FlexNFV: Flexible Network Service Chaining with Dynamic Scaling

Xincai Fei, Fangming Liu*, *Senior Member, IEEE,* Hai Jin, *Fellow, IEEE,* and Bo Li, *Fellow, IEEE*

***Abstract*—Efficiently processing packets through network service chains requires instantaneous regulatory mechanism while incorporating traffic fluctuations. We propose FlexNFV, a flexible network service chaining framework designed to provide automatic and efficient Network Function (NF) scaling capabilities on Network Function Virtualization (NFV) platforms. The FlexNFV framework periodically monitors each NF load on a service chain, which can be accurately measured by the combination of different per-packet processing costs of different NF types, traffic characteristics and queues between NFs. Based on the chain-level information, our proposed FlexNFV performs timely scaling of NFs to evenly distribute the load among available instances, thereby avoiding performance degradation such as packet drops. By abstracting the implementation logic inside the NF runtime, NF scaling can be triggered using efficient and separate message channels for inter-process communication, which only requires minor modifications to existing NFV frameworks. We build and implement FlexNFV upon a DPDK-based NFV framework. Our testbed experiments show that our proposed FlexNFV: 1) improves service chain performance by a factor of two via matching the required load, when compared to the existing NFV framework; 2) enables dynamic NF scaling adapting to real-time traffic demand; and 3) achieves 17.5% better performance than NFVnice, the state-of-the-art NFV platform, under the same settings.**

***Index Terms*—NFV, service chaining, dynamic scaling.**

I. Introduction

THE deployment of network functions or middleboxes is ubiquitous in data center and enterprise networks [1]. Middleboxes provide a wide range of network functionalities, such as firewalls for ensuring security and WAN optimizers for improving performance. Network Function Virtualization (NFV) is a promising paradigm to implement middleboxes in software with the form of virtualized network functions (NFs) by employing commercial-off-the-shelf (COTS) hardware, thus enabling cost savings and service flexibility [2]. Generally in NFV, packets are often required to go through multiple NFs in a particular order, which is also known as *service chaining*.

Since running software-based NFs through the network stack incurs costly performance degradation, it is necessary to optimize the performance of service chains in order to meet their forwarding requirements. As such, many NFV platforms have been proposed to provide high-performance packet processing, by leveraging user space I/O frameworks such as DPDK [3] and netmap [4]. In fact, these high-performance I/O frameworks are designed for standalone applications (or to say NFs) not directly for service chains, in which traffic-steering techniques make it possible for the right traffic going through the right NFs. Even though these NFV platforms are capable of processing packets close to the line rate, service chain performance cannot be guaranteed without a proper regulatory mechanism, since any bottleneck NF can incur packet drops when arrival traffic is dynamically changed. An NFV platform should be able to create new instances for bottleneck NFs according to the real-time traffic demand. The regulatory mechanism, in brief, is to *perform dynamic scaling for right NFs at the right time*.

Although NF scaling has been explored in the literature [5], [6], [7], we argue that dynamic NF scaling in a service chain behaves differently from monolithic NFs. For service chains, due to processing heterogeneity of different NF types [1], satisfying the processing requirement of a single NF does not necessarily imply satisfying the requirement of other NFs. NFs should be handled in different ways in that the traffic flowing into a service chain imposes different loads on NFs.

However, it is challenging to fulfill this requirement since it involves a combination of chain-level parameters such as the NF load, traffic arrival rate and the queues between NFs. If the processing capacities of NFs are lower or higher than their processing requirements, the NFV platform should be capable of performing dynamic scaling for each related NF, while others keep running normally. On one hand, the NF load that depends on both the per-packet processing cost of the NF and traffic arrival rate, should be accurately computed and rapidly updated, in order to detect the overload or underload status of the NF. On the other hand, the length of the queues between each NF pair dictates the next work that the NF needs to process. If the queue is full in a certain period of time, it means that the pending packet processing could lead the NF to be overloading. In addition, an NF instance can also be detected as underload status, when the remaining processing capabil-

*The corresponding author: Fangming Liu (fmliu@hust.edu.cn). This work was supported in part by the NSFC under Grant 61722206 and 61761136014 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, in part by RGC GRF grants under the contracts 16206417 and 16207818, and a Guangzhou Development Zone Project Grant 2017GH23.

Xincai Fei, Fangming Liu, and Hai Jin are with 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, 1037 Luoyu Road, Wuhan, 430074, China.

Bo Li is with the department of Computer Science and Engineering, Hong Kong University of Science and Technology.

[1]Some lightweight NFs may just require an inexpensive header match, but others may perform complex encryption algorithms.

ities of other available instances are sufficient to satisfy the current workload even if the instance is turned into blocking status. Note that when an NF gets blocked, it stops receiving, processing and transmitting packets, while providing power savings for the NFV platform. An efficient and flexible NFV platform should make automatic and rapid scaling decisions in order to cope with the changing network workloads.

