NFP: Enabling Network Function Parallelism in NFV

Chen Sun
Tsinghua University
c-sun14@mails.tsinghua.edu.cn

Jun Bi*
Tsinghua University
junbi@tsinghua.edu.cn

Zhilong Zheng
Tsinghua University
zhengzl15@mails.tsinghua.edu.cn

Heng Yu
Tsinghua University
hengyu1213@163.com

Hongxin Hu
Clemson University
hongxih@clemson.edu

ABSTRACT

Software-based sequential service chains in Network Function Virtualization (NFV) could introduce significant performance overhead. Current acceleration efforts for NFV mainly target on optimizing each component of the sequential service chain. However, based on the statistics from real world enterprise networks, we observe that 53.8% network function (NF) pairs can work in parallel. In particular, 41.5% NF pairs can be parallelized without causing extra resource overhead. In this paper, we present NFP, a high performance framework, that innovatively enables *network function parallelism* to improve NFV performance. NFP consists of three logical components. First, NFP provides a policy specification scheme for operators to intuitively describe sequential or parallel NF chaining intents. Second, NFP orchestrator intelligently identifies NF dependency and automatically compiles the policies into high performance service graphs. Third, NFP infrastructure performs light-weight packet copying, distributed parallel packet delivery, and load-balanced merging of packet copies to support NF parallelism. We implement an NFP prototype based on DPDK in Linux containers. Our evaluation results show that NFP achieves significant latency reduction for real world service chains.

CCS CONCEPTS

• **Networks → Middle boxes / network appliances**; *Network performance analysis*; Network control algorithms;

KEYWORDS

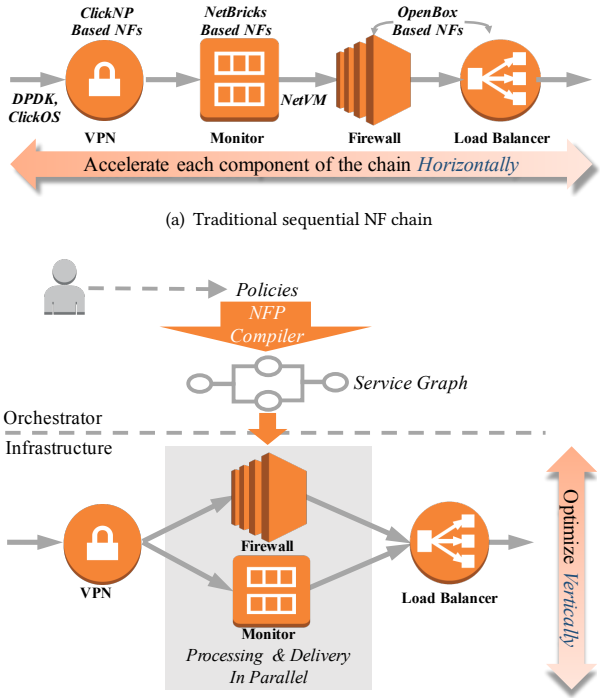
NFV, network function parallelism, service chain

ACM Reference format:
Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. 2017. NFP: Enabling Network Function Parallelism in NFV. In *Proceedings of SIGCOMM '17, Los Angeles, CA, USA, August 21–25, 2017,* 14 pages.
https://doi.org/10.1145/3098822.3098826

*Chen Sun, Jun Bi, Zhilong Zheng, and Heng Yu are with Institute for Network Sciences and Cyberspace, Tsinghua University, Department of Computer Science, Tsinghua University, and Tsinghua National Laboratory for Information Science and Technology (TNList). Jun Bi is the corresponding author.

SIGCOMM '17, August 21–25, 2017, Los Angeles, CA, USA

ACM ISBN 978-1-4503-4653-5/17/08…$15.00
https://doi.org/10.1145/3098822.3098826



(a) Traditional sequential NF chain

(b) NFP framework supporting parallel NFs

Figure 1: Traditional sequential chain derived from [36] v.s. NFP service graph with parallel NFs

1 INTRODUCTION

Network Functions Virtualization (NFV) addresses the problems of traditional proprietary middleboxes [61] by leveraging virtualization technologies to implement network functions (NFs) on commodity hardware, in order to enable rapid creation, destruction, or migration of NFs [24]. In operator networks [52], data centers [32, 36], mobile networks [25] and enterprise networks [60], network operators often require traffic to pass through multiple NFs in a particular *sequence* (e.g. firewall+IDS+proxy) [7, 26, 50], which is commonly referred to as service chaining. Meanwhile, Software-defined Networking (SDN) is used to steer traffic through appropriate NFs to enforce chaining policies [2, 16, 23, 32, 50]. Together, NFV and SDN can enable flexible and dynamic *sequential* service chaining.

However, the benefits of NFV come with considerable compromises [24]. Especially, software-based NFs could introduce significant performance overhead (e.g., Ananta Software Muxes running on commodity servers can add from 200μs to 1ms latency at 100 Kpps [21]). Moreover, the service chain latency may grow linearly with the length of the chain (possibly *seven* or *longer* [36]), which may be unacceptable for some applications that work under tight latency constraints. For example, real-time analytics and Online Data-Intensive (OLDI) applications like web search and online retail generate short messages that are sensitive to packet delay, which partially comes from NFs inside data centers [31]. Applications, such as algorithmic stock trading and high performance distributed memory caches, require ultra-low (a few microseconds) latency from NFs in the cloud infrastructure [21, 37].

Some research efforts have been devoted to addressing the performance drawback of software-based NFV. We mark their optimization points on the example service chain in Figure 1(a). #1: *Individual NF acceleration:* ClickNP [37] proposes to offload software logic onto programmable hardware (e.g. FPGA) to accelerate *individual NFs*. NetBricks [47] abandons VMs or containers, and runs NFs on a single CPU core to improve NF performance. #2: *Packet delivery acceleration:* Intel DPDK [30], ClickOS [38] and NetVM [28, 64] optimize packet delivery from the network interface cards (NICs) to VMs, and between VMs. #3: *NF modularization:* OpenBox [7] modularizes NFs (which is also proposed in [2, 46, 60]) and improves overall performance by *sharing* common building blocks between NFs and chaining the remaining blocks together. We summarize that #1, #2 and #3 address the NFV performance challenge in a *horizontal* scope, i.e. accelerating each component in a service chain horizontally, while still following *sequential* composition of NFs.

However, a closer look into the NFs in a service chain shows that some NFs share no dependency and could work *in parallel*. For example, in the service chain shown in Figure 1(a), the Monitor NF only maintains packet statistics without modifying packets. Therefore, as shown in the *service graph* in Figure 1(b), we could send traffic into the Monitor and the Firewall simultaneously, pick the output of the Firewall, and achieve the same result as sequential composition. In this way, the equivalent chain length is *three* and could bring a theoretical latency reduction by 25%. Moreover, our study on NFs deployed in enterprise networks (§4) reveals that 53.8% NF pairs could work in parallel. Especially, 41.5% pairs can be parallelized without introducing extra resource overhead.

Therefore, orthogonal to above NFV acceleration efforts, we exploit opportunities to enhance NFV performance from a *vertical* scope. Based on above observations, we refer to the idea of Instruction-Level Parallelism (ILP), an acceleration technology widely adopted in modern CPUs [13], and propose NFP, a high performance framework, that innovatively embraces *NF parallelism* to reduce NFV latency. As shown in Figure 1(b), NFP framework consists of three logical components including a policy specification scheme, NFP orchestrator, and NFP infrastructure. Our main contributions are:

- We present the motivation and design challenges of introducing NF parallelism into NFV, and propose the NFP framework that exploits *NF parallelism* to improve NFV performance. (§2)
- NFP provides a *policy specification scheme* for intuitively representing sequential or parallel chaining intents of network operators to improve the parallelism optimization effect of NFP. (§3)
- We design NFP *orchestrator* that can identify NF dependency and automatically compile policies into high performance service graphs with parallel NFs. (§4)
- We design NFP *infrastructure* that efficiently supports NF parallelism based on light-weight packet copying, distributed parallel packet delivery, and load-balanced merging of packet copies. (§5)
- We implement NFP based on DPDK in Linux containers. Evaluations show that NFP could achieve up to 35.9% latency reduction for real world service chains. (§6)

2 MOTIVATION AND CHALLENGES

This section first describes the background and motivation for adopting NF parallelism in NFV. We then introduce design challenges of NF parallelism in NFV.

2.1 Background and Motivation

Background: Parallelism has been well studied in computer programming [1, 29] and high performance computing [14]. A type of parallelism named Instruction-Level Parallelism (ILP) reorganizes sequential instructions and executes independent instructions in parallel [1]. ILP has been widely adopted in modern processors [13]. Analogous to ILP, we intend to embrace NF parallelism to improve NFV performance by identifying independent NFs and making them work in parallel.

To determine the *optimization scope* of NF parallelism for NFV, we investigate and conclude from the literature that there exist two models to support service chains in NFV, including the *pipelining* model [28, 38, 64] and the *run-to-completion (RTC)* model [27, 47]. The *pipelining* model uses multiple cores to carry a chain, while the *RTC* model consolidates an entire service chain as a native process on a CPU core. NF parallelism targets at accelerating *pipelining model* based NFV networks. We present a more detailed discussion about two models in § 7.

