU resources for softirq can gain performance improvement.

Table 5. CPU Usage Distribution of Softirq After Being Balanced

Network Configuration	Application	Client Node (%)	Server Node (%)
Linux bridge	UDP	51.2	34.3

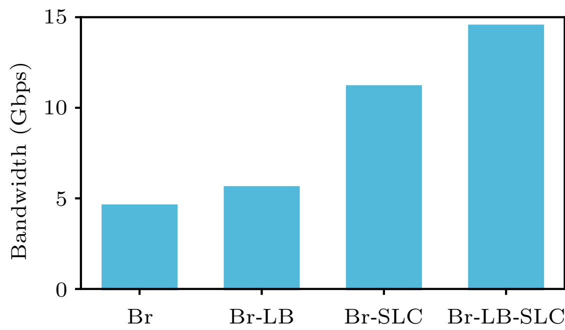


Fig.18. Bandwidth of UDP after optimizations.

5.2.3 *UDP Slicing*

Though sending packets of the same size, the bandwidth of UDP is 71.5% lower than that of TCP, even after the UDP softirq imbalance optimization. Therefore, we perform a delay analysis on the forwarding process of UDP packets on Linux bridge, as shown in Fig.19.

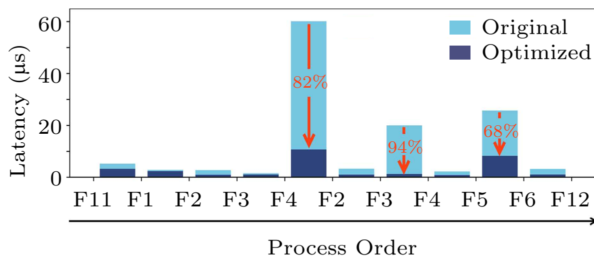


Fig.19. Network stack latency distribution of UDP before and after slicing optimization.

The latency distribution has three spikes. We analyze the code of network stack to figure out the root cause. From F11 to F3, the packet is sent from the sender to the bridge. The packet is sliced by the L3 protocol because NIC does not support UFO (UDP fragmentation offload). Then the bridge receives the packet after F4 and sends it out in F2. In this phase, the packet has been merged once for Netfilters PREROUTING and sliced again for further forwarding. The receiver receives the packet in the second F4, and processes it afterwards. Between F5 and F6, the packet is merged again on L3. Thus, UDP packets' slicing and merging for internal transmitting happen several times, resulting in long latencies. There is no need to slice and merge the packets if they do not go out of the physical NIC.

We modify the network stack to avoid slicing in both L3 and L2, and the three latency spikes are eliminated as shown in Fig.19. The bandwidth of Br-SLC increases by 143.3% shown in Fig.18. We apply softirq optimization together with UDP slicing and gain 216.1% performance improvement as shown in Br-LB-SLC.

6 Related Work

We survey the related studies on the performance of the NFV system, and we summarize them from three aspects, which include monitoring and analyzing the performance, accelerating the infrastructure, and optimizing the programming model.

Performance Analysis. We list and compare the frameworks focusing on monitoring and analyzing the NFV performance in Table 6.

ConMon [18] and NFVPerf [19] utilize the OVS mirroring mechanism and packet capturing libraries to record packets passively, forming packets logs. The logs are parsed and analyzed online in a specified interval. The information includes the user-level bandwidth and latency, and even packet loss and jitter are obtained. These two frameworks have similar features, and they

Table 6. Performance Analysis Frameworks

Related Work	VNF Topology	Instance Scale-out	Instance Scale-up	CPU Schedule	Memory Access	Network Stack	Virtualization		Online Analysis	Offline Performance
							Docker	KVM		
ConMon [18]							√		√	
NFVPerf [19]								√	√	
MeDICINE [20]			√				√			√
SCC [21]				√	√		√			√
Probius [12]	√			√	√			√		√
NfvInsight	√	√	√	√	√	√	√			√

only differ in the virtualization platform. NfvInsight does not use the high-overhead method of packets capturing to collect metrics, but leverages the user-level reports of the packet generator. Another framework [20], which is built on MeDICINE, considers the performance change when the CPU resource of a single VNF scales up. However, it still leaves many other factors to be concerned.