We propose FlexNFV, an NFV regulatory framework to provide automatic and efficient NF scaling capabilities for service chains. FlexNFV performs dynamic scaling according to the global information on the service chain. Specifically, it makes scaling decisions based on a variety of factors such as the NF processing cost, traffic arrival rate, the queues between NFs and the time duration of NF status (e.g., NF overloaded status). In particular, an NF is immediately turned into a power-efficient running mode when it gets blocked, as described in the section of system optimizations. The other benefit of NF blocking lies in when the running instance gets overloaded again in the future, the blocked instances of this type can be readily awakened to rerun normally rather than respawning new instances on other CPU cores, which takes time and incurs delay. FlexNFV makes the following contributions:

- Rapidly deploying the required NF instances to accommodate the traffic rate changes, in order to prevent dropping packets at bottleneck NFs and improve performance, along the entire service chain.
- Abstracting the logic of dynamic scaling inside the FlexNFV framework as a library, to automatically perform NF scaling via exploiting efficient message channels for inter-process communication.
- Designing a launching avoidance strategy to effectively reuse the blocked NF instances when the processing demands can be covered, by reinstating the blocked instances.
- Introducing a set of optimizations such as power-aware management and conflict elimination of message queues, via a power management library provided by DPDK and the allocation of separate queues for related NFs, respectively.

We have implemented FlexNFV upon OpenNetVM [8], a DPDK-based NFV platform that runs NFs in isolated secondary processes or containers, and ensures steering packets through a service chain with zero-copy packet delivery. Our evaluation shows that FlexNFV boosts the throughput by 2×, and dramatically reduces the packet drop rate, when compared to the vanilla OpenNetVM framework. For flexibility, FlexNFV can rapidly react to changing traffic arrival rates by performing dynamic NF scaling. FlexNFV achieves 17.5% higher performance than NFVnice [9], the state-of-the-art NFV platform, with the same settings.

The remainder of this article is organized as follows: Sec. II presents related works of dynamic NF scaling, and improvement of NFV performance. In Sec. III, we introduce the system architecture, design and implementation of FlexNFV. We present system optimizations in Sec. IV. Sec. V analyzes the performance of FlexNFV and compares it with other frameworks. Finally, we conclude this article in Sec. VI.

II. Related Work

Dynamic NF scaling is key to the success of NFV, which promises to flexibly deploy NF instances whenever and wherever needed. Recent work has explored many different aspects of NF scaling. Some [5], [6], [7] focus on the state migration problem during scaling events, since the state holds the necessary information to correctly process a packet. For example, StatelessNF [6] overcomes the challenge of tight coupling between NF processing and state migration with a centralized data store, and it enables a more flexible NF architecture; S6 [7] improves performance during scaling events by introducing a distributed shared state model and transparently migrates state for dynamic scaling. However, they all target at standalone NFs and cannot be directly applied to service chains. Others [8] devote to running many NF instances on a small number of cores, and NFs of the same type can be replicated or stopped if necessary. Yet, they cannot perform NF scaling in an automatic manner in order to deal with various traffic conditions.

In addition, two NFV platforms, E2 [10] and Metron [11] also consider dynamic scaling while improving service chain performance. To avoid the complex state migration problem, E2 proposes a migration avoidance scheme for maintaining flow affinity. As a result, the NF state can be easily replicated across multiple instances. In our case, we focus on layer 2/3 packet processing, where we adopt consistent hashing for evenly redirecting packets to available instances, while ensuring the flow affinity. Similar to E2, FlexNFV does not require state migration and instead uses a flow table or predefined NF actions for traffic steering. However, E2 does not present a complete NF scaling scheme since it ignores the case for contraction when NFs are underloaded. Supporting stateful dynamic scaling for service chains would leave to be our future work.

By encoding a service chain as multiple individual traffic classes, Metron adopts SNF [12] to synthesize the service chain as an equivalent NF by removing the redundant packet and I/O operations for each traffic class. In this way, Metron enables better flexibility when it requires dynamic scaling. Instead of splitting the entire flows across multiple NF instances as E2 does, Metron just splits the traffic classes that belong to the equivalent NF across the corresponding instances. Specifically, Metron aggregates traffic classes with the same write operations together as a group. For simplicity, it splits the group into two subgroups for scaling out and merges the two subgroups into one for scaling in, respectively, in order to minimize the state transfer across the CPU cores. To provide state management, Metron duplicates the stateful tables of the group, or merge the stateful tables of two subgroups across the involved cores, for maintaining state consistency of related traffic classes. However, Metron inherits the limitations of SNF such as the assumption of synthesizing an equivalent NF with a single read and single write operation, while its flexibility of dynamic scaling can be restricted.