NF Parallelism brings significant latency benefit: To get intuitive feelings about the optimization effect of NF parallelism, we collect commonly deployed NFs, their actions, and percentages in enterprise networks [60, 61] (see Table 2 in §4). Based on the statistics, we find that 53.8% of NF pairs can be parallelized (§4), which promises the optimization range of NF parallelism. Furthermore, according to our measurement, parallelizing the Firewall and Monitor NFs in Figure 1 brings 12.9% latency reduction. For some real world service chains, NFP can achieve up to 35.9% latency reduction (§6).

NF Parallelism can *work with* and *benefit* other optimization techniques: First, for individual NF acceleration techniques [37, 47], NF parallelism can parallelize independent accelerated NFs to achieve higher performance. Second, we can use fast packet delivery technologies [28, 30, 64] to accelerate packet delivery in NF parallelism. Third, NF parallelism could benefit both monolithic and modularized NFs [2, 7, 60]. After decomposing NFs into building blocks, common modules can be shared, and NF parallelism can be implemented in the granularity of building blocks. We provide an example of combining parallelism and modularity in §7.

2.2 Design Challenges

Based on the above motivation, we propose a novel framework, NFP, to enable NF parallelism in NFV to improve its performance. We encounter four key challenges in the design of NFP.

Policy design to describe service graphs: For sequential service chaining, network operators assign specific positions for NFs in a service chain. However, when we intend to support NF parallelism in NFV, traditional approach for specifying NF positions cannot be used to describe NF parallelism intents. Therefore, supporting NF parallelism requires a new, intuitive way to describe both sequential and parallel NF composition intents to construct optimized service graphs. To this end, NFP proposes a policy specification scheme with richer semantics to address this challenge. We introduce its definition in §3.

Orchestrator design to construct service graphs: With NF parallelism, traditional sequential service chains are optimized into high performance service graphs. Thus, supporting NF parallelism challenges the orchestrator to *identify* NF dependencies and automatically *compile* NFP policies into high performance service graphs. However, with the fast innovation and booming NFs in NFV, exhaustively and manually analyzing each pair of NFs is time consuming and lacks scalability. In response, we propose NF dependency principles along with an automatic dependency identification algorithm running in the NFP orchestrator. We present them in §4.

Orchestrator design to optimize resource overhead: NF parallelism may introduce two or more copies of every packet that could occupy extra network bandwidth resource and largely deteriorate throughput. Thus, the orchestrator is challenged to construct high performance service graphs with marginal resource overhead. One strawman solution is to adopt *consolidation* [28, 60] and place parallel NFs in the same hardware box to store packet copies in the memory. However, this solution still suffers from the trade-off of gaining 2× *performance* at the cost of 2× *memory resources*. In response, we propose an optimized solution to mitigate the resource overhead brought by NF parallelism. We carefully design the NFP orchestrator that intelligently identifies opportunities to realize NF parallelism without packet copying. Furthermore, we introduce several resource optimization techniques. We discuss them in §4.

Infrastructure design to support NF parallelism: Introducing NF parallelism into NFV incurs several concerns on the infrastructure design. First, the infrastructure should support light-weight packet copying to minimize the copy overhead. Second, the infrastructure requires a merging module to merge processed packets from parallelized NFs into the final output. However, the merger is burdened to process massive packet copies and could become a performance bottleneck. Finally, current solutions on packet delivery between NFs [28, 38, 64] depend on a centralized virtual switch. However, packet queuing in this centralized switch would compromise the performance. The problem is even worse when supporting NF parallelism, which might double the number of packets to be forwarded at the same time. We introduce the NFP infrastructure design to address above challenges in §5.

3 POLICY DEFINITION

For traditional sequential service chaining, network operators need to specify a policy, which sequentially assigns positions to NFs in the chain, as shown in the first row of Table 1. However, NFP attempts to construct high performance service graphs with parallel NFs, requiring an intuitive way to describe both sequential and parallel NF composition intents. Therefore, we define a policy specification scheme, which includes three *types of rules*, in NFP. Network operators can compose several rules together into a policy to describe chaining intents.

Table 1: Sequential v.s. NFP description of chaining intents based on the example in Figure 1. (FW stands for Firewall, and LB represents Load balancing)

Traditional description of the service *chain* in Fig 1(a)	Assign(VPN, 1) Assign(Monitor, 2) Assign(FW, 3) Assign(LB, 4)
NFP Policy for the service *chain* in Fig 1(a)	Order(VPN, *before*, Monitor) Order(Monitor, *before*, FW) Order(FW, *before*, LB)
NFP Policy for the service *graph* in Fig 1(b)	Position(VPN, *first*) Order(FW, *before*, LB) Order(Monitor, *before*, LB)

Order (NF1, *before*, NF2): This rule expresses the desired execution order of two NFs. For example, in the service chain shown in Figure 1(a), the network operator can first send the traffic to the VPN and then the Monitor by specifying `Order(VPN, before, Monitor)`. This `Order` rule type can be used to describe a sequential NF composition intent. Multiple `Order` rules can describe a sequential service chain as shown in the second row in Table 1, which is equivalent to the traditional description in the first row. This ensures the *compatibility* of NFP to support sequential service chains. Network operators can simply provide a traditional service chain specification without using NFP policies, and we are able to automatically transfer it to NFP policies. Then, NFP orchestrator could explore parallelism opportunities for the NFs in `Order` rules for better performance. In this way, NFP could optimize traditional sequential service chains into high performance service graphs. We elaborate this in §4.

Priority (NF1 > NF2): In NFP, network operators should be able to describe the intent of executing two NFs in parallel. However, the actions of the two NFs may conflict. For instance, Firewall and Intrusion Prevention System (IPS) may disagree on whether to drop packet or not. Therefore, network operators can specify the `Priority(IPS > Firewall)` rule to parallelize the two NFs while indicating the system should adopt the processing result of IPS during conflicts. NFP orchestrator will automatically identify conflicting actions between NFs and parallelize them accordingly.

A seemingly equal rule to the above rule is `Order (Firewall, before, IPS)`. Although they may provide the same result, they are used in different situations. An `Order` rule is used to *intuitively* describe sequential NF composition intents, while two NFs in a `Priority` rule are intended to be executed in parallel. NFP orchestrator further inspects the dependency of NFs in an `Order` rule to see whether they are parallelizable. If they are, an `Order` rule is converted into a `Priority` rule, and the NF with the back order is assigned a higher priority. If not, the two NFs should still be chained in sequence.

Table 2: A non-exhaustive list of commonly deployed NFs and their actions on packets [8, 60, 61]. The % column presents the percentage of the NF deployed in enterprise networks derived from [60]. (R for Read, W for Write, T for True, and Add/Rm for Add headers to or Remove headers from packets)

NF	Products	%	SIP	DIP	SPORT	DPORT	Payload	Add/Rm	Drop
Firewall	iptables	26%	R	R	R	R			T
NIDS	NIDS cluster [62]	20%	R	R	R	R	R		
Gateway(Conf /Voice/Media)	Cisco MGX [11]	19%	R	R					
Load Balance	F5 [45], A10 [44]	10%	R/W	R/W	R	R			
Caching	Nginx [54]	10%	R		R		R		
VPN	OpenVPN [17]	7%	R	R			R/W	T	
NAT	iptables		R/W	R/W	R/W	R/W			
Proxy	Squid [56]		R/W	R/W					
Compression	Cisco IOS [10]						R/W		
Traffic Shaper	Linux tc [22]								
Monitor	NetFlow [12]		R	R	R	R			

Position (NF, *first*/*last*): The service chain for north-south traffic in data centers [36] in Figure 1(a) requires all packets to be processed by the VPN first. This raises the requirement of placing an NF in a specific position in the service graph. However, we cannot pre-acknowledge the final optimized graph structure. Thus, we can only assign an NF as the *first* or *last* one in the service graph. We design the `Position(NF, first/last)` rule to describe such type of intents. For instance, we can specify a `Position(VPN, first)` to ensure packets traverse the VPN first. NFP orchestrator places the VNF in the service graph in a *sequential* manner, as illustrated by the VPN NF in Figure 1(b).

With above rules, network operators can define chaining intents by composing multiple rules into a policy (the third row in Table 1) to describe a service graph (Figure 1(b)). Note that traditional sequential description of a service chain has to assign *all* NFs with positions in the chain. With NFP policy, however, operators can only specify chaining intents for *partial* NFs as needed. This looses the constraints and leaves a larger space for NFP to seize all parallelism opportunities to improve performance. Finally, the rules manually written by operators could possibly conflict with each other. For example, an operator could write two rules with conflicting orders, i.e. `Order(NF1, before, NF2)` and `Order(NF2, before, NF1)`, or assign an NF at different positions, i.e. `Position(NF1, first)` and `Position(NF1, last)`. The challenges of policy conflict detection and resolution have been recognized and studied in [34, 40, 49]. We will refer to prior wisdom and leave them to our future work.

4 ORCHESTRATOR DESIGN

NFP orchestrator takes the NFP policies as input, *identifies* NF dependencies, and automatically *compiles* policies into high performance service graphs possibly with parallel NFs. The optimization goals of the compilation is to fully benefit from the *high performance* brought by NF parallelism, while introducing *very little resource overhead*. This section will introduce each step in the service graph construction process in detail.

4.1 NF Parallelism Analysis

As introduced above, for NFP policies, we take the two NFs in a `Priority` rule as directly parallelizable, and place the NF assigned in a `Position` rule in the head or tail of the graph in a sequential manner. However, for two NFs in an `Order(NF1, before, NF2)` rule, we need to further explore their parallelism possibility.