In addition, SCC [21] and Probius [12] conduct the in-depth analysis of the cache and memory system, as well as CPU scheduling. Under the container environment, CPU scheduling happens with context switching between processes wrapped in containers. For virtual machines, VM state transition results in CPU scheduling. SCC and Probius analyze these two virtualization platforms separately. Probius does anomaly detection in NFV systems. The online data is monitored and compared with offline performance metrics to discover abnormal issues. It also takes the topology of VNF chains as a factor which influences the performance.

NfvInsight aims to cover all these performance-related factors. Apart from the aspects concerned in other frameworks, NfvInsight takes into account the instance scale-out situation and latency breakdown of the network stack. Besides, for the VNF topology, NfvInsight collects rules to form a knowledge base and generates reasonable chain graphs. To the best of our knowledge, this is a first attempt.

Programmable Optimization. Some studies follow the idea of SDN to expose programming interfaces for service chain optimization. For example, FreeFlow [22] proposes a new abstraction for transparent and balanced migration of virtual middleboxes, and enables dynamic and stateful traffic scheduling. OpenNF [3] decouples VNF states from their processing logic and proposes APIs for state management and optimizations. PGA [15] checks user-specified policies and composes conflict-free chains accordingly. OpenBox [4] tears down VNFs into several primitive functions and removes duplicated parts for the resource efficiency. NFP [6] lets users specify processing orders of VNFs and identify the feasibility of parallelism. These studies rely on domain knowledge from network operators. NfvInsight differs from them by benchmarking enumerated graphs, which minimizes the requirement for the domain knowledge.

Infrastructure Acceleration. Some studies address to accelerating the underlying NFV infrastructures. For example, NetVM [1] and OpenNetVM [2] leverage the DPDK library㉓ to speed up the packet delivery for KVM and Docker, respectively. Metron [5] takes the advantages of smart NICs to offload stateless VNFs processing to hardware. Our work NfvInsight is orthogonal, and its testbed component can adopt these techniques to boost the service chain benchmarking.

Graph-Related Acceleration. NFP [6] and Parabox [23] have a similar idea and provide methods to judge whether two NFs can be paralleled. According to the actions NFs done to packets (e.g., reading or writing the 5-tuple, adding or deleting the header, and dropping packets), their algorithms give out parallelism identification results. In the future, we can add their algorithms into the knowledge base of NfvInsight to determine which NF can be paralleled.

Model-Based Optimization. Some studies leverage the methodology of formalization to quantify the factors influencing the performance of the NFV platform [24]. Models are built for various performance issues including resource requirement [25], traffic changing effects (VNF sequence effects) [11], CPU allocation [26], latency constraint [27], virtual switch cost [28], and operational costs and utilization [29]. However, existing models lack the generality because each model only addresses one issue. Further, it remains an open issue to model performance uncertainties such as performance interference. NfvInsight covers all performance issues because it performs comprehensive benchmarks, and the performance issues are all reflected in the measurement results.

7 Conclusions

We designed and implemented NfvInsight, a distributed testing system automatically deploying VNFs and steering the network traffic to simplify the benchmarking procedure. Through iterative measurement, NfvInsight is able to pinpoint the performance bottleneck of NFV systems. Our framework aids network operators in the cumbersome and error-prone routine of benchmarking the NFV system, and helps them figure out performance issues. We conducted two case studies to prove the effectiveness of NfvInsight and make the following conclusions.

- The VNFs can be chained with different sequences. NfvInsight provides rules for selecting chains with reasonable sequences. The measurement results provided evidence for the claim that the order of VNFs in chain influences chain performance.