Other research such as NFVnice [9] also utilizes the chain-level information to improve the service chain performance,

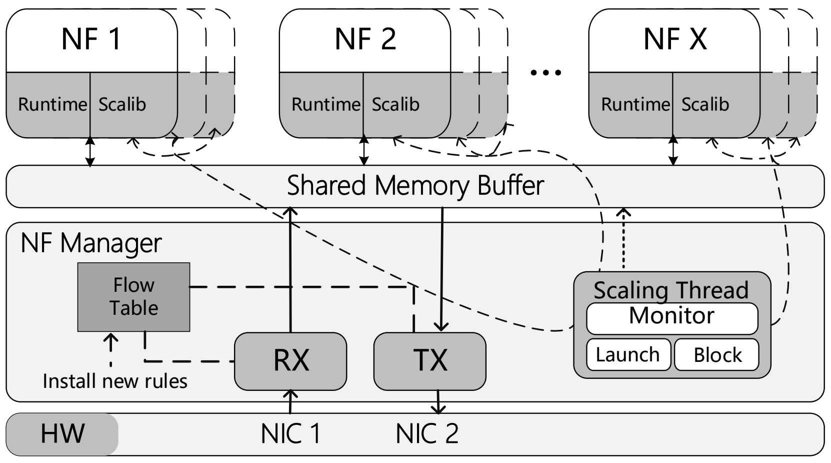

Fig. 1: An illustration of FlexNFV Architecture.

by automatically tuning the CPU share allocated to each NF. This will provide a fair allocation for packet flows and prevent wasted work across the chain, but at the cost of throttling the transmitted traffic since it has to shed load early in the chain. Orthogonal to NFVnice, our work seeks to provide dynamic NF scaling for service chains from a horizontal scope, while following the original traffic rate. We believe that achieving this goal with dynamic scaling is relatively simple compared to scheduling the CPU share by leveraging kernel-space tools such as cgroups. Although FlexNFV may involve more cores, we believe this is feasible for handling traffic peaks while maintaining high throughput in a multi-core platform.

III. FlexNFV Design and Implementation

To support high-performance service chains in an NFV platform, NFs therein must have sufficient processing capacities to handle the incoming packets. As the packet arrival rate changes, the processing requirements of NFs need to be rapidly configured, so as to prevent packet drops and NF halt at any time. This means that the regulatory mechanism inside the NFV platform must be able to perform NF scaling in an automatic manner. FlexNFV provides such capabilities by capturing the required information from the underlying system and NFs at the runtime.

A. System Architecture

Figure 1 illustrates the key building blocks of the FlexNFV platform. As a whole, we leverage DPDK for high-speed packet delivery. Our platform consists of two major components, (1) NFs running on the top layer, and (2) the NF manager running at the bottom. The NF manager serves as the primary DPDK process and NFs run as secondary processes. Both the NF manager and NFs run on a number of pre-set cores. By leveraging DPDK, these two kinds of processes can work together using efficient message queues allocated from a shared memory for inter-process communication. Furthermore, all packet data can also be buffered in the shared memory, making them accessible to all NFs without the need for replication. This results in zero-copy packet delivery through NFs in a service chain, while FlexNFV just copies a packet descriptor between the NF queues (i.e., ring buffers).

More specifically, the NF manager is in charge of transmitting packets between the NIC queues and NF queues, and delivering NF scaling messages when necessary. The RX thread constantly reads packets from a NIC, and puts them into a buffer queue that belongs to a target NF after a flow table lookup. If the lookup fails, it uses a default service chain or installs new flow entries in the flow table. Note that the service chains can be configured in advance. The NF then dequeues packets from its receive ring, and pulls them into its transmit ring after processing. The TX thread is responsible for moving packets to subsequent NFs along the chain and finally transmits them out through another NIC.

The Scaling thread in the manager determines whether an NF should perform dynamic scaling. There are three components in this thread: `Monitor` periodically checks the status of each NF (e.g., the load and NF queues); `Launch` sends the scaling out messages (i.e., launch new NF instances); and `Block` sends the blocking messages (i.e., suspend NFs and turn into a power-efficient mode). We believe it is reasonable to aggregate these components on a single core, since all of them are triggered during a certain time interval (set to 1 second in FlexNFV). The strategies for NF scaling are presented in subsequent sections.

The NF threads run as secondary DPDK processes on one or multiple cores. When launching a new NF instance, the NF goes through a series of NF-specific phases including NF starting, ready and stopping. OpenNetVM provides these functions in its *libnf* library, but can only create one instance each time. FlexNFV therefore provides an abstracted library called *scalib* for implementing NF scaling. For seamless integration to existing NFV frameworks, our *scalib* library only requires minor modifications to NF implementations that *libnf* provides. That is, in FlexNFV, all NFs can be linked with *libnf* library. When creating a new instance, *scalib* can then leverage *libnf* library to get it linked, in order to share all related functions. This makes either parent NF instance or child NF instances (i.e., spawned from parent NF instance) be deployed in a more unified manner.

