Finedge: A Dynamic Cost-Efficient Edge Resource Management Platform for NFV Network

Miao Li Qixia Zhang Fangming Liu*
National Engineering Research Center for Big Data Technology and System,
Key Laboratory of Services Computing Technology and System, Ministry of Education,
School of Computer Science and Technology, Huazhong University of Science and Technology, China

***Abstract*—With the evolution of network function virtualization (NFV) and edge computing, software-based network functions (NFs) can be deployed on closer-to-end-user edge servers to support a broad range of new services with high bandwidth and low latency. However, due to the resource limitation, strict QoS requirements and real-time flow fluctuations in edge network, existing cloud-based resource management strategy in NFV platforms is inefficient to be applied to the edge. Thus, we propose *Finedge*, a dynamic, fine-grained and cost-efficient edge resource management platform for NFV network. First, we conduct empirical experiments to find out the effect of NFs' resource allocation and their flow-level characteristics on performance. Then, by jointly considering these factors and QoS requirements (e.g., latency and packet loss rate), Finedge can automatically assign the most suitable CPU core and tune the most cost-efficient CPU quota to each NF. Finedge is also implemented with some key strategies including real-time flow monitoring, elastic resource scaling up and down, and also flexible NF migration among cores. Through extensive evaluations, we validate that Finedge can efficiently handle heterogeneous flows with the lowest CPU quota and the highest SLA satisfaction rate as compared with the default OS scheduler and other state-of-the-art resource management schemes.**

I. Introduction

By shifting computing and storage capacities from the remote cloud to the network edge, *edge computing* emerges as a promising technology to provide elastic resources for end users with high bandwidth and low latency [1], [2]. As compared with cloud servers, *edge servers* are deployed in close proximity to the source of data and end devices. In short, edge computing can significantly reduce the amount of user-cloud data exchange and decrease end-to-end latency, which can support a wide range of services, such as virtual reality (VR), augmented reality (AR) and other real-time and computation-sensitive services [3].

Moreover, with network function virtualization (NFV), traditional hardware-implemented middleboxes can be implemented as software-based network functions (NFs). These software-based NFs can be placed on commercial-off-the-shelf (COTS) servers, including the remote cloud servers and the closer-to-end-user edge servers, which brings great benefits like decreased Capital Expenditure (CAPEX), reduced Operational Expenditure (OPEX), fast deployment of network services and flexible resource provisioning [4], [5]. However, edge servers are usually not as sufficient as cloud servers in computation and storage resources [6], [7], thus how to efficiently allocate edge resources to the service-required NFs and how to dynamically manage them are two important issues needed to be solved.

In recent years, several NFV platforms arise to support efficient packet processing, service provisioning and resources management for core cloud and datacenters, e.g., E2 [8], OpenNF [9], OpenNetVM [10], Flurries [11], NFVnice [12], NFP [13], FERO [14] and Metro [15]. In [8], [9], [15], an NF can exclusively occupy a CPU core and its relevant resources (e.g. cache or memory) in a cloud server; however, considering the limited resource capacity of edge servers, multiple NFs usually have to share a CPU core and its resources of an edge server. Even though in some existing NFV platforms (e.g., [10]–[14]), multiple NFs can share the same core, they have not fully captured that some service requests in edge network have strict QoS requirements (e.g., high throughput, low latency and/or low packet loss rate) and their flow rates can be unpredictable and time-varying [16]. By jointly considering the resource limitation, QoS constraints and dynamic flow characteristics in edge-based NFV networks, most of the existing cloud-based NFV platforms are inefficient to be directly applied to the edge.

Different from these existing NFV platforms, a dynamic, fine-grained and cost-efficient edge resource management platform is still needed for effectively assigning and adjusting edge resources to NFs. However, considering the special character-

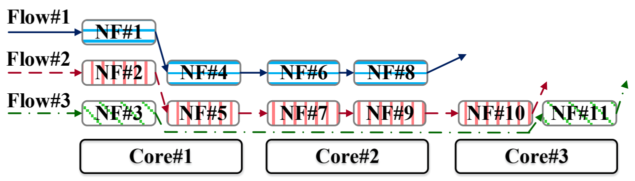


Fig. 1: The NFV platform provides flow-customized network function chain deployment for the edge, where multiple NFs can share the same CPU core.

*Corresponding author: Fangming Liu (fmliu@hust.edu.cn). This work was supported in part by the NSFC under Grant 61722206 and 61761136014 (and 392046569 of NSFC-DFG) and 61520106005, in part by National Key Research & Development (R&D) Plan under grant 2017YFB1001703, in part by the Fundamental Research Funds for the Central Universities under Grant 2017KFKJXX009 and 3004210116, in part by the National Program for Support of Top-notch Young Professionals in National Program for Special Support of Eminent Professionals.

istics of the edge, we meet three challenges to be solved. First, to make full use of edge resources, a flexible, light-weight service provisioning method is needed to initiate and terminate NFs, and a fine-grained resource management platform is needed to allocate the most cost-efficient resource utilization to each NF. Second, the flow of a network service often needs to be steered through a chain of NFs, known as *service chaining* [17], while NFs of heterogeneous flows can be deployed on the same edge server and even on the same CPU core. Fig. 1 depicts an example of this. These NFs will compete with each other for CPU and other resources, which will degrade the performance [18]. Besides, as we will disclose in Sec. II, CPU allocation and flow-level characteristics (e.g., flow rate, packet size and packet loss rate) have a great effect on performance (i.e., throughput and latency), which needs careful decisions on assigning edge resources (e.g., CPU quota) to each NF. Third, since edge flows typically exhibit great variations, real-time flow monitoring, elastic resource scaling up and down, and also flexible NF migration among cores are all needed so as to handle the real-time flow fluctuations.

Therefore, to integrally deal with these challenges, we propose *Finedge*, a dynamic, fine-grained and cost-efficient edge resource management platform for NFV network, which can assign the most cost-efficient CPU quota (i.e., CPU utilization proportion) to each NF while satisfying most flows' service level agreement (SLA). To address the first challenge, we implement Finedge on top of OpenNetVM platform [10], a DPDK-based NFV platform that provides dynamic packet steering, flexible service provisioning, load balancing and flow management, where DPDK is the Intel Data Plane Development Kit consisted of a collection of data plane libraries and NIC drivers for high-speed packet processing [19]. Especially, we run each NF in a Docker container instance, which is much more lightweight than a virtual machine (VM) and can be switched on and off very quickly.

To address the second challenge and the third challenge, we firstly want to find out the relationship between NF's performance and its resource allocation. Thus, we conduct empirical experiments with different flow-level characteristics (e.g., flow rate, packet size and packet loss rate), resource allocation and their performance (i.e., throughput and latency). By analyzing and summarizing the experimental results, we obtain several useful findings and insights as disclosed in Sec. II, which are used in designing the resource allocation and scaling strategy of Finedge. Specifically, Finedge can automatically tune the most suitable CPU core with the most cost-efficient CPU quota for each NF, considering its flow-level traffic characteristics and QoS requirements. Meanwhile, Finedge provides real-time flow monitoring, and it can dynamically adjust NF's processing CPU core and its CPU quota to adapt to flow rate variation. In finedge, we leverage Docker update command to implement efficient edge resource management without modifying the OSs internal scheduling mechanism. Through extensive evaluations, we verify that Finedge can manage the edge resources cost-efficiently when deploying NF chains for real-time heterogeneous flows.