㉓https://www.intel.com/content/www/us/en/developer/topic-technology/networking/dpdk.html, May 2022.

• Chained VNFs and the forwarding path form a water pipe, especially for the services setting up the L3 connection. The input workload is like water flowing through the tube and limited by the thinnest point. In our case, the VNF Cache throttles the end-to-end performance.

• It is a typical scenario that several VNFs are allocated on one single server. NfvInsight can detect the narrowness of the packet forwarding path. The measurement results helped locate and analyze the design flaws in the system.

Acknowledgements We thank all the anonymous shepherd and reviewers for their valuable comments. We are especially grateful to Zhi-Cheng Yao (Institute of Computing Technology, Chinese Academy of Sciences, Beijing) for his generous help.

References

[1] Hwang J, Ramakrishnan K K, Wood T. NetVM: High performance and flexible networking using virtualization on commodity platforms. *IEEE Transactions on Network and Service Management*, 2015, 12(1): 34-47. DOI: 10.1109/TNSM.2015.2401568.

[2] Zhang W, Liu G, Zhang W, Shah N, Lopreiato P, Todeschi G, Ramakrishnan K, Wood T. OpenNetVM: A platform for high performance network service chains. In *Proc. the 2016 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, August 2016, pp.26-31. DOI: 10.1145/2940147.2940155.

[3] Gember-Jacobson A, Viswanathan R, Prakash C, Grandl R, Khalid J, Das S, Akella A. OpenNF: Enabling innovation in network function control. *ACM SIGCOMM Computer Communication Review*, 2014, 44(4): 163-174. DOI: 10.1145/2740070.2626313.

[4] Bremler-Barr A, Harchol Y, Hay D. Openbox: A software-defined framework for developing, deploying, and managing network functions. In *Proc. the 2016 ACM SIGCOMM Conference*, August 2016, pp.511-524. DOI: 10.1145/2934872.2934875.

[5] Katsikas G P, Barbette T, Kostic D, Steinert R, Maguire G Q. Metron: NFV service chains at the true speed of the underlying hardware. In *Proc. the 15th USENIX Symposium on Networked Systems Design and Implementation*, April 2018, pp.171-186.

[6] Sun C, Bi J, Zheng Z, Yu H, Hu H. NFP: Enabling network function parallelism in NFV. In *Proc. the Conference of the ACM Special Interest Group on Data Communication*, August 2017, pp.43-56. DOI: 10.1145/3098822.3098826.

[7] Yi X, Duan J, Wu C. GPUNFV: A GPU-accelerated NFV system. In *Proc. the 1st Asia-Pacific Workshop on Networking*, August 2017, pp.85-91. DOI: 10.1145/3106989.3106990.

[8] Bronstein Z, Roch E, Xia J, Molkho A. Uniform handling and abstraction of NFV hardware accelerators. *IEEE Network*, 2015, 29(3): 22-29. DOI: 10.1109/MNET.2015.7113221.

[9] Kachris C, Sirakoulis G, Soudris D. Network function virtualization based on FPGAs: A framework for all-programmable network devices. arXiv:1406.0309, 2014. https://arxiv.org/ftp/arxiv/papers/1406/1406. 0309.pdf, Dec. 2021.

[10] Ersue M. ETSI NFV management and orchestration—An overview. https://www.ietf.org/proceedings/88/slides/slides-88-opsawg-6.pdf, Dec. 2021.

[11] Ma W, Sandoval O, Beltran J, Pan D, Pissinou N. Traffic aware placement of interdependent NFV middleboxes. In *Proc. the 2017 IEEE Conference on Computer Communications*, May 2017. DOI: 10.1109/INFOCOM.2017.8056993.

[12] Nam J, Seo J, Shin S. Probius: Automated approach for VNF and service chain analysis in software-defined NFV. In *Proc. the Symposium on SDN Research*, March 2018, Article No. 14. DOI: 10.1145/3185467.3185495.