B. Dynamic Scaling Strategies

To leverage chain-level information, FlexNFV first needs to get the information pertinent to each NF, such as load and queue status. Similarly as the way in [9], we compute the NF load using the product of its packet arrival rate and per-packet processing time.

```
// Check the scaling-specific message from the NF manager
void scalib_check_msg(struct flexnfv_nf *nf);

// Launch new NF instances on a number of designated cores,
//    num_nfs indicates the required number of instances
int scalib_lanuch_nfs(struct flexnfv_nf_info *info,
uint16_t num_nfs);

// Block an NF from running status
void scalib_block_nf(struct flexnfv_nf_info *info);

// Rerun an NF from blocking status
void scalib_rerun_nf(struct flexnfv_nf_info *info);
```

Listing 1: *scalib* APIs exposed to NF implementations.

Once an NF starts up, FlexNFV estimates its per-packet processing time by using timestamps. The NF manager uses a status thread to record statistics for each NF once a second, thus the packet arrival rate to an NF can be easily obtained. With these results, the Scaling thread periodically computes the load for each NF and compares it with a high watermark (H^{wm}) and a low watermark (L^{wm}), respectively. Meanwhile, FlexNFV checks the length of the ring queues (Q) associated with each NF and compares it with a threshold (e.g., the predefined ring queue size). Specifically, for each NF i with load $Load_i$, we have:

- when $Load_i$ is above H_i^{wm}, an overload flag related to NF i is set in its structure. The required number of instances $num_i = \lfloor \frac{Load_i}{H_i^{wm}} \rfloor$; meanwhile, if Q_i is larger than the threshold and the corresponding overload flag is not set, an extra instance is added to num_i;
- when $Load_i$ is below H_i^{wm} and above L_i^{wm}, NF i will be running normally;
- when $Load_i$ is below L_i^{wm}, NF i is treated as underload and a block flag is set in the structure of NF i; or if Q_i is smaller than the threshold and the corresponding block flag is not set, NF i will also be blocked.

Note that the comparison between Q_i and the threshold takes effect only when the same comparison result maintains in a certain check interval (set to 5 in FlexNFV), for each NF i. According to the flag set in the shared NF structure, the Scaling thread sends related messages to NFs through a shared message queue. On the NF side, each NF employs *scalib* library to receive its message from the NF manager. Then, the *scalib* library launches or blocks NF instances dictated by the specific messages.

As shown in Listing 1, the *scalib* library exposes a simple set of Application Program Interfaces (APIs) for various NF implementations. The `scalib_check_msg()` API will be constantly called to ensure that the required NF messages can be immediately received. The `scalib_launch_nfs()` API will only be called with a positive `num_nfs`, since `num_nfs` $\leq$ 0 means no NFs needs to be launched. It returns a false value unless all launching requirements are executed. The last two APIs `scalib_block_nf()` and `scalib_rerun_nf()`, as their names suggest, execute NF blocking and wake up an NF from blocking status to running status, respectively.

Algorithm 1 The mapping from NF to instance

```
Input: NF type: $i$, the number of instances of NF $i$: $j$, global
    mapping matrix: $\mathbf{M}$, maximum instance ID: $ID$
Output: Target instance ID
 1: function MAP_NF_INST($i, j$, $\mathbf{M}$, $ID$)
 2:     $index \leftarrow hash.rss \ \% \ j$;
 3:     $id \leftarrow \mathbf{M}[i][index]$;
 4:     if the block flag of instance $id$ is not set then
 5:         return $id$;
 6:     end if
 7:     while there exist unblocked instances do
 8:         $id++$;
 9:         if $id \geq ID$ then
10:             $id \leftarrow id \ \% \ ID$;
11:         end if
12:         if the block flag of instance $id$ is not set then
13:             return $id$;
14:         end if
15:     end while
16: end function
```

After launching a new NF instance whose block flag is not set, the NF manager will register it to a global mapping two-dimensional matrix, which records the instance ID for each NF type. We rewrite a function `map_nf_inst()` about the mapping from an NF type to a specific instance. As shown in Algorithm 1, we first extract the packet's symmetric RSS hash modulo the number of existing instances of that type to pick out an instance. If the block flag of the instance is not set, it is exactly the target instance; else, we find the next instance unless its block flag is not set. In case that the next instance ID exceeds the maximum ID allowed, we perform a modulo operation to get a valid ID. This ensures flow affinity while automatically load balancing packets across running instances.

C. Launching Avoidance

For an efficient NFV platform, it is vital to avoid launching NFs frequently since existing NFs may be blocked. That is, the processing capabilities of NFs cannot be fully utilized when traffic rate declines, which leaves a great deal of unused NF instances. To mitigate such inefficiency, we design and adopt a simple launching avoidance strategy, which maximally reuses the blocked NFs to prevent launching NFs.