Table 3: For `Order(NF1, before, NF2)`, whether the two NFs are parallelizable, and whether we need to copy packets if the two NFs can be executed in parallel.
Green blocks denote *parallelizable, no need to copy.*
Orange blocks denote *parallelizable, need copy pkts.*
Gray blocks denote *not parallelizable situations.*
For read-write or write-write case, we need not copy packets if two NFs modify different fields.

NF1 \ NF2	Read	Write	Add/Rm	Drop
Read				
Write				
Add/Rm				
Drop				

NFs may perform various actions on packets including *Reading* or *Writing* headers or payloads, *Adding* or *Removing* header fields, and *Dropping* packets. Table 2 presents a summary of some commonly deployed NFs and their actions on packets, derived from [8, 60, 61]. We observe that the actions of different NFs may conflict with each other. For instance, the NAT and the Load Balance both modify the destination IP address of a packet. If the operator inputs an `Order(NAT, before, LB)`, the orchestrator is challenged to identify the parallelism possibility of the two NFs for high performance.

To analyze whether two NFs are parallelizable, we propose a *result correctness* principle: *Two NFs can work in parallel, if parallel execution of the two NFs results in the same processed packet and NF internal states as the sequential service composition.* Based on this principle, we summarize parallelizable and unparallelizable situations in Table 3. The green and orange blocks represent *parallelizable* situations. For example, suppose NF1 reads the packet header, and NF2 later modifies the same header field. To ensure that NF1 reads the original header that has not been changed by NF2, we could copy the packets and send two copies into NF1 and NF2 *in parallel*. Gray blocks denote *unparallelizable* situations. For

example, if NF1 first writes a packet header and later NF2 reads this header, the operator intends to transmit the modification of NF1 to NF2. Therefore, the two NFs should work in sequence.

4.2 Resource Overhead Optimization

In a `Priority` rule or a parallelizable `Order` rule, the two NFs can work in parallel with packet copying, which could incur resource overhead. Furthermore, large memory block copying could degrade latency and throughput performance [28, 30]. To address this challenge, NFP explores opportunities to support NF parallelism without packet copying, and proposes optimization techniques to reduce copying overhead.

We refer to the *result correctness* principle and summarize the situations where packet copying is not necessary for parallelism in green blocks in Table 3. For example, suppose NF1 and NF2 both read the packet. Since the reading action does not modify packets, the two NFs can read the same packet simultaneously. Orange blocks represent situations where packet copying is needed for NF parallelism. For example, if NF1 reads a header and NF2 later modifies it, we could copy the original packet, send two copies into NF1 and NF2 *in parallel*, and select the output of NF2 as the final output, which could still achieve *result correctness*.

To further reduce copying overhead, we propose the following resource optimization techniques based on our insights on NF actions and dependencies.

OP#1: Dirty Memory Reusing: As represented by blocks with both green and orange colors in Table 3, for read-write or write-write situations, we decide the necessity for packet copying depending on whether the two actions operate on the same packet field. If two NFs read or write different fields of a packet, they can operate on the same packet copy. We name this optimization technique as Dirty Memory Reusing, which could reduce packet copying necessities.

Note that when two NFs on two CPU cores operate the same packet copy simultaneously, the header fields they manipulate could possibly map to the same cache line, incurring cache contention and degrading performance. However, according to our evaluation in § 6, despite the possible existence of cache contention, by adopting NF parallelism, NFP could still significantly outperform sequential service chaining in NFV. Dirty Memory Reusing is designed to reduce resource overhead when pursuing NF parallelism, and provided as an optimal feature. If a network operator cares little about resource consumption in NFV, this feature could be switched off to provide safe performance enhancement.

OP#2: Header-Only Copying: We observe from Table 2 that only few NFs (7%) modify packet payloads. Besides, we derive from [4] that the average packet size in data centers is around 724 bytes. For TCP packets, the header only occupies 8.8% of the total size. Therefore, we propose Header-Only Copying that only copies packet headers for some cases of NF parallelism. Multiple NFs that modify the payload will be executed in sequence, which is a very rare situation according to Table 2. Header-Only Copying could improve performance and save memory by shortening the length of memory to be copied. Note that after header copying, we should modify the *packet length* field of the copied header to the length of the header itself, ensuring that parallel NFs receive valid packets.

Algorithm 1: NF Parallelism Identification

```
   Input: Order(NF1, before, NF2).
   Output: Parallelizable p, conflicting actions ca.
1  actionList1 = fetchAction(AT, NF1) ;
2  actionList2 = fetchAction(AT, NF2) ;
3  p = TRUE ;
4  ca = NULL ;
5  foreach (a1, a2) ∈ (actionList1, actionList2) do
6      if (a1, a2) = (read, write) or (write, write) then
7          if (a1, a2) operate the same field then
8              ca.append(a1, a2);
9          continue;
10     switch fetchParallelism(DT, (a1, a2)) do
11         case NOT_PARALLELIZABLE
12             p = FALSE ;
13             return;
14         case PARALLELIZABLE_NO_COPY
15             continue;
16         case PARALLELIZABLE_WITH_COPY
17             ca.append(a1, a2) ;
```

4.3 NF Parallelism Identification Algorithm

Based on above NF parallelism analysis and resource optimization techniques, we propose an NF parallelism identification algorithm for two NFs obeying an `order` rule as shown in Algorithm 1 [1]. NFP orchestrator maintains an NF action table (AT, i.e. Table 2) and an action dependency table (DT, i.e. Table 3), and takes an `Order` rule as input. The algorithm can determine that the two NFs can be parallelized without packet copying or with packet copying, or cannot be parallelized. First, the algorithm fetches all the actions of the two NFs from AT (lines 1-2). Then it exhaustively goes over all action pairs from the two NFs (lines 5-17) to figure out the parallelism possibility for the two NFs based on the DT. For the read-write or write-write case, we need to further decide if the two actions operate on the same field (lines 6-9). If the two NFs can be parallelized with packet copying, we need to record the conflicting actions (lines 16-17). Finally, the algorithm generates the output of whether the two NFs are parallelizable (p) and possible conflicting actions (ca), whose existence indicates the necessity of packet copying.

We input all possible NF pairs from Table 2 into the algorithm. According to the algorithm output and the appearance probabilities of the NF pairs, we find that 53.8% NF pairs can work in parallel. In particular, 41.5% pairs can be parallelized without causing extra resource overhead, which promises the optimization effect of NF parallelism for NFs summarized in Table 2.

To accommodate a new NF into NFP, network operators could generate an *action profile* of the NF manually or with the analysis tool provided by NFP (§5.4), and *register* it into Table 2. NFP would then be able to construct service graphs containing this new NF.

For two NFs specified in a `Priority` rule, we still need to decide whether they require packet copying to work in parallel. In this situation, we still use Algorithm 1 to identify possible conflicting actions of the two NFs.

[1] For parallelism identification among multiple NFs, we run Algorithm 1 for each pair of NFs, as demonstrated in §4.4.2.

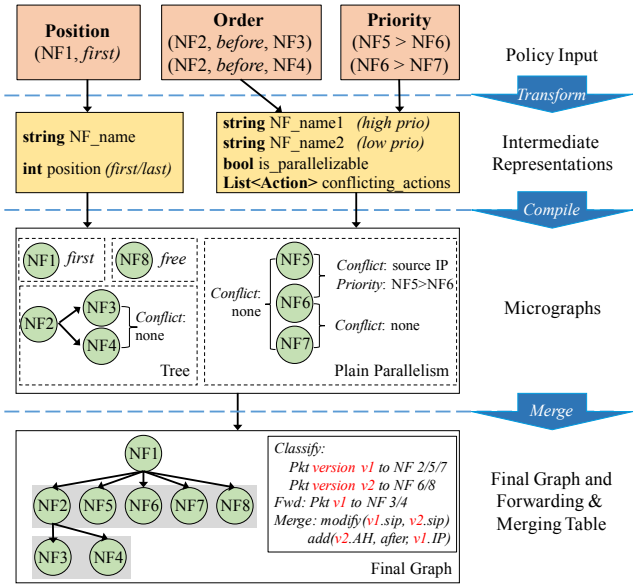


Figure 2: Service Graph Construction Workflow

4.4 Service Graph Construction

The NFP compiler constructs service graphs based on NFP policies to pursue *high performance* with *marginal resource overhead*. It first transforms policies into pre-defined intermediate representations, then compiles the intermediate representations into independent micrographs, and finally merges the micrographs to generate the final service graph. We next introduce each step of the service graph construction process, as demonstrated in Figure 2.

4.4.1 Transforming Policies into Intermediate Representations. We design two types of intermediate representations to store NFP policies, as shown in Figure 2. For `Position` rules, we maintain the NF type and its position in the left representation block, which records the placement of a single NF. For `Order` rules, we implement the Algorithm 1 to check whether they can be parallelized and identify conflicting actions. For `Priority` rules, we still need Algorithm 1 to find out conflicting actions. The `Order` and `Priority` rules are finally transformed into the representation shown on the right, which reveals the relationship between two NFs.