The main contributions of this paper are as follows:

- We collect a series of useful findings and insights from empirical experiments, which disclose the effect of NFs' resource allocation and their flow-level characteristics on performance.
- We propose a dynamic, fine-grained and cost-efficient edge resource management platform, Finedge, for NFV network, which can automatically tune the most suitable CPU core and the most cost-efficient CPU quota for each NF, considering its flow-level traffic characteristics and QoS requirements.
- Finedge supports real-time flow monitoring, elastic resource reallocation, and also flexible NF migration among cores to handle the flow fluctuations.
- The evaluation results show that Finedge outperforms the default OS scheduler and other state-of-the-art resource management schemes with lower CPU quota and higher SLA satisfaction rate.

II. Motivation

In order to design a fine-grained edge resource management platform, we firstly conduct a series of empirical experiments on heterogeneous flows with different flow-level characteristics (i.e., flow rate and packet size), QoS requirements (i.e., latency and packet loss rate) and different resources assigned to each NF.

We use a testbed consisting of three commodity servers each equipped with two Intel(R) E5-2630 v3 8-core CPUs at 2.40 GHz, 128 GB memory, running Ubuntu SMP Linux kernel 3.13.0-100-generic. Each server has two Intel i350 1Gb NICs and two Dual-port Intel X520-DA2 10 Gb DPDK compatible NICs. Servers are connected in a back-to-back manner with two dual-port 10 Gbps DPDK compatible NICs to reduce switch overheads. We use MoonGen [20], a DPDK-based high-speed traffic generators for generating linear rate traffic.

A. NF's Flow-level Characteristics and CPU Allocation on Performance

To disclose the effect of NFs flow-level characteristics and CPU allocation on performance, we conduct a series of experiments with different flow rates, packet loss rates and packet sizes, as the CPU quota of one NF varies from 10% to 100%. We adopt a commonly-used software NF—`Basic Monitor`* in our experiments, where `Basic Monitor` (BM) prints information of arriving packets and then directly steers them out. The experiment results together with analyses and insights are as follows.

Packet Loss Rate. As plotted in Fig. 2, we measure the packet loss rates of flows with different flow rates (i.e., 1,000 Mbps and 9,000 Mbps) and the same packet size—1,400 Bytes. We discover that with more CPU quota allocated to BM, the packet loss rate decreases nearly linearly. Besides, to satisfy a certain packet loss rate bound, the higher flow rate is, the more CPU quota is needed.

*`Basic Monitor`: https://github.com/sdnfv/openNetVM/tree/develop/examples/basic_monitor

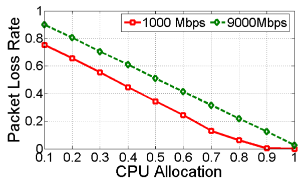


Fig. 2: The effect of CPU allocation on packet loss rate with different flow rates.

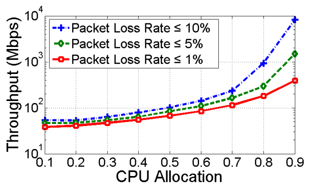


Fig. 3: The effect of CPU allocation on throughput with different packet loss rates.

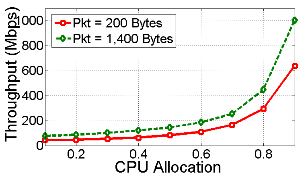


Fig. 4: The effect of CPU allocation on throughput with different packet sizes.

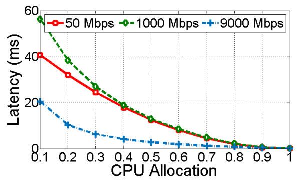


Fig. 5: The effect of CPU allocation on latency with different flow rates.

Throughput. As plotted in Fig. 3, we measure the maximal throughput of a BM with three different packet loss rates (i.e., 10%, 5% and 1%). We discover that: (1) at a certain packet loss rate, the maximal throughput of BM grows nearly exponentially as more CPU quota is allocated it; (2) at a certain CPU allocation to BM, the lower packet loss rate that BM's flow requires, the lower maximal throughput it gets.

These results can be explained as follows. Generally, NF's packets are processed in a roll polling manner, where each NF's RX ring will be checked in turn to see if there is any packet waiting to be processed. When an NF's RX ring is fulfilled with packets but it is not the NF's turn to process them, this NF cannot receive any more arriving packets and thus the arriving packets will be lost. When allocated a low proportion of CPU utilization, NF gets less CPU time slices to process its packets, which results in a higher packet loss rate. As more CPU quota is allocated to it, the packet loss rate decreases accordingly. However, due to the maximum capacity of NF's RX ring, when NF's flow rate grows large enough to fulfill its RX ring, packets will also be lost and consequently, the packet loss rate is increased.

Packet Size. We measure these two representative packet sizes (i.e., 200 Bytes and 1,400 Bytes) as the CPU quota of one NF varies from 10% to 100%, and the required packet loss rate is no more than 5%. Fig. 4 shows the experiment results, where Pkt is the packet size, TP is the throughput and CPU is the CPU quota allocated to the BM. The measurements show that: with the same CPU quota, the maximal throughput of BM with 1,400 Bytes packet size is always higher than it with 200 Bytes. Similar to the aforementioned results, we can also find the trend that the maximal throughput of BM does not grow linearly as more CPU quota is assigned, but nearly exponentially. The exponential trends are more obvious even with different packet sizes.

Latency. To disclose NF's CPU allocation on latency, we also measure the average latency with three different flow rates (i.e., 50 Mbps, 1,000 Mbps and 9,000 Mbps), as the CPU quota of one NF varies from 10% to 100%. As plotted in Fig. 5, we can see that (1) as the CPU quota increases, the end-to-end latency decreases accordingly, no matter with a large or small flow rate. This is because when CPU quota is low, the arrived packets cannot be disposed of in time until NF gets its the time slice. (2) When increasing the flow rate, latency is not continuously increased with the same CPU quota. For example, the latency of 50 Mbps flow is lower than that of 1000 Mbps flow, but is larger than that of 9000 Mbps. This is because the latency here is the average latency of packets that are not lost. When the flow rate is low, the RX ring will not be fulfilled very fast and the packet loss rate is not so high. However, these packets have to wait for its time slice to be processed. When the flow rate grows higher, it will collect more packets until it gets the time slice, thus the throughput will be higher. However, many packets with high latency will be dropped before the NF gets its required time slice. Thus, when the flow rate grows, the latency of packets becomes lower instead, since a large number of packets that are not dropped get the time slots quickly and do not have to wait long for being processed.

TABLE I: Throughput and CPU quota of NF chains

Flow Rate (Mbps)	NF#1		NF#2		NF#3		TH (Mbps)
	Type Name	CPU	Type Name	CPU	Type Name	CPU	
50	FW	2%	AE	40%	AD	40%	49.6
50	FW	2%	AE	35%	AD	45%	42.4
50	FW	2%	AE	45%	AD	35%	42.4
50	FW	7%	AE	35%	AD	40%	41.9
50	FW	7%	AE	40%	AD	35%	41.9
50	FW	1%	AE	41%	AD	40%	24
50	FW	1%	AE	40%	AD	41%	24.2

B. NF Chain's CPU Allocation on Performance

To explore the performance of NF chain, we adopt an NF chain consisting of three NFs—`Firewall`[†], `Aes Encrypt`[‡], `Aes Decrypt`[§]. In particular, `Firewall` (FW) drops or forwards packets based on longest prefix matching (LPM) rules specified in the rules.json file. `Aes Encrypt` (AE) encrypts UDP packets with the advanced encryption standard (AES) cypher and then forwards them to a specific NF destination or port. `Aes Decrypt` (AD) decrypts UDP packets with the AES cypher and then forwards them to a specific NF destination or the port. We measure the throughput and latency of this NF chain as the CPU quota of one NF varies from 10% to 100%. The experiment results together with analyses and insights are as follows.