We define a global array for all NF types, which records the number that excludes blocked instances from all instances. After calculating the number of instances that requires to be launched (*launch_req*) when an NF gets overloaded, we compare it with the current number of blocked instances (*block_num*). If *launch_req* is not larger than *block_num*, the manager wakes up the blocked instances by sending rerunning messages in sequence to the NF side until *launch_req* instances are woken up. In this case, the `launch_num` field in the NF's structure is set to be 0 and launching new instances is no longer required. Otherwise, the manager first wakes up *block_num* instances, then sets `launch_num` = $launch\_req - block\_num$, and sends corresponding scaling out messages to the NF side. The evaluation later verifies this strategy.

All the operations are performed at per-determined intervals, making it possible to maintain an appropriate number of child NFs. The time interval is configurable by `C-Macros`.

IV. System Optimizations

In the process of FlexNFV design and implementation, we need to consider: 1) the unnecessary system power consumption under NF blocking status, and 2) the conflict of the multi-instance multi-NF messages in the shared message queues during the process communication. To this end, we introduce the following two system optimizations.

A. Power-aware Management

In FlexNFV, each NF thread has a forwarding loop (DPDK poll mode) to continuously handle arrival packets, unless the loop is terminated with a semaphore. However, when an NF gets blocked and needs to be awakened again in the future, the occupied core wastes a lot of power as the NF no longer requires handling packets.

To relieve this cost, we adopt a P-states (processor performance states) power management approach provided by DPDK. Specifically, when detecting a blocked NF, we decrease the frequency of the CPU core occupied by this NF to the minimum, via a power management library. The NF first needs to perform power initialization when it starts up on a specific core. Once the NF is detected as blocking status, the frequency of its core is set to the minimum by invoking `rte_power_freq_min()` API, for manipulating a virtual file device to allow the NF to set the CPUFreq governor on its core [13]. Once the NF is woken up from blocking status, the frequency is immediately set to normal with `rte_power_set_freq()` API. If the NF is shut down, a power exit stage will be required. This allows an NF to run in two forms according to its status, and provides power savings for the NFV platform.

Limitations: Obviously, a blocked NF still occupies a CPU core. In the early stage of developing FlexNFV, we sought an approach to completely release an occupied core. Apart from P-states power management, DPDK also provides C-states (sleep states) power management, which puts a core into an idle state and can be awakened via an interrupt. However, we found that the overhead to wake up an NF from the idle state inevitably harms the performance (e.g., incur exit latency). Furthermore, a blocked NF still needs to constantly check the incoming messages from the NF manager in its loop, which cannot be achieved in C-states power management without NF polling. Therefore, we adopt P-states power management to reach a compromise. The other limitation is that we could not quantify the power savings of P-States power management in the evaluation, instead we only provide a qualitative analysis above. We leave it to be our future work and we believe even P-states power management would make a great difference in large-scale deployment of NFV.

B. Elimination of Queue Conflicts

Replicating multiple instances from a single console incurs a new problem, in which all instances competing for a single message channel causes that scaling messages cannot be correctly sent to the desired NF instances. For instance, a blocking message sent to an NF instance may not be received on the NF side. During the system debugging phase, we find that after the NF manager enqueues the scaling messages of multiple NF instances, the sequence of these messages will be disordered when they are dequeued in the shared message queue, and the unique identification, instance ID, cannot be recognized. We call it *queue conflicts* since scaling messages from all NF instances are put and mixed in a shared channel. As a result, on the NF side, the *scalib* library cannot make the right scaling decisions for the right NFs.

To resolve the queue conflicts in a message channel, we add a dedicated message queue in the structure related to each NF. Meanwhile, we use separate ring queues allocated from the shared memory pool for sending messages in NF manager. On the NF side, instead of dequeuing a number of messages from all instances at the same time, `scalib_check_msg()` dequeues only one message from the exclusive message queue related to the instance each time. This ensures that a message sent from the NF manager can be accurately received by the appropriate instance and all messages from the manager will no longer be mixed up across the instances.

V. Evaluation

A. Testbed Setup and Approach

Our experimental testbed contains two servers: a Dell R730 server with dual Intel Xeon E5-2620 v4 @ 2.10GHz CPUs (32 cores), a Dual port 82599ES 10Gb NIC and 64GB memory; and a Dell R740 server with dual Intel Xeon Silver 4116 @ 2.10GHz CPUs (48 cores), a Dual-port Intel 710 10Gb NIC and 128GB memory. Both servers run CentOS 7.5 with kernel 3.10.0 and use Intel DPDK v18.05. We use Dell R730 to run Pktgen-DPDK v3.5.9 to generate 64B packets [14], and use Dell R740 as the system under test for running FlexNFV. The two Dell servers are connected back-to-back with the 10GB NICs that are compatible with DPDK.