4.4.2 Compiling Intermediate Representations into Micrographs. After transforming policies into intermediate representations, we first sequentially chain NFs that are not parallelizable (e.g. NF2 and NF3 in Figure 2). Then we concatenate intermediate representations with overlapping NFs into a *micrograph* by using overlapping NFs as junction points. There are three types of micrograph structures including *Single NF* (e.g. NF1, NF8), *Tree* (e.g. NF2, NF3 and NF4), and *Plain Parallelism* (e.g. NF5, NF6 and NF7). Single NF micrographs come from NFs assigned in `Position` rules (e.g. NF1), or *free* NFs with no rule restrictions (e.g. NF8). Tree micrographs come from unparallelizable NFs. We exhaustively check dependencies of all leaf NF pairs with the same root to figure out whether the leaf NFs can work in parallel. For plain parallelism micrographs, we exhaustively check the dependencies of all NF pairs to calculate how many packet copies are needed for them. So far, we have generated micrographs with no overlapping NFs.

4.4.3 Merging Micrographs into the Final Graph. Finally, we merge micrographs to generate the service graph. NFs assigned by `Position` rules are first placed in the head/tail of the chain (NF1). Then we wrap up *each* remaining micrograph (including free NFs) as *one NF*, and exhaustively check the dependency of each micrograph pair to decide their parallelism. If any dependency is detected between micrographs, network operators will be informed to further regulate execution priority of them. Finally, we place independent micrographs in parallel (Figure 2).

Based on the final graph structure, NF dependencies, and NF priorities, we create a classification table that records how to direct a packet to its corresponding service chain, a forwarding table that records how to steer different packet copies (*version1* and *version2* in Figure 2), and a merging table that stores how to merge packet copies. We introduce the detailed table design in §5.

5 INFRASTRUCTURE DESIGN

Figure 3 illustrates the design overview of NFP infrastructure. As mentioned in §2, NFP adopts consolidation to avoid occupying extra network bandwidth resource. For packet delivery among NFs, to pursue high performance, we use the zero-copy packet delivery proposed in NetVM [28, 64]. As shown in Figure 3, each NF owns a receive ring buffer and a transmit ring buffer, which are stored in a shared memory region allocated in huge pages [41] accessible to all NFs. Received packets also reside in the shared memory, while an NF simply writes packet *references* into the receive ring buffer of the other NF to realize packet delivery. Such a zero-copy delivery eliminates the copy overhead for packet delivery.

However, the infrastructure design for supporting NF parallelism is challenged in several aspects.

- NFs may drop packets. The infrastructure is challenged to deal with the situation where one of the two parallel NFs drops the packet, and the other one reads or modifies it.
- The infrastructure needs to *merge* multiple versions of a packet to create the final output, which incurs the challenge for the merging module to handle heavy load without becoming a performance bottleneck.
- In previous work [28, 64], packet steering among NFs relies on a *centralized* virtual switch, which according to our evaluation incurs a performance overhead due to packet queuing. The infrastructure requires a more efficient approach in delivering packets among NFs.

NFP carefully designs the infrastructure to address above challenges. The red solid line in Figure 3 shows a packet processing path inside the infrastructure. When a packet enters an NFP server, we introduce a *classifier* that sends the packet reference into the proper service graph. For the packet delivery, we design a *distributed NF runtime* to efficiently deliver packets among NFs *in parallel*. Finally, multiple copies of a packet are sent into the *merger* module to generate the final output. These modules can be dynamically configured by the orchestrator. Next we introduce each module in detail.

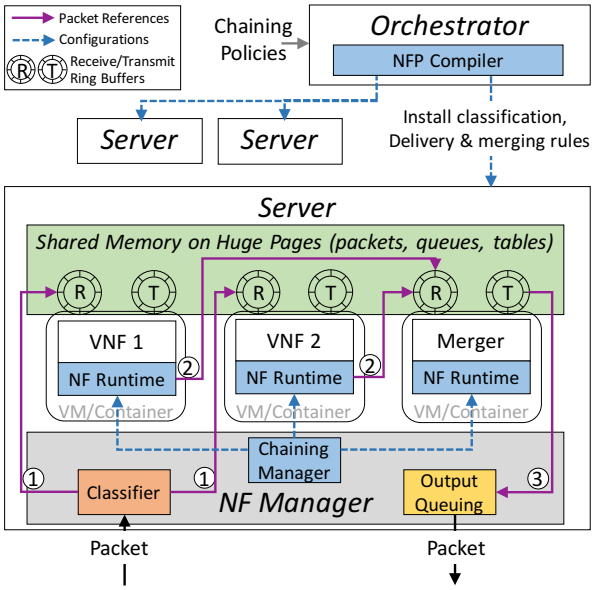


Figure 3: NFP infrastructure design overview

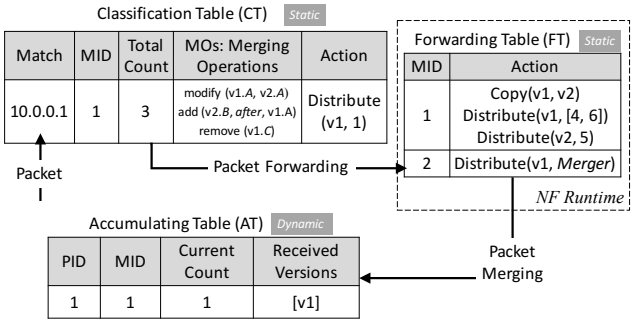


Figure 4: NFP infrastructure workflow

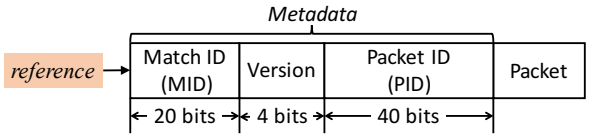


Figure 5: NFP packet data structure

5.1 Packet Classification

The *classifier* module takes an incoming packet from the NIC and finds out the corresponding service graph information for the packet, including how many packet copies are expected in the merger, how to merge different copies of a packet, and the first hop(s) of the service graph. Therefore, the classifier maintains a *Classification Table (CT)* shown in Figure 4 to store the *match* fields (e.g. five tuple), the *total packet copy count* to be received in the merger, the *merging operations (MOs)* to merge packet copies (details are presented in §5.3), and the *actions* indicating the first NF(s) of the service graph, based on which the classifier sends the packet into the entrance of the graph.

Note that different packets in a flow, or different flows that follow the same service graph are forwarded and merged in the same pattern according to its service graph structure. Therefore, we tag those packets that follow the same service graph with the same *Match ID (MID)* to avoid repeated storage of the service graph information. Latter modules could identify the service graph to which the packet belongs based on MID to forward or merge packets. To transmit the MID to latter modules, we tag it into packet metadata shown in Figure 5. Twenty bits of MID could express 1M service graphs.

However, despite each packet in a flow should be merged in the same way, the merger needs to collect all versions of *each* packet to generate the output for this packet. For this purpose, we need to grant an *identification* to each packet in a flow. Therefore, we design a *Packet ID (PID)* identifier of 40 bits and tag it into the packet metadata. Furthermore, to identify different versions of a packet, we assign a *version* to each packet copy, which is also tagged into packet metadata, for the merger to generate the final output.

Therefore, the NFP classifier attaches a 64 bits metadata to a packet, recording the *MID*, *PID* and *version* of a specific packet copy. The data structure of the packet is shown in Figure 5. Each Classification Table entry is generated by the orchestrator to direct a flow into a specific service graph. It is then installed to the classifier, and will remain *static* when processing and delivering packets of this flow.

5.2 Packet Delivery Among NFs

After an NF processes a packet, NFP should steer the packet to the subsequent NFs in the service graph without copying, or copy the packet and send the copies into parallel NFs. As mentioned above, using a centralized virtual switch as the forwarder might incur performance overhead. To address this challenge, NFP *distributes the packet forwarding task* and enables each NF to independently forward packets to subsequent NFs *in parallel*. To make this process *transparent* to NF developers and incur no NF modifications, we design an *NF runtime* for each NF to perform traffic steering, as shown in Figure 3. After packet processing, the NF could delegate the packet to the NF runtime, which copies the packet reference to the next NFs' ring buffer to realize packet forwarding. Through the distributed NF runtime, we could *parallelize* the packet delivery process and alleviate the forwarding hot spot.

Each NF runtime maintains a *forwarding table (FT)* (see Figure 4), which stores a local view of the entire service graph. The global forwarding table is generated at the end of the service graph construction process (§4.4.3), and then *statically* installed to the *Chaining Manager* (as shown in dashed blue line in Figure 3). The chaining Manager splits the global table and installs the forwarding rules to each NF runtime. When an NF delegates a packet to the NF runtime, the MID in the packet metadata is used to look up the *actions* in FT. We design four types of actions.

ignore: When an NF intends to *drop* the packet, it conveys the dropping intention to the NF runtime. The NF runtime then implements the `ignore` action to ignore original actions in the FT entry for this packet. Furthermore, the NF runtime sends a *nil packet* to deliver the dropping intention to the merger.

distribute(version, targets): This action sends the reference of a specific *version* of a packet to one or multiple *target* parallel NFs without packet copying.

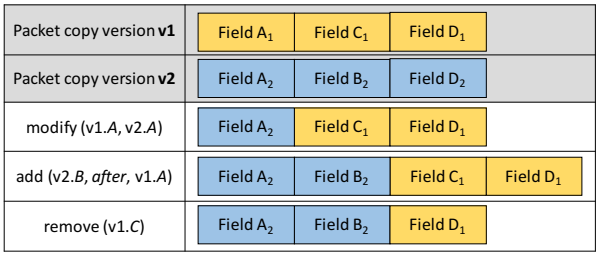


Figure 6: An example of NFP merging process

copy(version1, version2): This action copies packet *version1* and tags the new copy as *version2*. We only copy packet headers and set the "packet length" field as the length of the header itself. Besides, we prepare memory blocks to store input or copied packets during the system initialization. Therefore, the header copying does not require dynamic allocation of the memory and could avoid performance degradation.

output(version): This action is used to output the packet after it has traversed the entire service graph. The `output` action is performed by the last NF in the service graph possibly assigned by a `Position` rule.