[13] Sherry J, Hasan S, Scott C, Krishnamurthy A, Ratnasamy S, Sekar V. Making middleboxes someone else's problem: Network processing as a cloud service. *ACM SIGCOMM Computer Communication Review*, 2012, 42(4): 13-24. DOI: 10.1145/2377677.2377680.

[14] Koshy T. Catalan Numbers with Applications. Oxford University Press, 2008.

[15] Prakash C, Lee J, Turner Y, Kang J M, Akella A, Banerjee S, Clark C, Ma Y, Sharma P, Zhang Y. PGA: Using graphs to express and automatically reconcile network policies. *ACM SIGCOMM Computer Communication Review*, 2015, 45(4): 29-42. DOI: 10.1145/2829988.2787506.

[16] Hubert B, Maxwell G, Van Mook R *et al*. Linux advanced routing & traffic control HOWTO. https://tldp.org/HOWTO/pdf/Adv-Routing-HOWTO.pdf, Dec. 2021.

[17] Blake G, Saidi A G. Where does the time go? Characterizing tail latency in memcached. In *Proc. the 2015 IEEE International Symposium on Performance Analysis of Systems and Software*, March 2015, pp.21-31. DOI: 10.1109/ISPASS.2015.7095781.

[18] Moradi F, Flinta C, Johnsson A, Meirosu C. ConMon: An automated container based network performance monitoring system. In *Proc. the 2017 IFIP/IEEE Symposium on Integrated Network and Service Management*, May 2017, pp.54-62. DOI: 10.23919/INM.2017.7987264.

[19] Naik P, Shaw D K, Vutukuru M. NFVPerf: Online performance monitoring and bottleneck detection for NFV. In *Proc. the 2016 IEEE Conference on Network Function Virtualization and Software Defined Networks*, November 2016, pp.154-160. DOI: 10.1109/NFV-SDN.2016.7919491.

[20] Peuster M, Karl H. Understand your chains: Towards performance profile-based network service management. In *Proc. the 5th European Workshop on Software-Defined Networks*, October 2016, pp.7-12. DOI: 10.1109/EWSDN.2016.9.

[21] Katsikas G P, Maguire G Q, Kostić D. Profiling and accelerating commodity NFV service chains with SCC. *Journal of Systems and Software*, 2017, 127: 12-27. DOI: 10.1016/j.jss.2017.01.005.

[22] Rajagopalan S, Williams D, Jamjoom H, Warfield A. Split/merge: System support for elastic execution in virtual middleboxes. In *Proc. the 10th USENIX Symposium on Networked Systems Design and Implementation*, April 2013, pp.227-240.

[23] Zhang Y, Anwer B, Gopalakrishnan V, Han B, Reich J, Shaikh A, Zhang Z L. ParaBox: Exploiting parallelism for virtual network functions in service chaining. In *Proc. the Symposium on SDN Research*, April 2017, pp.143-149. DOI: 10.1145/3050220.3050236.

[24] Duan Q. Cloud service performance evaluation: Status, challenges, and opportunities—A survey from the system modeling perspective. *Digital Communications and Networks*, 2017, 3(2): 101-111. DOI: 10.1016/j.dcan.2016.12.002.

[25] Feng H, Llorca J, Tulino A M, Raz D, Molisch A F. Approximation algorithms for the NFV service distribution problem. In *Proc. the 2017 IEEE Conference on Computer Communications*, May 2017. DOI: 10.1109/INFOCOM.2017.8057039.

[26] Agarwal S, Malandrino F, Chiasserini C F, De S. Joint VNF placement and CPU allocation in 5G. In *Proc. the 2018 IEEE Conference on Computer Communications*, April 2018, pp.1943-1951. DOI: 10.1109/INFOCOM.2018.8485943.