For all the packet throughput in figures and drop rate in the table, we adopt the average values collected from repeated experiments. We evaluate the performance of FlexNFV by comparing it with OpenNetVM and NFVnice, and show its flexibility by suddenly changing the traffic arrival rate. It's worth noting that we do not compare FlexNFV against both E2 and Metron, since FlexNFV adopts a different forwarding model from that of E2 and Metron to support NFV service chains. Specifically, FlexNFV adopts pipeline model that uses multiple cores to carry a chain, while E2 leverages SoftNIC [15] and Metron relies on SNF, both of which belong to run-to-completion (RTC) model as they consolidate an entire chain on a CPU core.

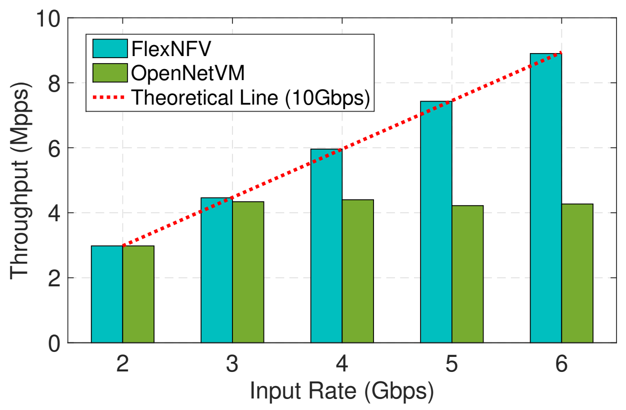

Fig. 2: Throughput in a service chain of 3 NFs for FlexNFV and OpenNetVM with varying input rate (64-byte packets).

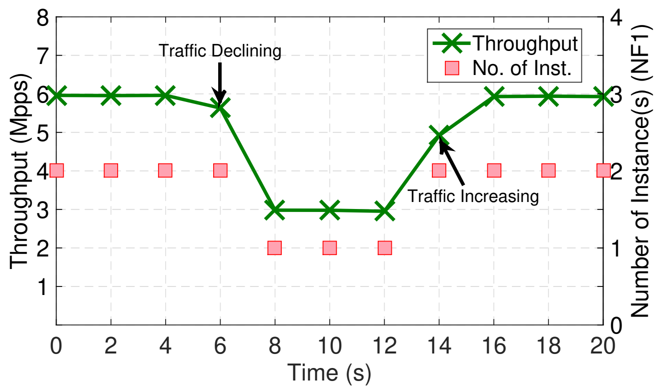

Fig. 3: Throughput in the service chain and number of instance(s) of NF1 when the arrival rate suddenly changes.

B. *FlexNFV Performance*

We first demonstrate the high performance of FlexNFV when compared to OpenNetVM. We consider a service chain consisting of three types of NFs, with low (NF1, 408), medium (NF2, 660) and high (NF3, 802) maximum load [2]. Specifically, NF1 is a basic monitor which prints information regarding incoming packets and then sends them to the next NF. NF2 is a forwarder that forwards packets to a specific NF destination. NF3 is an encryptor that uses the AES (Advanced Encryption Standard) cypher to encrypt packets, and then forwards them out through the NIC. The maximum load is computed by the product of the maximum arrival rate that the NF supports (just no packet loss) and the average per-packet processing time. The maximum arrival rate of an NF can be observed and measured in the experiments, and the average per-packet processing time is estimated by using timestamps when the NF starts up. Each NF is pinned to a dedicated core.

As shown in Fig. 2, the throughput of FlexNFV improves by a factor of 2 when the input rate reaches 6 Gbps. OpenNetVM achieves no improvement in throughput when the input rate exceeds 3 Gbps, while the throughput of FlexNFV keeps growing in line with the theoretical line. Note that the theoretical line here means the maximum achievable throughput corresponding to the input rate for 64-byte packets. FlexNFV benefits from this for the capability of dynamic NF scaling when it detects that the load imposed by the increasing packet arrival rate is larger than the maximum load of NFs. However, OpenNetVM cannot perform dynamic scaling even when NF1, for example, is overloaded, thereby resulting in considerable packet drops and no throughput improvement. To maximally utilize FlexNFV's scaling ability, we allocate a dedicated core once it requires a new instance.

Table I further verifies this. It shows that FlexNFV has no packet drops on NF1 under all input rates, while the drop rate of OpenNetVM dramatically grows with the increase of input rate (more than half in 6 Gbps). Meanwhile in FlexNFV, we can also see that the required number of instances for each type of NF increases with the input rate.

[2]The watermarks are obtained based on maximum load. We regard the NF with the lowest maximum load as the bottleneck NF, i.e., NF1.

TABLE I: The packet drop rate of NF1 in the service chain for FlexNFV and OpenNetVM, and the number of instance(s) of NFs in FlexNFV with varying input rate (NF scaling).