Note that the *actions* field of the Classification Table also includes above actions. The classifier may copy and send a packet into one or more NFs as a start, according to the service graph structure.

5.3 Load Balanced Packet Merging

After all NFs have processed a packet, multiple copies of this packet is sent into a *merger* module to generate the final output. The merger maintains a *dynamic Accumulating Table (AT)* as shown in Figure 4. Each entry records the *received packet copy count* and the *received packet copy versions* of a packet. Note that the number of received packet versions may not be equal to the received packet count, since several NFs may process the *same* packet copy and send it to the merger independently. Next we introduce the merging process in detail.

Packet Merging: When the *current count* field in AT reaches the *total count* recorded in CT referenced by the key of MID, the merger will merge the packet copies according to the *merging operations (MOs)* in the CT. The original packet copy is tagged as version *v1* by the classifier, and MOs record how to merge the rest of packet copies into *v1* to create the final output. In other words, MOs indicate which bits of different packet versions should be included in the final processed packet. MOs include three types of operations.

- `modify(v1.A, v2.A)`: This operation overwrites the packet field *A* of *v1* with that of *v2*. For example, `modify(v1.SIP, v2.SIP)` changes the source IP address of packet *v1* into that of *v2*.
- `add(v2.B, before/after, v1.A)`: This operation adds the packet field *B* of *v2 before/after* the field *A* of *v1*. For example, operation `add(v2.AH, after, v1.IP)` adds the Authentication Header (AH) of *v2* after the IP header of *v1*.
- `remove(v1.C)`: This operation removes the field *D* from *v1*. For example, operation `remove(v1.AH)` removes the AH header from packet version *v1*.

We show an example set of MOs in Figure 6. The merger goes over each MO, executes the operation, and generates the final output as shown in the last row of Figure 6. Note that Field D_1 of packet *v1* is not referred to by any operation. Therefore, this field remains unmodified and is written directly into the final packet. Field D_2 of *v2* is also not mentioned by any operation, meaning that D_2 will be ignored and not included in the final packet. Current design and implementation of MOs are protocol dependent. As our future work, we will refer to protocol independent definition of packet fields such as P4 [5] to support advanced and customized NFs.

To deal with NF *dropping* actions on a packet *p*, we enable the NF runtime to send a *nil packet*, which has the same metadata as *p* to the merger. When the merger receives a nil packet, it considers the packet *p* as dropped. We then remove the related AT entry and release the memory of all received packet copies. This could provide consistent processing results with sequential NF processing.

A possible design choice of packet merging is to maintain an extra copy of the original packet, simply *xor* the processed and original packets to find the modified bits. In this way, we are relieved from the burden of inspecting NF actions and generating merging operations. However, there are several drawbacks of this approach. First, we previously inspect NF actions to identify NF dependency. Without NF action profiles, NF parallelism identification would become adhoc by exhaustively analyzing each NF pair. Second, the *xor* mechanism cannot easily handle header addition/removal or dropping actions. Finally, maintaining the original copy of the packet brings unnecessary resource overhead.

Merger Load Balancing: The merger is heavily burdened to process all copies of every packet and could introduce performance bottleneck. To address this challenge, we propose to deploy multiple mergers in one NFP server and design a *merger agent* to balance the load among the instances. A merger instance maintains a local AT, and could merge any packet from any service graph.

To ease its instantiation and destruction, we implement the merger as an NF. A merger instance can be dynamically instantiated or destroyed by the orchestrator similar to other NFs. Packets to be merged are first sent to the merger agent, which then performs simple load balancing to split the load. Note that we should ensure that multiple copies of the same packet are sent to the same merger instance. However, packet copies may be modified by NFs. Therefore, the merger agent performs a simple and fast hashing on the immutable *PID* field of a packet and steers the packet to a merger instance. Note that different packets in a flow own different PIDs and could be distributed to different merger instances. We evaluate the merger overhead and the load balancing effect in §6.

5.4 Integrating Network Functions into NFP

NFP provides NFs with interfaces to access and modify packets, and an NF runtime to drop or deliver packets after processing. To integrate a new NF into the system, NFP needs the actions of the NF for parallelism identification and service graph construction. To this end, NFP provides an inspection tool for operators that can inspect NF codes to find the usage of interfaces that operate on packets, including reading, writing, dropping and adding/removing bits. Operators can run the inspector against their NF code to automatically generate an action profile, which can be registered into NFP to integrate the new NF into NFP.

In our current implementation, we provide *DPDK based interfaces* for NFs to access and modify packets. DPDK could parse the packet and provide NFs with a data structure to read and write the packet headers or payloads. The inspection tool analyzes the calls of the packet data structure to determine actions of NFs. In future, we plan to integrate advanced modular NF specifications [18, 35, 47] into NFP, and identify actions of NFs by inspecting their component modules and combining the actions of the modules together.

6 IMPLEMENTATION AND EVALUATION

We implement the NFP framework and the NF action inspector (14,000 LoC in total) based on DPDK version 16.11. We evaluate NFP based on a testbed with a number of servers, each of which is equipped with two Intel(R) Xeon(R) E5-2690 v2 CPUs (3.00GHz, 10 physical cores), 256G RAM and two 10G NICs. The servers run Linux kernel 4.4.0-31.

We run NFs using Docker [39]. Each NF runs inside a container on a physical CPU core. Besides, each merger instance occupies a container, while the merger load balancer itself also occupies a container. All containers are configured to run in *privileged mode* to access host resources including the NIC and shared memory on huge pages. We isolate and dedicate each core to a container alone, which could ensure that the core will not be scheduled by the operating system. The classifier also consumes a separate CPU core. It is implemented as a process running in the user space on the host operating system. It constantly pulls packets from all active NICs through DPDK interfaces, classifies each packet, attaches metadata to it, and sends the packet to the corresponding service chain, by writing the packet's reference into the *receive ring buffer* of the first NF in the chain. The NF runtime is implemented to initialize the ring buffers of the NF in the shared memory on huge pages. After an NF is deployed, its runtime collects packets from the receive ring buffer, delivers packets to NF logic, and takes over the packet for further delivery after NF processing.

For test traffic, we use a DPDK based packet generator that runs on a separate server and is directly connected to the test server. The generator sends and receives traffic to measure the latency and the maximum throughput without packet loss.

NFP achieves high performance with marginal resource overhead. We evaluate NFP with the following goals:

- demonstrate that NFP can support sequential service chains without introducing extra performance overhead compared with state of the art software based high performance platforms such as OpenNetVM [64], a container implementation of NetVM [28] (Figure 7).
- study the performance improvement brought by NFP based on the variables of NF complexity, parallelism degree, and service graph structure (Figures 8, 9, 11, 12).
- demonstrate that the overhead brought by NFP is minimal, including the resource overhead from packet copying and the performance overhead incurred by packet copying and merging. (§6.3)
- demonstrate that NFP achieves significant latency reduction and marginal resource overhead for real world service chains and traffic in data centers (Figure 13).

6.1 Network Functions

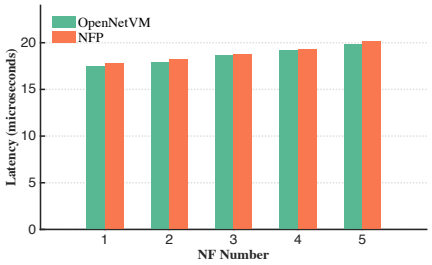
To evaluate NFP, we implement a range of network functions:
L3 Forwarder: A simple forwarder that obtains the matching entry from a longest prefix matching table with 1000 entries to find out the next hop.
Load Balancer: We implement the commonly used ECMP mechanism in data centers [21] that hashed the 5-tuple of the packet to balance the load.
Firewall: This is a firewall similar to the Click IPFilter element. It passes or drops packets according to the Access Control List (ACL) containing 100 rules.
IDS: A simple NF similar to the core signature matching component of the Snort intrusion detection system [55] with 100 signature inspection rules.
VPN: It implements the tunnel mode of IPsec Authentication Header (AH) protocol. It encrypts a packet based on the AES algorithm and wraps it with an AH header.
Monitor: It maintains per-flow counters, which can be obtained by the operator. The counter table uses the hash value of the 5-tuple as the key.

6.2 Performance Improvement

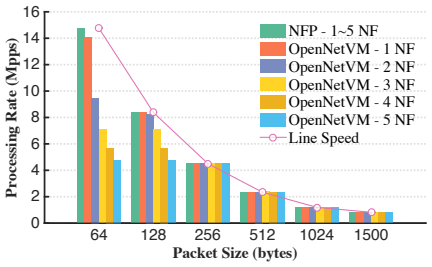
6.2.1 Sequential Service Chain Performance. Despite that NFP accelerates NFV through NF parallelism, there may be situations where the policy is compiled into a sequential service chain. Therefore, NFP is challenged to support sequential service chains with no performance overhead compared with existing systems such as OpenNetVM. In fact, a sequential service chain does not require packet copying and merging, which theoretically incurs no performance penalty.