[27] Cziva R, Anagnostopoulos C, Pezaros D P. Dynamic, latency-optimal vNF placement at the network edge. In *Proc. the 2018 IEEE Conference on Computer Communications*, April 2018, pp.693-701. DOI: 10.1109/INFOCOM.2018.8486021.

[28] Luizelli M C, Raz D, Sa'ar Y. Optimizing NFV chain deployment through minimizing the cost of virtual switching. In *Proc. the 2018 IEEE Conference on Computer Communications*, April 2018, pp.2150-2158. DOI: 10.1109/INFOCOM.2018.8486315.

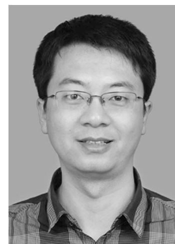
[29] Bari M F, Chowdhury S R, Ahmed R, Boutaba R. On orchestrating virtual network functions. In *Proc. the 11th International Conference on Network and Service Management*, November 2015, pp.50-56. DOI: 10.1109/CNSM.2015.7367338.

Tian-Ni Xu is a Ph.D. candidate at University of Chinese Academy of Sciences (UCAS), and Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing. She received her B.S. degree in network engineering from Beijing University of Posts and Telecommunications, Beijing, in 2013. Her research interests include computer network, network function virtualization, operating system, and system performance modeling and evaluation.

Hai-Feng Sun received his B.S. degree in computer science from Beijing Forestry University, Beijing, in 2017, and his M.S. degree in computer architecture from Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, in 2020. His research interests include distributed systems and computer networks.

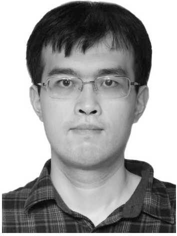
Di Zhang is a Ph.D. candidate at University of Chinese Academy of Sciences (UCAS), and Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing. She received her B.S. degree in computer science from Harbin Institute of Technology, Harbin, in 2014. Her research mainly focuses on system performance analysis.

Xiao-Ming Zhou received his B.S. degree in computer science from Nankai University, Tianjin, in 2016, and his M.S. degree in computer architecture from University of Chinese Academy of Sciences (UCAS), and Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, in 2019. His research mainly focuses on computer networks.



Xiu-Feng Sui received his B.S. degree from Harbin Institute of Technology, Harbin, in 2005, and his Ph.D. degree in computer science from University of Science and Technology of China, Hefei, in 2010. He is currently an associate professor in the School of Information and Electronics, Beijing Institute of Technology (BIT), Beijing. Before joining BIT, he worked at Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, from March 2011 to December 2018. His research interests include computer architecture and system performance modeling and evaluation.

Sa Wang received his B.S. degree in computer science from University of Science and Technology of China, Hefei, in 2009, and his Ph.D. degree in computer science from the Chinese Academy of Sciences, Beijing, in 2016. He is an associate professor in Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing. His current research interests include operating system, system performance evaluation and optimization, and distributed system. He is a member of CCF and ACM.



Qun Huang received his B.S. degree in computer science from Peking University, Beijing, in 2011, and his Ph.D. degree in computer science in 2015 in Chinese University of Hong Kong, Hong Kong. He is now an assistant professor (Tenure-Track) at Department of Computer Science and Technology, Peking University (PKU), Beijing. Before joining PKU, he worked at Institute of Computing Technology, Chinese Academy of Sciences, Beijing, from September 2017 to May 2020, and at Huawei Future Network Theory Laboratory from September 2015 to September 2017. His research mainly focuses on distributed stream processing and network measurement. He is a member of CCF, ACM, and IEEE.

Yun-Gang Bao received his B.S. degree from Nanjing University, Nanjing, in 2003, and his Ph.D. degree in computer science from Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, in 2008. He is a professor at ICT, CAS, Beijing. From 2010 to 2012, he was a postdoctoral researcher at Department of Computer Science, Princeton University, New York City. His current research interests include computer architecture, operating system, and system performance modeling and evaluation. He is a senior member of CCF and a member of ACM and IEEE.