		Drop Rate (Mpps)				
Input Rate (Gbps)		**2**	**3**	**4**	**5**	**6**
FlexNFV		0	0	0	0	0
OpenNetVM		0	0.03	1.59	3.21	4.63
		Number of Instance(s)				
FlexNFV	NF1	1	2	2	3	4
	NF2	1	2	2	2	3
	NF3	1	1	2	2	2

C. *Load-aware Flexibility of FlexNFV*

We next show the flexibility of FlexNFV, by changing the traffic rate during the runtime. We consider the same service chain used above. The experiment is to demonstrate that the NF manager is capable of sending blocking messages to the NF side, when it monitors that the traffic declines to a certain extent (i.e., underload status). Then the NF manager sends rerunning messages to the blocked NFs as the traffic recovers, thus maintaining an appropriate number of child NFs.

As shown in Fig. 3, FlexNFV starts with an initial input rate of 4 Gbps and runs the service chain with doubled instances for all NF types. In time 6, the input rate drops to 2 Gbps. The Scaling thread immediately detects this change and accurately computes that only one instance (such as type NF1) is sufficient to hold the total traffic. Meanwhile, the Scaling thread sets the block flag of one instance and sends a corresponding blocking message to the NF side. Then the `map_nf_inst()` function is called to redirects all incoming packets to the other instance. After that in time 14, the Scaling thread again detects that the input rate reramps to 4 Gbps. By this time, the running instance gets overloaded and requires launching an instance. However, since there already exists one blocked instance, it wakes up and reruns this blocked instance for processing incoming packets again (i.e., launching avoidance). The entire process only incurs a small number of packet drops and the drop rate rapidly turns to zero.

In case of launching a new instance (rather than reusing the blocked instance), we also measure the scaling latency by using timestamps. The whole process includes receiving

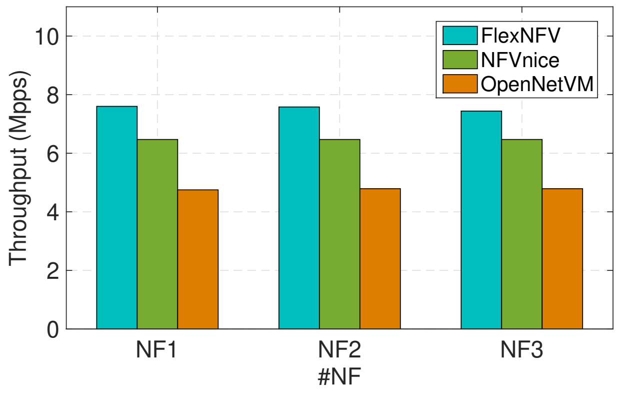

Fig. 4: Throughput of NFs in the service chain for FlexNFV, NFVnice and OpenNetVM.

a scaling message from the NF manager, initializing the NF information, waiting for the NF manager to assign a valid ID, and starting the NF. From repeated experiments on our testbed, the scaling latency for NFs is about 998 ms, which is sufficient for handling the traffic rate changes.

The results indicate that FlexNFV can flexibly react to varying loads in a rapid and efficient manner.

D. Comparison with NFVnice

We now compare the performance of FlexNFV with NFVnice in throughput. Both works consider the chain-level information but improve service chain performance by different methods (as discussed earlier). Since NFVnice is not open-sourced, we are unable to reconduct their experiments using our platform. To fairly compare with NFVnice using the same settings, we retrieve the desired results (Chain 1 reported in Section 4.2.2) in their article, since both running NFs in a service chain on separate and dedicated cores. Meanwhile, we adjust the order of NFs in the service chain (NF3 → NF2 → NF1) to match the setting in their work, and allocate at most one extra core to each NF for scaling. We generate packets at line rate this time.

Figure 4 exhibits the experimental results, including the throughput of OpenNetVM for better comparison. From the figure, we can see that both the throughput of FlexNFV and NFVnice are higher than OpenNetVM. In particular, FlexNFV achieves ~1.13 Mpps over NFVnice, yielding an improvement of 17.5% in throughput. This confirms that FlexNFV can achieve more performance benefits with dynamic scaling than NFVnice with backpressure and CPU scheduling approaches.

VI. Conclusion

Dynamic NF scaling holds the key to the flexibility of NFV. Existing NFV frameworks either focus on standalone NFs or cannot automatically perform NF scaling. To collectively consider the set of factors that affect the service chain performance, the FlexNFV architecture accurately updates the chain-level information such as the current status of each NF, and performs dynamic NF scaling under various traffic conditions. We abstract the implementation logic for dynamic NF scaling as a library, which provides a set of APIs for developing NFs with load-aware flexibility. We also introduce optimizations such as power-aware processing while achieving high performance. We believe that our proposed FlexNFV framework is in the right direction of the success of NFV.