To control the variable of NF complexity, we generate sequential chains by composing multiple instances of the L3 forwarder that is implemented in OpenNetVM and NFP in the same way. We vary the chain length from 1 to 5. We use 64B to 1500B packets to evaluate the throughput, and focus on the latency for (min-size) 64B packets. Experimental results in Figure 7 show that NFP suffers a tiny latency overhead, and outperforms OpenNetVM by achieving line rate for packets of any sizes. OpenNetVM dedicates a CPU core for the centralized switch to forward packets, while NFP relies on the distributed NF runtime that shares the CPU core with the NF. Therefore, NFP could suffer a little latency overhead, but could alleviate the performance bottleneck of the centralized switch during high packet rates and achieve a higher throughput.

6.2.2 Effect of Different NF Complexity. We evaluate the optimization effect of NFP for NFs with different complexity. First, we measure the performance of the six NFs implemented in NFP. To study the effect of the complexity itself, we control the parallelism grade as two, meaning that we compare the performance of sequential or parallel composition of two instances of the same NF. We also compare the optimization effect with or without packet copying and merging using 64B packets. We present the evaluation setup in Figure 10. Figure 8 shows the performance of NFs with different complexity. The L3 Forwarder simply performs one table look up, while VPN needs to encrypt and encapsulate packets. We observe that the latency benefit brought by NF parallelism increases with the rise of NF complexity.

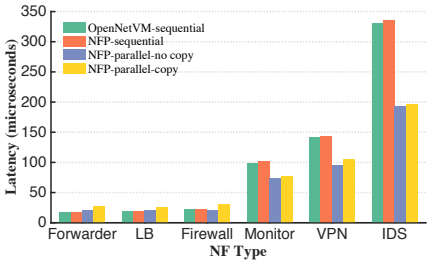


(a) Latency

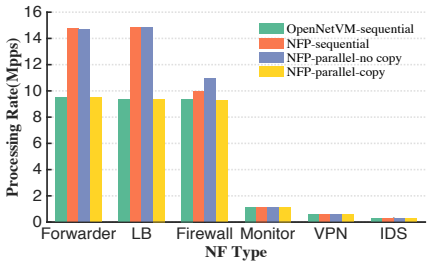


(b) Processing Rate

Figure 7: Performance of sequential chains (latency for 64B pkts)

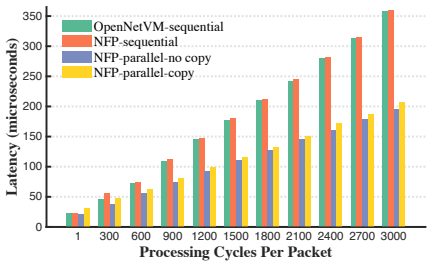


(a) Latency

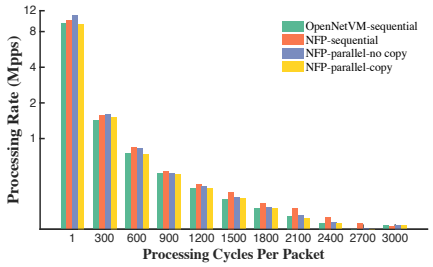


(b) Processing Rate

Figure 8: Performance of different NFs with different complexity

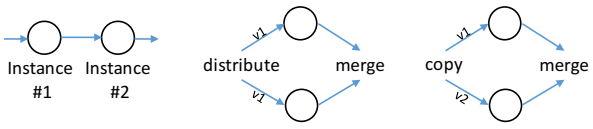


(a) Latency



(b) Processing Rate

Figure 9: Performance of Firewall NF with different complexity



(1) Sequential (2) NFP with no copy (3) NFP with copy

Figure 10: Setup to evaluate optimization effect for NFs with different complexity

To create a function of optimization effect and NF complexity, we modify the Firewall NF so that it busily loops for a given number of cycles after modifying the packet, allowing us to vary the per-packet processing time as a representation of NF complexity. We evaluate the performance using 64B packets. As shown in Figure 9, the forwarding latency optimization effect rises with the increase of NF complexity. For the most complex NF (3000 cycles), NFP brings around 45% latency reduction. Besides, we can observe that the performance overhead brought by packet copying is minimal.

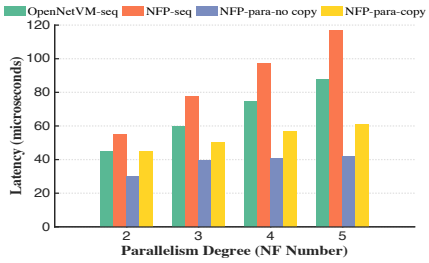
6.2.3 Effect of Parallelism Degree. NF parallelism could reduce overall processing latency by executing several parallel NFs simultaneously. Theoretically, parallelizing more NFs could reduce latency to a larger extent. To create a function of optimization effect and the parallelism degree, we vary the instance numbers of the Firewall NF (with 300 cycles) from 2 to 5 and evaluate the performance of sequential/parallel composition of these instances with or without packet copying and merging using 64B packets. We observe from Figure 11 that with the increase of parallelism degree, the latency reduction rises from 33% to 52% for no-copy setups, and up to 32% for copy setups. We could conclude that higher parallelism degree brings larger latency benefit while the throughput is not much affected. However, the latency reduction cannot reach the theoretical value of 80% for 5 degree parallelism. We attribute this to the merging process. With higher degree, the merger has to collect and merge more packets, which increases latency.

6.2.4 Effect of Graph Structure. NFP orchestrator could construct various types of service graphs comprising the same number of NFs. For instance, for a service graph containing 4 NFs, there exists 6 possible structures (non-exhaustive) as shown in Figure 14. We evaluate their performance with or without packet copying and merging using 64B packets. Figure 12 reveals a better latency optimization effect for graphs with shorter equivalent chain length. For example, graph(2) enjoys the biggest latency benefit since the equivalent chain length is 1, while graph(5) sees little latency reduction since the equivalent length is 3.

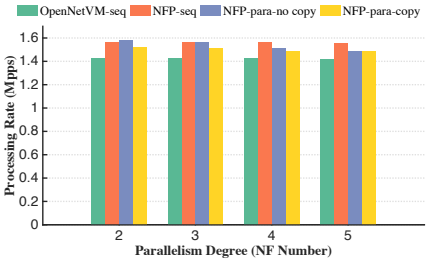
6.3 Overhead

Although NF parallelism could bring high performance, in 12.3% situations, packet copying is needed for parallel processing (§4), incurring extra resource overhead to accommodate copied packets, and the performance overhead brought by packet copying/merging actions.

6.3.1 Resource Overhead. NF parallelism could occupy extra resource to store copied packets. To evaluate the resource overhead brought by NFP, we calculate the extra resource usage percentage as a function of TCP packet size (64B to 1500B) and parallelism degree. According to the header-only copying optimization, only packet headers are copied. Therefore, for a TCP packet of any size on the Ethernet, packet copying only occupies 64B extra memory. We construct the equation of resource overhead (ro), packet size (s) and parallelism degree (d): $ro = 64 \times (d-1)/s$. We refer to the packet size distribution in data centers from [4] and calculate that

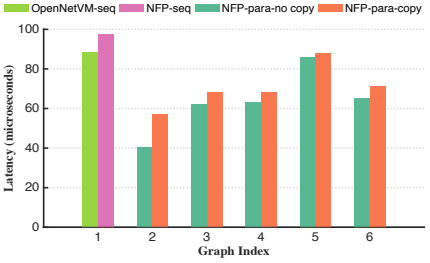


(a) Latency

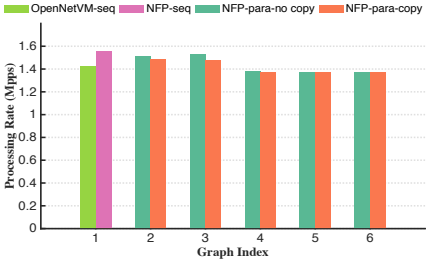


(b) Processing Rate

Figure 11: Performance of graphs with different parallelism degree



(a) Latency



(b) Processing Rate

Figure 12: Performance of different graph structures comprising 4 NFs

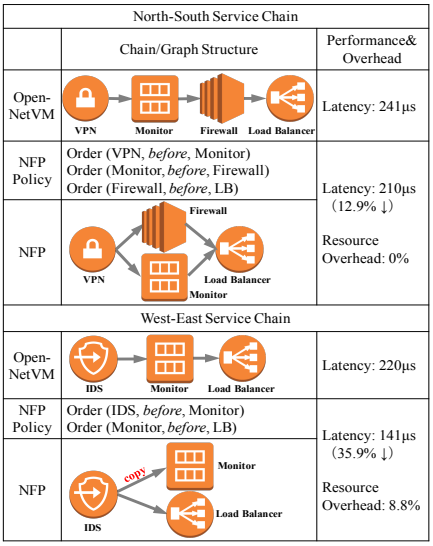


Figure 13: Performance of real world service chains

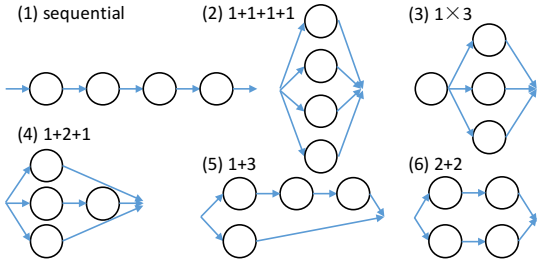


Figure 14: Possible service graphs comprising 4 NFs

the resource overhead of NFP is $ro = 0.088 \times (d-1)$, which is only 8.8% for the parallelism degree of 2, while achieving 30% latency reduction derived from Figure 8.