Throughput. As plotted in Table I, we measure the throughput of the NF chain with different CPU allocation. The "CPU" in Table I represents the CPU quota assigned to each NF, the

[†]`Firewall`: https://github.com/sdnfv/openNetVM/tree/develop/examples/firewall

[‡]`Aes Encrypt`: https://github.com/sdnfv/openNetVM/tree/develop/examples/aes_encrypt

[§]`Aes Decrypt`: https://github.com/sdnfv/openNetVM/tree/develop/examples/aes_decrypt

TABLE II: Latency and CPU quota of NF chains

Flow Rate (Mbps)	NF#1		NF#2		NF#3		Latency (ms)
	Type Name	CPU	Type Name	CPU	Type Name	CPU	
50	FW	2%	AE	40%	AD	40%	3446.9
50	FW	2%	AE	81%	AD	40%	240.3
50	FW	2%	AE	40%	AD	81%	231.1
50	FW	43%	AE	40%	AD	40%	229.1
50	FW	2%	AE	61%	AD	60%	221.7
50	FW	2%	AE	60%	AD	61%	218.9
50	FW	23%	AE	60%	AD	40%	196.9
50	FW	23%	AE	40%	AD	60%	228.5
50	FW	3%	AE	60%	AD	60%	109
50	FW	15%	AE	54%	AD	54%	57.5

"TH" represents the throughput of the NF chain and the packet size is 200 Bytes. We discover that: (1) the throughput of an NF chain varies as the CPU quota assigned to each NF varies, where the total CPU quota of the three NFs is the same; (2) the throughput of an NF chain is maximal when the ratio of the CPU quotas of NFs in the chain reach a certain value. In further experiments, we discover that the certain value is determined by the type of NF and the packet size of its flow.

Latency. As plotted in Table II, we measure the latency of the introduced NF chain with different CPU allocation. The "CPU" in Table II represents the CPU quota assigned to each NF and the packet size is 200 Bytes. We discover that: (1) the latency of NF chain decreases apparently as more CPU quota is allocated to any NF in an NF chain; (2) the latency of an NF chain can be reduced to a minimum value by increasing the CPU quota of the NF with the minimal incremental CPU quota, where the minimal incremental CPU quota is the current CPU quota of the NF minus the CPU quota of the NF when its NF chain can meet the throughput and packet loss rate requirements; (3) when the total CPU quota is equal, the more similar the incremental CPU quota of each NF is, the lower the latency of NF chain can achieve.

In short, to design a fine-grained and cost-efficient edge resource management platform for NFV, we have to take account of all the mentioned factors including flow rate, packet size and QoS requirements, and figure out how to balance the trade-offs among them by using these experimental findings and insights.

III. System Design and Implementation

In this section, we begin with the system architecture and core components of Finedge. Then we introduce the system procedure of Finedge in detail with the core functions.

A. Architecture of Finedge

To capture the flow-level characteristics and QoS requirements, we design Finedge with four core components: Flow Director, Flow Table, NF Manager and Resource Table. The overall system architecture of Finedge is shown in Fig. 6, and the related flow steering and resource management tables of Finedge are plotted in Fig. 7.

Flow Table. Flow Table is a hash table that stores the hash value of a flow's packets (i.e., the Key of Flow Table) and

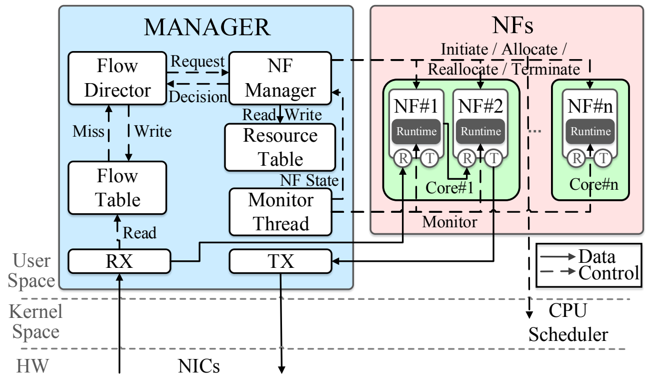


Fig. 6: The system architecture of Finedge. The full lines show packet steering, while the dashed lines are for system control.

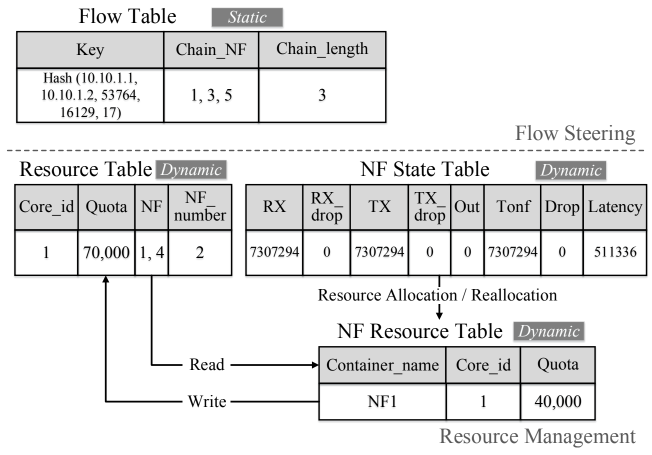
the information of the NF chain of each flow. The hash value of a packet is a quintuple including the source IP, destination IP, the port number of both source and destination and the IP protocol. For example, the flow table stores a flow as shown in Fig. 7, whose source IP is 10.10.1.1, destination IP is 10.10.1.2, source port number is 53764 and destination port number is 16129 and IP protocol is UDP-17. This flow has an NF chain consisted of three NFs.

Packet of the same hash value belongs to the same flow, requiring the same NF chain for steering and processing packets. Homogeneous flows (i.e., with the same NF service chain) can be merged into one if their SLAs are the same (i.e., same throughput, latency and packet loss rate requirements). Note that traffic in cloud and edge network usually consists of countless flow with low flow rate [11], by efficiently merging homogeneous flows, Finedge can efficiently improve the edge resource utilization and simplify the service deployment.

Flow Director. Based on the Flow Table look-up results, we design a Flow Director to make appropriate packet steering decisions. If a flow's NF chain is already deployed, the Flow Director will steer all its packet to its first flow-required NF according to the service id; if not, the Flow Director will inform the NF manager to create a new NF service chain and steer the new flow's packet to its first NF.

NF Manager. NF Mananger is the core function to implement NF assignment and fine-grained edge resource management. By jointly considering traffic characteristics, SLAs of heterogeneous flows and current resource allocation state, the NF Mananger can dynamically create a new service chain, automatically assign the most appropriate core and tune the most cost-efficient CPU quota for each NF. Besides, the NF Manager will be aware of the change of flow rate by acquiring the packet arrival rate of an NF chain from the Monitor in real time. Then the NF Manager will adjust the edge resource allocation to each NF in the flow-relevant NF chain. More details about flow monitoring, resource allocation and CPU core selection are stated in Sec. III-B.