References

[1] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: Network processing as a cloud service," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 13–24, Aug. 2012. [Online]. Available: http://doi.acm.org/10.1145/2377677.2377680

[2] R. Mijumbi, J. Serrat, J. 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, Firstquarter 2016.

[3] DPDK, "Data plane development kit," https://dpdk.org, 2018.

[4] L. Rizzo, "netmap: A novel framework for fast packet i/o," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, 2012, pp. 101–112.

[5] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "Opennf: Enabling innovation in network function control," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, pp. 163–174. [Online]. Available: http://doi.acm.org/10.1145/2619239.2626313

[6] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless network functions: Breaking the tight coupling of state and processing," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 97–112. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/kablan

[7] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker, "Elastic scaling of stateful network functions," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, 2018, pp. 299–312. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/woo

[8] 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*, ser. HotMIddlebox '16. New York, NY, USA: ACM, 2016, pp. 26–31.

[9] S. G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. K. Ramakrishnan, T. Wood, M. Arumaithurai, and X. Fu, "Nfvnice: Dynamic backpressure and scheduling for nfv service chains," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: ACM, 2017, pp. 71–84. [Online]. Available: http://doi.acm.org/10.1145/3098822.3098828

[10] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: A framework for nfv applications," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. New York, NY, USA: ACM, 2015, pp. 121–136. [Online]. Available: http://doi.acm.org/10.1145/2815400.2815423

[11] G. P. Katsikas, T. Barbette, D. Kostić, R. Steinert, and G. Q. Maguire, Jr., "Metron: Nfv service chains at the true speed of the underlying hardware," in *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'18. Berkeley, CA, USA: USENIX Association, 2018, pp. 171–186. [Online]. Available: http://dl.acm.org/citation.cfm?id=3307441.3307457

[12] G. P. Katsikas, M. Enguehard, M. Kuźniar, G. Q. Maguire Jr, and D. Kostić, "Snf: synthesizing high performance nfv service chains," *PeerJ Computer Science*, vol. 2, p. e98, Nov. 2016.

[13] "Sample application user guides - dpdk," https://fast.dpdk.org/doc/pdf-guides-18.05/sample_app_ug-18.05.pdf, 2018.

[14] K. Wiles, "Pktgen-dpdk - traffic generator powered by dpdk," https://git.dpdk.org/apps/pktgen-dpdk/, 2018.

[15] 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, May 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html

Xincai Fei received the B.Eng. degree from Huazhong University of Science and Technology, Wuhan, China. He is currently a Ph.D. student in the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China. His current research interests focus on Network Function Virtualization, resource allocation and system optimization.

Fangming Liu received the B.Eng. degree from the Tsinghua University, Beijing, and the Ph.D. degree from the Hong Kong University of Science and Technology, Hong Kong. He is currently a Full Professor with the Huazhong University of Science and Technology, Wuhan, China. His research interests include cloud computing and edge computing, datacenter and green computing, SDN/NFV/5G and applied ML/AI. He received the National Natural Science Fund (NSFC) for Excellent Young Scholars, and the National Program Special Support for Top-Notch Young Professionals. He is a recipient of the Best Paper Award of IEEE/ACM IWQoS 2019, ACM e-Energy 2018 and IEEE GLOBECOM 2011, as well as the First Class Prize of Natural Science of Ministry of Education in China.

Hai Jin is a Cheung Kung Scholars Chair Professor of computer science and engineering at Huazhong University of Science and Technology (HUST) in China. Jin received his PhD in computer engineering from HUST in 1994. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. Jin worked at The University of Hong Kong between 1998 and 2000, and as a visiting scholar at the University of Southern California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. Jin is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientists of National 973 Basic Research Program Project of Virtualization Technology of Computing System, and Cloud Security. Jin is an IEEE Fellow and a member of the ACM. He has co-authored 15 books and published over 600 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security.

Bo Li is a professor in the Department of Computer Science and Engineering, Hong Kong University of Science and Technology. He holds the Cheung Kong chair professor in Shanghai Jiao Tong University. Prior to that, he was with IBM Networking System Division, Research Triangle Park, North Carolina. He was an adjunct researcher with Microsoft Research Asia-MSRA and was a visiting scientist in Microsoft Advanced Technology Center (ATC). He has been a technical advisor for China Cache Corp. (NASDAQ CCIH) since 2007. He is an adjunct professor with the Huazhong University of Science and Technology, Wuhan, China. His recent research interests include: large-scale content distribution in the Internet, Peer-to-Peer media streaming, the Internet topology, cloud computing, green computing and communications. He is a fellow of the IEEE for “contribution to content distributions via the Internet". He received the Young Investigator Award from the National Natural Science Foundation of China (NSFC) in 2004. He served as a Distinguished lecturer of the IEEE Communications Society (2006-2007). He was a corecipient for three Best Paper Awards from IEEE, and the Best System Track Paper in ACM Multimedia (2009).