6.3.2 Copying and Merging Performance Overhead. Memory copying and additional packet merging would be time consuming and could degrade performance [28, 30]. However, NFP needs packet copying and merging to support NF parallelism. Therefore, NFP proposes Dirty Memory Reusing to reduce copying necessities, and uses Header Only Copying to shorten copied memory length to a fixed 64B for TCP traffic. Furthermore, for packet copying implementation, we use optimized fast memory copy interfaces provided by DPDK to reduce copying overhead. As shown in Figure 11, for the firewall NF, packet copying and merging could bring an average of 15 μs latency penalty and minimal throughput penalty, while still achieving 20% latency reduction compared with sequential composition. Furthermore, packet copying is only necessary in 12.3% situations. With longer chains and more complex NFs (e.g. VPN), the latency overhead percentage of copying and merging would be further reduced.

6.3.3 Merger Load Balancing. The merger module is burdened to collect and merge multiple copies of a packet. Therefore, it suffers a heavier load than NFs. To understand its capability, we evaluate the peak processing rate with no packet loss of a merger instance for 64B packets. We use the Firewall NF and set the parallelism degree as 2. We find that one merger instance can handle 10.7 Mpps processing rate with no packet loss. According to our experiments, for packets of any size (including 64B), *two* merger instances are sufficient to support full speed packet processing with the parallelism degree of up to 5, which could demonstrate the effectiveness of the merger load balancing mechanism.

6.4 Real World Service Chains

We evaluate NFP based on real world service chains in data centers [32, 36] including service chains for north-south and east-west traffic. For NFP policies, we assume the operator assigns a sequential chain description based on `Order` rules for neighboring NFs in the chain. We generate test packets according to the packet size distribution derived from [4]. We present the original sequential service chain, NFP policy description, optimized service graph structure, and performance gain in Figure 13. For the north-south service chain, NFP parallelizes the Firewall and the Monitor, and could achieve 12.9% latency reduction with *zero* resource overhead. For the west-east service chain, NFP executes the Monitor and the Load Balancer in parallel, resulting in 35.9% latency reduction with only 8.8% resource overhead. This demonstrates that NFP can achieve significant performance improvement with marginal overhead.

Moreover, to verify the correctness of NFP composition of NFs in NFV, we generate a series of packets based on our DPDK packet generator, tag each packet with a unique *packet ID* in the payload,

Table 4: Performance of OpenNetVM, NFP, and BESS for service chains of different lengths. When the chain length is n, we use $n+2$ CPU cores to support each system.

Chain Length	CPU Cores	Latency (μs)			Processing Rate (Mpps)		
		OpenNetVM	NFP	BESS	OpenNetVM	NFP	BESS
1	3	25	23	11.308	9.38	10.92	14.7
2	4	33	27	11.370	9.36	10.92	14.7
3	5	47	31	11.407	9.38	10.90	14.7

and replay them to the sequential service chain and the optimized NFP service graph. We compare the processed packets and find that NFP service graph could provide the same execution results as the sequential service chain, which follows the *result correctness principle* proposed in §4.1.

7 DISCUSSION

NFP on Containers v.s. Virtual Machines (VMs): NFP prototype is implemented based on Linux Containers instead of VMs since containers such as Docker [39] are more light-weight [43] and can provide faster service instantiation and higher performance [43, 47]. However, NFP can also be implemented on VMs with the similar infrastructure design based on the fast packet delivery technique between VMs proposed in NetVM [28]. We will create a VM implementation in our future work.

Pipelining Model *vs.* Run-to-Completion Model for Service Chains: We conclude from the literature that two models have been proposed to support service chains in NFV networks, including the *pipelining* model [28, 38, 63, 64] and the *run-to-completion (RTC)* model [27, 47]. The *pipelining* model deploys an NF inside an isolated VM or container with dedicated CPU cores. Meanwhile, the *RTC* model abandons virtualization techniques and consolidates the entire service chain inside one CPU core. NFP aims to accelerate *pipelining model based NFV networks* by exploiting parallelism opportunities of NFs to reduce the chain latency. In this section, we briefly state our insights on the two models, while detailed discussion and comparison of them are *out of the scope* of this paper.

We conduct a simple experiment to measure the performance of a service chain composed of 1 to 3 firewall NFs in BESS [27] (*RTC*), OpenNetVM (*Pipelining*), and NFP (*Pipelining*), when processing 64B packets. We run sequential service chains in OpenNetVM. We enable NFP to run all NFs in parallel for the highest performance. When chaining 3 NFs, NFP utilizes 5 CPU cores to carry 3 NFs, a classifier, and a merger. Given 5 cores, BESS could duplicate 5 entire chains to place on the 5 cores, and perform hashing in the NIC to split traffic across cores. As shown in Table 4, *RTC* could achieve lower latency and higher processing rate. Note that without the limitation of the 10G NICs adopted in our experiment, the processing rate of BESS could be even higher. As the chain length n grows, with more cores to scale out, BESS could theoretically achieve $27.2 \times (n+2)$ Mpps processing rate [27].

Despite its high performance, *RTC* could possibly fall short in supporting NFV's elasticity of easy scaling out when an NF instance is overloaded. In the *pipelining* mode, we could simply create a new instance on a VM or container, migrate some states [23, 53], and modify the forwarding table to redirect some flows to the new instance. However, in *RTC*, if *one* NF instance is overloaded, we need to introduce another core to alleviate the hot spot by either (1) duplicating the *entire* chain to the new core, which could introduce unnecessary states to be migrated and more NF instances to be managed, or (2) duplicating the overloaded NF to the new core, and redirecting flows between two cores for load balancing, which would introduce cross-core communication and degrade the performance.

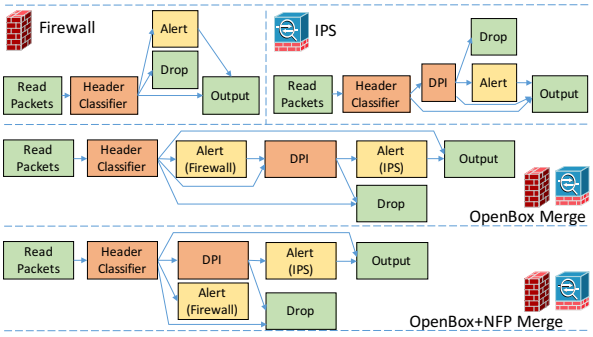


Figure 15: OpenBox+NFP graph merging result

NFP Scalability: NFP consolidates NFs in a service graph inside one server to optimize resource overhead. According to [25, 32, 36, 52, 60], the lengths of currently deploy service chains normally do not exceed seven. By allocating one core to each container, a service graph can be entirely accommodated inside a server with 20 physical CPU cores in our testbed. NFP can support NF scaling inside one server by allocating remaining CPU cores to new NF instances with new IDs and constructing service graphs containing these new instances.

However, sometimes there may be too many NFs in a graph to fit into one server. We need to revisit and propose resource overhead optimization techniques to compress overhead and achieve high performance when supporting NF parallelism across multiple servers. For example, NFP could partition the service graph onto multiple servers obeying: each server sends only *one copy* of a packet to the next server. In this way, we could still benefit from NF parallelism without introducing extra network bandwidth resource overhead. Packet delivery between servers could refer to Flowtags [16] or Network Service Header (NSH) [51]. As our next step, we will focus on the design supporting cross-server NF parallelism.

Combining Parallelism and Modularity: OpenBox [7] decomposes NFs into building blocks, many of which share no dependencies. Therefore, NFP can be used here to exploit block level parallelism. After decomposing NFs and sharing common modules, we could identify dependencies of building blocks, seize parallelism opportunities, and build an optimized module graph. The OpenBox+NFP graph merging in Figure 15 could parallelize independent building blocks, such as Alert (firewall) and DPI, to further reduce latency. Thus, NFP provides a general optimization that can adapt to both monolithic and modular NFs.

8 RELATED WORK

Parallelism in Packet Processing: Some recent works touched the idea of using parallelism in packet processing. Cisco Vector Packet Processing (VPP) [9] applies operations directly to a vector of packets, similar to the data-level parallelism that performs the single instruction on multiple data (SIMD) simultaneously [48].

ClickNP [37] benefits from the parallelism capability of FPGA and implements parallelism inside a building element and across elements inside an NF. NetVM [28] adopts batch processing, i.e. polling multiple packets (a batch) each time from NICs to ameliorate IO bottleneck. Above research efforts are orthogonal and complementary to NFP. NFP already implements batch packet processing. Parallelism based NF acceleration could also fit into the NFP framework. P4 [6] abstracts NFs into multiple stages of tables, and identifies *table dependencies* to enable parallelism inside an NF or across NFs. P4 explores inter-table parallelism while NFP focuses on inter-NF parallelism. However, P4 can be used as a possible hardware infrastructure for NFP.