Resource Table. The Resource Table stores the CPU core id and the CPU quota of each deployed NF. When assigning CPU core for a new created NF, the NF manager will first check the Resource Table to find out: (i) what kinds of NF(s) and



Flow Table Static

Key	Chain_NF	Chain_length
Hash (10.10.1.1, 10.10.1.2, 53764, 16129, 17)	1, 3, 5	3

Flow Steering

Resource Table Dynamic

Core_id	Quota	NF	NF_number
1	70,000	1, 4	2

NF State Table Dynamic

RX	RX_drop	TX	TX_drop	Out	Tonf	Drop	Latency
7307294	0	7307294	0	0	7307294	0	511336

Resource Allocation / Reallocation

NF Resource Table Dynamic

Container_name	Core_id	Quota
NF1	1	40,000

Read
Write
Resource Management

Fig. 7: The flow steering and resource management tables of Finedge.

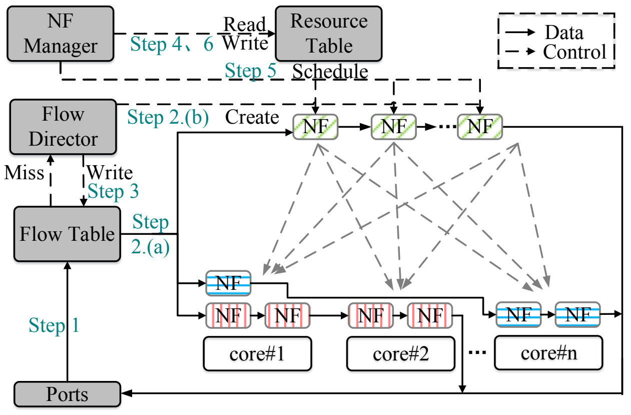


Fig. 8: The system procedure of Finedge.

how many of them are occupying a core, (ii) how much CPU quota is allocated to each NF, and (iii) how much CPU quota of each core is left for allocation. The NF Manager will then make corresponding decisions according to these messages.

As shown in Fig. 7, the Resource Table records the core ID, the CPU quota on the core, the number of NFs placed on the core and the NF instance id. For example, there are two NFs deployed on the core with ID equal to 1. The instance ID of two NFs are 1 and 4, respectively. The two NFs occupy a total CPU quota of 70,000 (the total CPU quota is 100,000). The CPU quota that has been allocated on the core is equal to the sum of the CPU quota allocated to all NFs on the core.

B. Procedure of Finedge

To support dynamic, fined-grained and cost-efficient edge resource management framework for edge-based NFV systems, we have to consider the edge resource limitation and high QoS/SLA requirements of edge services. Thus, we implement Finedge with three core functions: **(i) Real-time Flow Monitoring; (ii) Fine-grained, Cost-efficient CPU Quota Allocation and Scaling; and (iii) Dynamic NF Migration with Flexible CPU Core Selection.** The overall system procedure is shown in Fig. 8, involving six major steps:
Step 1: when a packet arrives at the ports, read its quintuple including IP protocol, IP address and port of both source and destination. Then check whether the hash value of the arriving packet hits in the Flow Table.
Step 2. (a): if the arriving packet hits in the Flow Table, transfer the packet to its first NF of the NF chain.
Step 2. (b): otherwise, create a new NF chain if the arriving packet does not hit in the Flow Table. Check the Resource Table for the current resource allocation state, then assign each NF to the CPU core with the highest remaining quota and allocate the default CPU quota.
Step 3: write the hash value and service ID of the arriving packet into the Flow Table.
Step 4: write the NF assignment results into the Resource Table and update the resource allocation state.
Step 5: tune the NF's CPU quota when the flow it servers violates SLA or the flow's flow rate decreases and even re-select a CPU core when the original core has no remaining CPU quota.
Step 6: update the NF reassignment results and resource allocation state.

Now we elaborate on the three core functions.

Real-time Flow Monitoring. To dynamically capture the SLA violation of each flow, we design and implement a flow monitoring module to monitor the real-time flow dynamics, including packet arrival rate, packet loss rate and average latency. To record these data, Finedge creates an NF running time table to store all the information in the shared memory of NFs and manager. The monitor module records and updates the data in this table, and calculates the packet arrival rate, packet loss rate and average latency at fixed time interval.

As plotted in Fig. 7, the NF flow table contains the key value, the destination NF's instance ID and the next action. NF state table records the total receiving and dropping packets from the RX RING named by RX and RX_DROP, the sending and dropping packets at the TX RING named by TX and TX_DROP and the total number of packets processing OUT, TONF and DROP. When the RX threads steer packets from ports or the last NF of an NF chain sends the processed packets to the NF's RX ring buffer successfully, the number of RX adds the number of packets. However, if the RX ring buffer is full, the packets are dropped and the number of RX_DROP adds the number of packets. If the NF sends processed packets to the next NF or ports successfully, the number of TX adds the number of packets, otherwise, the number of TX_DROP adds the number of packets. The monitor calculates the receiving and dropping packets rate from RX RING named by RX_PPS and RX_DROP_PPS and the sending and dropping packets at the TX RING named by TX_PPS and TX_DROP_PPS at regular intervals. The packet arrival rate of NF is equal to the sum of the RX rate and RX drop rate, and the flow's packet arrival rate is equal to the packet arrival rate of the first NF in the flow-relevant NF chain. The packet loss rate of NF is equal to the RX rate dividing the packet arrival rate of NF, and the flow's packet loss rate is approximately equal to multiplying the packet loss rate of each NF in the flow-relevant NF chain. Besides, the monitor can automatically calculate the average latency by making timestamps of packets. The NF resource table stores NF's container name, its running core ID and the CPU quota. NF Manager can dynamically adjust the resource allocation based on these tables.

Fine-grained, Cost-efficient CPU Quota Allocation and Scaling. Since we implemented Finedge on top of OpenNetVM platform [10], while its NFs run in a roll polling manner, which means it will alternately check NF's RX ring to see whether there is any unprocessed packet. Unlike the interrupt manner, which will assign the resource to other NFs once all the packets are processed, NFs in OpenNetVM will keep occupying the CPU resource once allocated to them. So we need to implement a fine-grained, cost-efficient edge resource allocation considering the flow-level traffic characteristics (i.e., packet size, packet sending rate) and SLAs (i.e., throughput, latency, packet loss rate) of heterogeneous flows.

In fact, there are two ways to allocate CPU resources for each NF. One approach is to modify the OS schedulers, i.e., changing the NF's CPU processing priority, or sleeping and waking up NF based on the queue length of the packets waiting to be processed. However, this will bring in extra system calls, which consume extra CPU cycles and increase the communication latency. Considering the strict QoS requirements of edge requests, we choose the other way, which is to leverage Docker update command, a quick and efficient container technology for Linux, to limit the resources that containers consume.

Based on the extensive empirical measurement results, we designed a dynamic, fine-grained CPU quota allocation and scaling mechanism in NF Manager to meet as many flows' SLA as possible. This mechanism includes two main algorithms—resource scaling up algorithm and resource scaling down algorithm. The NF manager will firstly check whether the packet loss rate and the latency of flow meet the flow's SLA. If the requirement on packet loss rate or the latency of flow is violated, the NF Manager will call the resource scaling up algorithm to increase the CPU quota of the corresponding NF. Otherwise, the NF manager will check whether the packet arrival rate of this flow decreases; if it decreases, the NF manager will call resource scaling down algorithm to decrease the CPU quota of that NF.

In particular, there are two ways for allocating the CPU quota by using Docker update command. One is to run the command `docker update --cpu-shares` to assign the CPU quota for containers, where the CPU quota is not the absolute quota of CPU processing time slices but the CPU sharing proportion among containers. Considering the limitation of edge resources, we choose the other way, using `docker update --cpu-quota` to assign the CPU processing time for containers. Specifically, the setting value of `cpu-quota` refers to the quota of CPU processing time slices for each container in a CPU processing period. The CPU processing period is defaulted by 100 ms.

In our implementation, we leverage Docker update command which does not need to modify the OS's internal scheduling mechanism. Specifically, we use `docker update --cpu-quota` command to assign the CPU quota, which can fully save and reserve the CPU resource meanwhile providing performance guarantees for edge network.

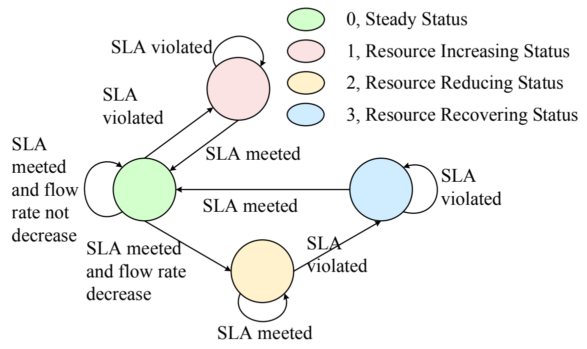

0, Steady Status
1, Resource Increasing Status
2, Resource Reducing Status
3, Resource Recovering Status
SLA violated
SLA violated
SLA meeted
SLA meeted and flow rate not decrease
SLA meeted
SLA violated
SLA meeted and flow rate decrease
SLA violated
SLA meeted

Fig. 9: The NF chain state transition diagram.

Now we elaborate on how to tune the CPU quota for each NF to meet each flow's SLA. As shown in Fig. 9, each NF chain has four states:
State 0: steady state. It means that the current CPU allocation can meet this flow's SLA and the NF Manager does not need to allocate more CPU quota to this flow's NF chain.
State 1: resource increasing state. It represents that this flow's NF chain needs more CPU quota to satisfy its SLA.
State 2: resource reducing state. It means that the flow's NF chain occupies some redundant CPU resources which can be recycled.
State 3: resource recovering state. In this state, flow's NF chain also needs more CPU quota to satisfy its SLA. Different from *State 1*, this state means that the CPU resources assigned to this flow's NF chain have been reduced too much, which actually needs some more CPU resources to satisfy its SLA.

In the initial allocation, the NF manager initiates the CPU quota with a fixed default value to each newly created NF. In Finedge, this default value is set to 5,000, which means that CPU utilization is 5%. The newly created NF chain's initial state is state 0. The NF manager will check each flow's packet loss rate and the latency in a fixed time interval. If the packet loss rate or the latency of flow violates, the NF manager will call resource scaling up algorithm to increase the CPU quota of the flow's NF chain. In resource scaling up algorithm, when the packet loss rate of flow is higher than the packet loss rate of flow's SLA, the CPU quota will be increased by a fixed value to an NF (which has the minimum ratio of CPU quota and weight in its NF chain), if the CPU core NF occupy has enough spare CPU resources. The weight is decided by the NF type and the flow's packet size, which is introduced in Sec. II. When the packet loss rate of a flow meets the requirement but the latency does not, the algorithm will select an NF with the least extra adding quota and then add its quota. The extra adding quota of an NF is the increased CPU quota of the NF after the packet loss rate of flow has been reduced to satisfy the packet loss rate requirement. In Finedge, if a flow's NF chain is at *State 0* or *State 1*, the increment value is 5,000 and this NF chain's state will be transited into *State 1*, otherwise, the increment value is 1,000 and this NF chain's state will be transited into *State 3*. If the CPU core does not

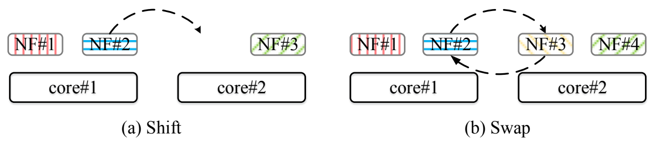


Fig. 10: Two manners of NF migration across cores.

have enough CPU resources, all the remaining CPU resources will be allocated to this NF. If the flow's SLA is satisfied, its NF chain's state is transited into *State 0* and the current packet arrival rate will be recorded.

Furthermore, the NF Manager will check whether the packet arrival rate of this flow decreases. If so, the NF Manager will call the resource scaling down algorithm to decrease the NF's CPU quota. In this algorithm, the CPU quota of an NF (which has the maximal ratio of CPU quota and weight in NF chain) will be decreased by the default value set by 1,000 and the NF chain's state will be transited into *State 2*. The NF Manager will decrease the CPU quota of this flow's NF chain until this flow's SLA is just not satisfied. Then the NF chain enters the resource recovering state and the NF Manager increases its CPU quota to meets its SLA.

Our proposed algorithm has the characteristics of fast increasing and slow decreasing, since the step of CPU quota increment is 5,000, while the step of CPU quota decrement and recovery is 1,000. This setting can short down the procedure time for resource allocation and maximize resource reservation, which shows good benefits in dealing with QoS-sensitive edge requests as we will show in the evaluation section. These default values can also be adjusted according to the edge service requirements.

Dynamic NF Migration with Flexible CPU Core Selection. Now we elaborate on how to select a core for each NF according to its resource demand. Note that currently, there are a number of existing works dealing with NF assignment, or precisely NF placement problem [21]–[24]. Differing from them, our approaches can effectively migrate NF and dynamically adjust its CPU core according to its CPU quota in a single node, which can be combined with those existing approaches in further work. We use `docker update --cpuset-cpus` command to assign CPU core to NF, which doesn't need to consider NF's state migration.

In the initial assignment, the NF Manager assigns an NF to the core with maximum spare CPU quota. When the CPU quota of NF needs to be increased but the core has no spare CPU resources, the NF Manager will migrate the NF to another core. As shown in Fig. 10, there are two manners of NF migration across cores:

Shift. This action means to reassign an NF from a "fulfilled" core to a "spare" core. The NF Manager will reassign the NF to the core with maximum spare CPU quota which should also be higher than NF's current CPU utilization.

Swap. This action means to exchange the assignment of two NFs on different cores. If the core with maximum spare CPU quota does not have enough CPU resources, the NF Manager will select an NF with the maximum CPU quota (which does not exceed the current CPU quota of reassigned NF) to exchange the core with reassigned NF.

IV. Evaluation

In this section, we first introduce the evaluation testbed and detailed experimental setup. Then we evaluate Finedge's performance in three different cases: two flows with the same NF chain and different flow-level characteristics, two flows with different NF chains and dynamic flow fluctuations, and multiple heterogeneous flows. We compare Finedge with the default OS scheduler and other state-of-the-art resource management schemes.

A. Testbed and Experimental Setup

Our experimental testbed is based on fifteen servers, each equipped with two Intel E5-2660 v3 10-core CPUs at 2.60 GHz, 160GB ECC memory, running Ubuntu SMP Linux kernel 3.13.0-100-generic. Each server has two Onboard Intel i350 1Gb NIC and two Dual-port Intel X520-DA2 10 Gb DPDK compatible NIC. In our experiments, servers were connected back-to-back with two dual-port 10 Gbps DPDK compatible NICs to reduce switch overheads. Servers are divided into two classes, fourteen for traffic generation and one for accepting and processing packets.

We adopt six commonly-used software NFs—BM, FW, AE, AD, `Simple Forward`¶, and `Flow Tracker`‖ in our experiments, where `Simple Forward` (SF) forwards packets to a specific destination, and `Flow Tracker` (FT) stores and displays information about incoming flows. We have make some modification to the codes in AD to make it forwarding packets to the port rather than next NF. We compare the performance of Finedge with the following two schemes in each experiment.

- **LBS.** A Load-Based Scheduler as implemented in NFVnice [12].
- **OS.** The OSs default Scheduler with an equal sharing of CPU utilization.

B. Performance Evaluation Results

We evaluate Finedge's overall performance in three different cases. In the first case, we set two flows which both need FW and BM in their NF chain, and we only change one flow characteristic among the four characteristics (i.e., flow rate, packet loss rate requirement, packet size and latency requirement). In the second case, we set two flows with different NF chains, i.e., one flow consisted of an FW and a BM, while the other consisted of an AE and an AD. The flow rate of both flows varies dynamically in real time. In the above two cases, the NFs of two heterogeneous flows are set to have two available CPU cores to process their packets. In the last case, we set 14 and 56 heterogeneous flows respectively, whose NF chains consist of two to five NFs. These NFs are

¶`Simple Forward`: https://github.com/sdnfv/openNetVM/tree/develop/examples/simple_forward

‖`Flow Tracker`: https://github.com/sdnfv/openNetVM/tree/develop/examples/flow_tracker

TABLE III: Two flows with different flow rates

Scheme	FR (Mbps)	NF#1		NF#2		TH (Mbps)	LA (ms)
		TN	CPU	TN	CPU		
OS	200	FW	50	BM	50	200	10.5
	3800	FW	50	BM	50	2008	58.9
LBS	200	FW	5	BM	5	181	533.2
	3800	FW	95	BM	95	3800	3.2
Finedge	200	FW	10	BM	10	200	46.1
	3800	FW	90	BM	90	3629	27.2

TABLE IV: Two flows with different packet loss rates

Scheme	SLA PLR	NF#1		NF#2		TH (Mbps)	LA (ms)
		TN	CPU	TN	CPU		
OS	0.01	FW	50	BM	50	1970	25.5
	0.2	FW	50	BM	50	1966	31.4
LBS	0.01	FW	50	BM	50	1964	29.3
	0.2	FW	50	BM	50	1979	31.4
Finedge	0.01	FW	55	BM	55	2100	15
	0.2	FW	45	BM	45	1761	54.6

TABLE V: Two flows with different packet sizes

Scheme	PS (Byte)	NF#1		NF#2		TH (Mbps)	LA (ms)
		TN	CPU	TN	CPU		
OS	200	FW	50	BM	50	800	14.1
	64	FW	50	BM	50	652	56.5
LBS	200	FW	50	BM	50	800	14.3
	64	FW	50	BM	50	658	57.2
Finedge	200	FW	25	BM	25	800	27.3
	64	FW	75	BM	75	800	9.5

from six different types and they are set to use ten available CPU cores.

Case 1: Performance of Two Flows with the Same NF Chain and Different Flow-level Characteristics. To illustrate the performance influence of the four flow characteristics, we successively evaluate the throughput and average latency of the two flows in four setups (e.g., different flow rates, different packet loss rate requirements, different packet size and different latency requirements).

In the first setup, we set two flows with different flow rates, i.e., 200 Mbps and 3,800 Mbps, while they both require less than 5% packet loss rate, maximum 100 ms latency and 200 Bytes packet size. We use 1 minus the ratio of the throughput and the flow rate to calculate the packet loss rate. The experimental results are listed in Table III. We can conclude that the packet loss rate of flow with 200 Mbps is more than 9.5% and the latency exceeds 100 ms when using LBS. The packet loss rate of the flow with 3800 Mbps is even 47.2% when using OS. Both of these two strategies violate the SLA of one of these two flows. Because the maximal throughput of NF does not grow linearly as more CPU utilization proportion is assigned, which we have demonstrated in Sec. II. In short, Finedge can satisfy both throughput and latency demands, i.e., with 0% and 4.5% packet loss rate for 200 Mbps and 3,800 Mbps flow, respectively.

In the second setup, we set two flows with different packet loss rate requirements, i.e., 1% and 20%, while both of their maximum latency requirements are 100 ms with 2,100 Mbps flow rate and 200 Bytes packet size. As listed in Table IV, the packet loss rate of flow with more strict packet loss rate requirement is more than 1% when using LBS and OS

TABLE VI: Two flows with different latency requirements

Scheme	SLA LA (ms)	NF#1		NF#2		TH (Mbps)	LA (ms)
		TN	CPU	TN	CPU		
OS	100	FW	50	BM	50	200	14.9
	10	FW	50	BM	50	200	15.3
LBS	100	FW	50	BM	50	200	13.8
	10	FW	50	BM	50	200	14.4
Finedge	100	FW	10	BM	10	200	45.8
	10	FW	85	BM	85	200	8.5

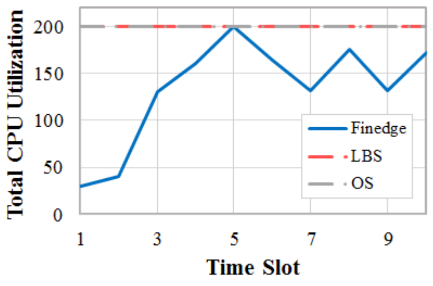


Fig. 11: Total CPU utilization of the two NF chains

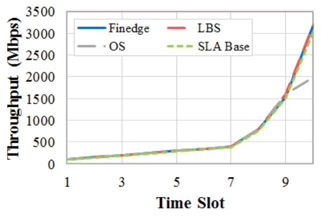
(i.e., 6.5% and 5.8%), while Finedge can satisfy this strict requirement.

In the third setup, we set two flows with different packet sizes, i.e., 64 Bytes and 200 Bytes, while both of their packet loss rate requirements are 5%, maximum latency requirements are 100 ms and flow rates are both 800 Mbps. Similarly, from Table V, we can see that the packet loss rates of flow with 64 byte are 18.5% and 17.8% when using LBS and OS, while Finedge can achieve 0% packet loss rate.

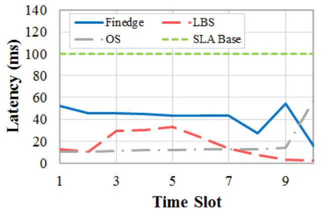
In the last setup, we set two flows with different latency requirements, i.e., 10 ms and 100 ms, they both require less than 5% packet loss rate with 200 Mbps flow rate and 200 Bytes packet size. As listed in Table VI, the latency of the first flow both exceed 10 ms (i.e., 14.4 ms and 15.3 ms) when using LBS and OS, while Finedge can achieve 8.5 ms latency.

In short, as compared with other strategies, Finedge can automatically assign the most suitable CPU allocation to each NF while assuring both heterogeneous flows' SLA requirements.

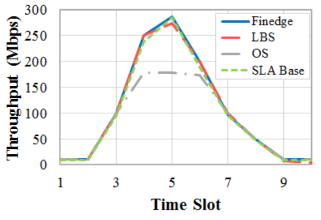
Case 2: Performance of Two Flows with Different NF Chains and Dynamic Flow Fluctuations. In this case, the flow rate of flow A increases from 100 to 3,200 Mbps as the time interval increases, while the flow rate of flow B dynamically increases from 10 to 300 Mbps and then drops to 10 Mbps. Flow A's NF chain consists of FW and BM with 200 Bytes packets, while flow B's NF chain consists of AE and AD with 64 Bytes packets. The packet loss rate requirements of both flows are 5% and the latency requirements are 100 ms. As plotted in Fig. 12, both OS and LBS can not satisfy the packet loss rate and latency requirements as the flow rate changes. In particular, when the flow rate of flow B increases to 250, 300 Mbps and decreases to 200 Mbps, OS can not satisfy both the packet loss rate and latency requirements. Differently, Finedge can always satisfy the SLA requirements even under dynamic flow fluctuations. Due to the complex trade-offs among different flow characteristics, CPU resource demand and flow's performance, taking only one characteristic



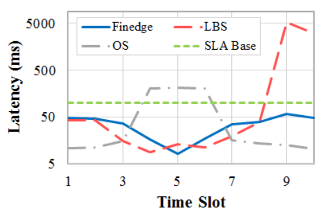
(a) Throughput



(b) Latency

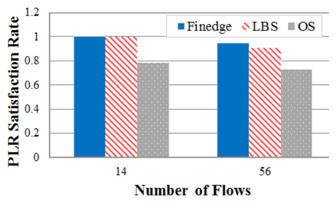


(c) Throughput

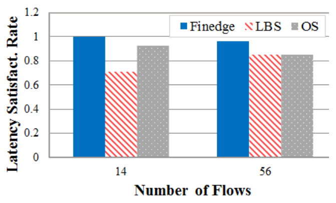


(d) Latency

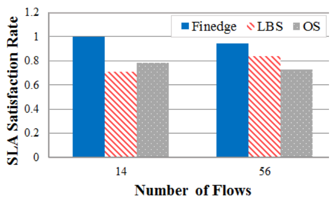
Fig. 12: The performance of Finedge and other strategies for two fluctuating flows. The throughput and latency of flow A are plotted in (a) and (b) while the throughput and latency of flow B are plotted in (c) and (d).



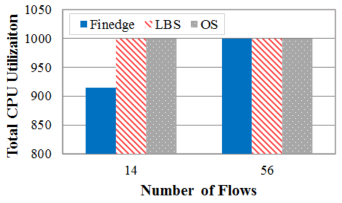
(a) Packet loss rate satisfaction rate



(b) Latency satisfaction rate



(c) SLA satisfaction rate



(d) Total CPU utilization

Fig. 13: The performance of Finedge and other strategies for multiple heterogeneous flows.

into consideration will naturally cause flow's SLA violation. In addition, as plotted in Fig. 11, although the latency of flow A in Finedge is higher than it in OS and LBS when the time slot is less than 9, it is always lower than SLA base since the goal of Finedge is to use less CPU resources to provide performance guarantee for flows rather than to minimize the latency or the total latency of all flows. We can see that Finedge can not only provide flow-level performance guarantee in real time, but also utilize the most cost-efficient CPU quota, while LBS and OS both occupy all the CPU quota. The remaining CPU quota in Finedge can be used for fault tolerance and flexible scaling up and down, which can be effectively applied to the resource-limited edge servers.

Case 3: Performance of Multiple Heterogeneous Flows. To evaluate the effectiveness of Finedge in the case of multiple heterogeneous flows, we set 14 and 56 heterogeneous flows in real time. As plotted in Fig. 13, Finedge can reserve about 915 CPU quota for assigning 14 flows while providing the 100% flow-level performance guarantee as compared with LBS and OS. Although the Finedge can not satisfy all flow's SLA when dealing with 56 flows, its SLA Satisfaction rate, as well as packet loss rate satisfaction rate and latency requirement satisfaction rate, are all higher than other strategies, i.e., 94.6%. 94.6% and 96.4% for Finedge, 83.9%, 91.1% and 85.7% for LBS, and 73.2%, 73.2% and 85.7% for OS. In short, Finedge can greatly reduce the SLA violations and provide flow-level performance guarantee for heterogeneous flows while using the lowest CPU quota.

V. Related Work

NFV Platform. A growing number of NFV platforms have been developed in recent years, such as E2 [8], NetVM [25], OpenNF [9], OpenNetVM [10], Flurries [11], NFP [13] and FERO [14]. With the development of these NFV platforms, efficient resource management and NF deployment can be provided for clouds/datacenters and the edge.

For instance, E2 [8] is a widely-used, comprehensive NFV orchestration framework to manage NF and distribute, but its software switch component, SoftNIC [26] requires at least one dedicated CPU core for traffic dispatching and steering. NetVM [25] is a DPDK-based platform for flexible network service deployment, which can run complex NFs at 10 Gbps line-speed in COST servers, generally in VMs. OpenBox [27] is a software-defined framework which merges similar packet processing element of NFs into one to reduce redundancy. OpenNetVM [10] is a scalable packet processing framework that provides dynamic packet steering, flexible flow management, efficient load balancing and service name abstractions, where NFs run as standard user-space processes inside Docker containers. Flurries [11] is a Docker-based NFV platform supporting large numbers of short-lived lightweight NFs and per-flow customization of processing and scheduling. NFP [13] implements NF parallelism on top of OpenNetVM and FERO [14] is a fast and efficient resource orchestrator for a data plane built on Docker and DPDK, which supports high performance and flexible programmability and control.

NF Scheduling and Resource Management. Some efficient NF management platforms arise, which enhance the NF assignment and state control along with appropriate resource scheduling. The resource provisioning strategy in Stratos [28] mainly captures the OS-level statistics but not specifies how to use them to detect and handle flow overload. The authors in [29] analyze the performance of applications and then allocate processors to them in shared-memory multiprocessor systems. NFV-RT [30] provides timing guarantees by dynamically placing and migrating NFs. Wang et al. [31] propose a multi-resource load balancing (MRLB) mechanism to improve the resource utilization efficiency. Jin et al. [32] optimize the resource utilization of both edge servers and physical links under the latency limitations. PSPAT [33] proposes a packet scheduling mechanism to support high packet rate with isolation and dependable service guarantees. PIFO [34] is a programmable packet scheduler, where the scheduling algorithms can be programmed into a switch without requiring hardware redesign. NFVnice [12] is a user-space NF scheduling and service chain management framework for providing fair and dynamic resource scheduling capabilities. However, NFVnice mainly considers the flow rate, regardless of other flow-level characteristics like packet loss rate, which can exacerbate dynamic bottlenecks and may cause violations on SLAs [11]. Arachne [35] is a core-aware thread management system that provides both low latency and high throughput, but it does not take account of the flow-level characteristics and dynamics, which is difficult to be adopted to edge-based NFV networks.

Different from these previous works, considering the limitation of edge resources and heterogeneity of flows, Finedge can provide dynamic, fine-grained edge resource management for NFV network, with the most cost-efficient CPU quota allocation and the highest SLA satisfaction rate.

VI. Conclusion

In this paper, we propose Finedge, a dynamic, fine-grained and cost-efficient edge resource management framework for NFV network. By summarizing the insights from empirical experiments, we implement Finedge with careful considerations on NF's flow-level characteristics, QoS requirements and CPU quota allocation on performance. Finedge provides real-time flow monitoring, elastic resource scaling up and down, and also flexible NF migration among cores to adapt to the flow fluctuations. Finedge is implemented on top of OpenNetVM platform, where NFs are deployed in lightweight Docker containers. Evaluation results validate that Finedge can efficiently handle heterogeneous flows with the lowest CPU quota and the highest SLA satisfaction rate as compared with the default OS scheduler and other state-of-the-art resource management schemes.

References

[1] S. Chen, L. Jiao, L. Wang, and F. Liu, "An online market mechanism for edge emergency demand response via cloudlet control," in *IEEE INFOCOM 2019*, pp. 2566–2574.

[2] J. Meng, H. Tan, C. Xu, W. Cao, L. Liu, and B. Li, "Dedas: Online task dispatching and scheduling with bandwidth constraint in edge computing," in *IEEE INFOCOM 2019*, pp. 2287–2295.

[3] M. Satyanarayanan, "The emergence of edge computing," *IEEE Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[4] B. Yi, X. Wang, K. Li, S. K. Das, and M. Huang, "A comprehensive survey of network function virtualization," *Elsevier Computer Networks*, vol. 133, pp. 212–262, 2018.

[5] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys Tutorials*, vol. 18, no. 1, pp. 236–262, 2016.

[6] G. Castellano, F. Esposito, and F. Risso, "A distributed orchestration algorithm for edge computing resources with guarantees," in *IEEE INFOCOM 2019*, pp. 2548–2556.

[7] J. Xie, C. Qian, D. Guo, M. Wang, S. Shi, and H. Chen, "Efficient indexing mechanism for unstructured data sharing systems in edge computing," in *IEEE INFOCOM 2019*, pp. 820–828.

[8] S. Palkar, L. Chang, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: a framework for NFV applications," in *ACM SOSP 2015*, pp. 121–136.

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

[10] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. Ramakrishnan, and T. Wood, "OpenNetVM: A platform for high performance network service chains," in *Proceedings of the 2016 workshop on Hot topics in Middleboxes and Network Function Virtualization*. ACM, 2016, pp. 26–31.

[11] Z. Wei, J. Hwang, S. Rajagopalan, K. K. Ramakrishnan, and T. Wood, "Flurries: Countless fine-grained nfs for flexible per-flow customization," in *ACM CoNEXT 2016*, pp. 3–17.

[12] S. G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, T. Wood, M. Arumaithurai, and X. Fu, "Nfvnice: Dynamic backpressure and scheduling for NFV service chains," in *ACM SIGCOMM 2017*, pp. 71–84.

[13] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu, "NFP: Enabling network function parallelism in NFV," in *ACM SIGCOMM 2017*, pp. 43–56.

[14] B. Sonkoly, M. Szabó, B. Németh, A. Majdán, G. Pongrácz, and L. Toka, "Fero: Fast and efficient resource orchestrator for a data plane built on docker and dpdk," in *IEEE INFOCOM 2018*, pp. 243–251.

[15] G. P. Katsikas, T. Barbette, D. Kostic, R. Steinert, and G. Q. Maguire Jr, "Metron: NFV service chains at the true speed of the underlying hardware," in *USENIX NSDI 2018*, pp. 171–186.

[16] X. Fei, F. Liu, H. Xu, and H. Jin, "Adaptive vnf scaling and flow routing with proactive demand prediction," in *IEEE INFOCOM 2018*, pp. 486–494.

[17] X. Li, X. Wang, F. Liu, and H. Xu, "DHL: Enabling flexible software network functions with fpga acceleration," in *IEEE ICDCS 2018*, pp. 1–11.

[18] C. Zeng, F. Liu, S. Chen, W. Jiang, and M. Li, "Demystifying the performance interference of co-located virtual network functions," in *IEEE INFOCOM 2018*, pp. 765–773.

[19] DPDK. (2019) Data Plane Development Kit (DPDK). [Online]. Available: https://dpdk.org

[20] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," in *ACM/USENIX IMC 2015*, pp. 275–287.

[21] Y. Xiao, Q. Zhang, F. Liu, J. Wang, M. Zhao, Z. Zhang, and J. Zhang, "NFVdeep: Adaptive online service function chain deployment with deep reinforcement learning," in *IEEE/ACM IWQoS 2019*, pp. 1–10.

[22] Q. Zhang, F. Liu, and C. Zeng, "Adaptive interference-aware vnf placement for service-customized 5g network slices," in *IEEE INFOCOM 2019*, pp. 2449–2457.

[23] X. Fei, F. Liu, H. Xu, and H. Jin, "Towards load-balanced vnf assignment in geo-distributed nfv infrastructure," in *IEEE/ACM IWQoS 2017*, pp. 1–10.

[24] Q. Zhang, Y. Xiao, F. Liu, J. C. S. Lui, J. Guo, and T. Wang, "Joint optimization of chain placement and request scheduling for network function virtualization," in *IEEE ICDCS 2017*, pp. 731–741.

[25] J. Hwang, K. K. Ramakrishnan, and T. Wood, "NetVM: high performance and flexible networking using virtualization on commodity platforms," *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, pp. 34–47, 2015.

[26] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "SoftNIC: A software NIC to augment hardware," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-155*, 2015.

[27] A. Bremler-Barr, Y. Harchol, and D. Hay, "Openbox: A software-defined framework for developing, deploying, and managing network functions," in *ACM SIGCOMM 2016*, pp. 511–524.

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

[29] J. Corbalan, X. Martorell, and J. Labarta, "Performance-driven processor allocation," *IEEE Transactions on Parallel Distributed Systems*, vol. 16, no. 7, pp. 599–611.

[30] Y. Li, L. T. X. Phan, and B. T. Loo, "Network functions virtualization with soft real-time guarantees," in *IEEE INFOCOM 2016*, pp. 1–9.

[31] T. Wang, H. Xu, and F. Liu, "Multi-resource load balancing for virtual network functions," in *IEEE ICDCS 2017*, pp. 1322–1332.

[32] P. Jin, X. Fei, Q. Zhang, F. Liu, and B. Li, "Latency-aware vnf chain deployment with efficient resource reuse at network edge," in *IEEE INFOCOM 2020*.

[33] L. Rizzo, P. Valente, G. Lettieri, and V. Maffione, "Pspat: software packet scheduling at hardware speed," *Elsevier Computer Communications*, vol. 120, pp. 32–45, 2018.

[34] A. Sivaraman, N. Mckeown, S. Subramanian, M. Alizadeh, and S. Katti, "Programmable packet scheduling at line rate," in *ACM SIGCOMM 2016*, pp. 44–57.

[35] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout, "Arachne: core-aware thread management," in *USENIX OSDI 2018*, pp. 145–160.