ParaBox [65] also attempts to explore NF parallelism in NFV. However, its NF parallelism detection remains preliminary and lacks a comprehensive analysis on NF action dependency. ParaBox has to provide different packet copies for NFs running in parallel, which introduces large resource overhead. In comparison, NFP proposes a comprehensive framework with three layers to enable NF parallelism and enhance NFV performance. Network operators could write NFP policies to intuitively orchestrate NFs. The NFP orchestrator could intelligently identify NF dependency and construct high performance service graphs with little resource overhead. The NFP infrastructure can perform light-weight packet copying, distributed parallel packet delivery, and load balanced packet merging to support NF parallelism.

Besides, some prior efforts [19, 40] have proposed the concept of parallel and sequential module composition to construct SDN applications. They focus on ameliorating the conflicts of module policies that operate on the same packets, due to the fact that different modules inside an SDN application, or different applications in SDN networks *share* the flow table resource. In comparison, NFP exploits NF parallelism in order to enhance the performance of NFV networks. NFs in NFV are placed on isolated VMs or containers. To support NF parallelism, a packet needs to be distributed or copied, and processed by multiple NFs simultaneously. Therefore, the ideas proposed by [19, 40] cannot be adopted directly to support NF parallelism in NFV.

NFV Acceleration Techniques: As summarized in §1, lots of efforts have been devoted to accelerating NFV, including individual NF performance enhancement [33, 37, 42, 47, 57], packet delivery acceleration [28, 30, 38, 64], and NF modularization and building block sharing [2, 7, 46, 60]. NFP is orthogonal and complementary to all above research efforts. First, §2 analyzes the approaches to adopt parallelism among accelerated individual NFs. Second, NFP already adopts fast packet delivery techniques in the system design to achieve high performance. Finally, §7 introduces the combination of parallelism and modularity for larger benefits.

NF Orchestration Techniques: For NFV networks, commercial solutions including OpenStack [59], OSM [15], and OPNFV [20] and research efforts including SIMPLE [50] and Flowtags [16] all propose NF orchestration techniques surrounding *sequential* service chains. NFP proposes to embrace parallelism and construct service graphs possibly with parallel NFs to improve NFV performance. On the other hand, [3, 58] propose policy based NF orchestration and management. However, their policies refer to Service Level Agreement (SLA) related objects, while NFs are still chained sequentially. Nevertheless, their policy abstraction partially enlightened us to provide policies to operators to describe chaining intents.

9 CONCLUSIONS AND FUTURE WORK

This paper presents NFP, a high performance framework, that innovatively enables NF parallelism in NFV to improve NFV performance. NFP defines a policy specification scheme for network operators to intuitively describe sequential or parallel NF composition intents. NFP orchestrator compiles the policies into optimized service graphs with marginal resource overhead. NFP infrastructure performs classification, parallel delivery and merging to support NF parallelism. We have implemented a prototype in Linux containers and demonstrated its performance and overhead. As our future work, we will enhance the policy specification scheme to represent more complex NF composition rules. We will also enable NFP to inspect and ameliorate policy conflicts. Besides, we will propose an inter-server NF parallelism design of NFP. Moreover, we will integrate advanced modular NF specifications into NFP to ease the identification of NF actions. Finally, we will study the combination of parallelism and modularity to further accelerate NFV, and demonstrate its performance with advanced Layer 7 NFs.

10 ACKNOWLEDGEMENT

We thank our shepherd Barath Raghavan and anonymous SIGCOMM reviewers for their valuable comments. We thank Masoud Moshref Javadi for proof-reading the paper and providing valuable suggestions. We also thank Xiao Zhang, Qingnan Duan, and Zili Meng from Tsinghua University for joining the discussion of this paper. This work is supported by National Key Research and Development Plan of China (2017YFB0801701), and National Science Foundation of China (No.61472213).

REFERENCES

[1] George S Almasi and Allan Gottlieb. 1988. Highly parallel computing. (1988).
[2] Bilal Anwer, Theophilus Benson, Nick Feamster, Dave Levin, and Jennifer Rexford. 2013. A slick control plane for network middleboxes. In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'13)*. ACM.
[3] Md Bari, Shihabur Rahman Chowdhury, Reaz Ahmed, Raouf Boutaba, and others. 2015. On orchestrating virtual network functions in NFV. *arXiv preprint arXiv:1503.06377* (2015).
[4] Theophilus Benson, Aditya Akella, and David A Maltz. 2010. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM, 267–280.
[5] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and others. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.
[6] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and others. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.
[7] Anat Bremler-Barr, Yotam Harchol, and David Hay. 2016. OpenBox: a software-defined framework for developing, deploying, and managing network functions. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 511–524.
[8] Anat Bremler-Barr, Yotam Harchol, David Hay, and Yaron Koral. 2014. Deep packet inspection as a service. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM, 271–282.
[9] Cisco. 2002. Vector Packet Processing. (2002). https://wiki.fd.io/view/VPP
[10] Cisco. 2017. IOS Technologies. (2017). http://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-technologies/index.html
[11] Cisco. 2017. MGX 8800 Series Switches. (2017). http://www.cisco.com/c/en/us/products/switches/mgx-8800-series-switches/index.html
[12] Benoit Claise. 2004. Cisco systems NetFlow services export version 9. (2004).

[13] David E Culler, Jaswinder Pal Singh, and Anoop Gupta. 1999. *Parallel computer architecture: a hardware/software approach*. Gulf Professional Publishing.

[14] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[15] ETSI. 2017. OSM. (2017). https://osm.etsi.org/

[16] Seyed Kaveh Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C Mogul. 2014. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In *Proceedings of the USENIX Symposium on Networked System Design and Implementation (NSDI'14)*.

[17] Markus Feilner. 2006. *OpenVPN: Building and integrating virtual private networks*. Packt Publishing Ltd.

[18] Daniel Firestone. 2017. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud.. In *NSDI*. 315–328.

[19] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. Frenetic: A network programming language. In *ACM Sigplan Notices*, Vol. 46. ACM, 279–291.

[20] Linux Foundation. 2017. OpNFV. (2017).

[21] Rohan Gandhi, Hongqiang Harry Liu, Y Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. 2014. Duet: cloud scale load balancing with hardware and software. In *Proceedings of the 2014 ACM conference on SIGCOMM*. ACM, 27–38.

[22] Yang GAO and Yong-feng NIE. 2006. Traffic control management architecture based on Linux TC [J]. *Computer Engineering and Design* 20 (2006), 056.

[23] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. 2014. OpenNF: Enabling innovation in network function control. In *Proceedings of the 2014 ACM conference on SIGCOMM*. ACM, 163–174.

[24] R Guerzoni and others. 2012. Network functions virtualisation: an introduction, benefits, enablers, challenges and call for action, introductory white paper. In *SDN and OpenFlow World Congress*.

[25] W Haeffner, J Napper, M Stiemerling, D Lopez, and J Uttaro. 2014. Service function chaining use cases in mobile networks. *draft-ietf-sfc-use-case-mobility-01* (2014).

[26] J Halpern and C Pignataro. 2015. Service Function Chaining (SFC) Architecture. *draft-ietf-sfc-architecture-07 (work in progress)* (2015).

[27] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. 2015. SoftNIC: A software NIC to augment hardware. *Dept. EECS, Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2015-155* (2015).

[28] Jinho Hwang, KK Ramakrishnan, and Timothy Wood. 2015. NetVM: high performance and flexible networking using virtualization on commodity platforms. *Network and Service Management, IEEE Transactions on* 12, 1 (2015), 34–47.

[29] Kai Hwang and A Faye. 1984. Computer architecture and parallel processing. (1984).

[30] Intel. 2012. Data Plane Development Kit (DPDK). (2012). http://dpdk.org

[31] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. 2015. Silo: Predictable message latency in the cloud. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 435–448.

[32] Dilip A Joseph, Arsalan Tavakoli, and Ion Stoica. 2008. A policy-aware switching layer for data centers. In *ACM SIGCOMM Computer Communication Review*, Vol. 38. ACM, 51–62.

[33] Christoforos Kachris, Georgios Sirakoulis, and Dimitrios Soudris. 2014. Network Function Virtualization based on FPGAs: A Framework for all-Programmable network devices. *arXiv preprint arXiv:1406.0309* (2014).

[34] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks.. In *NSDI*, Vol. 12. 113–126.

[35] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. 2000. The Click modular router. *ACM Transactions on Computer Systems (TOCS)* 18, 3 (2000), 263–297.

[36] S Kumar, M Tufail, S Majee, C Captari, and S Homma. 2015. Service Function Chaining Use Cases in Data Centers. *IETF SFC WG* (2015).

[37] Bojie Li, Kun Tan, Layong Larry Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, and Peng Cheng. 2016. ClickNP: Highly flexible and High-performance Network Processing with Reconfigurable Hardware. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 1–14.

[38] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14). Seattle, WA: USENIX Association*. 459–473.

[39] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* 2014, 239 (2014), 2.

[40] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, David Walker, and others. 2013. Composing Software Defined Networks.. In *NSDI*, Vol. 13. 1–13.

[41] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. 2016. Trumpet: Timely and Precise Triggers in Data Centers. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 129–143.

[42] Jad Naous, Glen Gibb, Sara Bolouki, and Nick McKeown. 2008. NetFPGA: reusable router architecture for experimental research. In *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*. ACM, 1–7.

[43] S Natarajan, R Krishnan, A Ghanwani, D Krishnaswamy, P Willis, and A Chaudhary. 2015. An analysis of container-based platforms for NFV. *IETF Draft, Oct* (2015).

[44] A10 Networks. 2017. aFleX advanced scripting for layer 4-7 traffic management. (2017). http://www.loadbalanceworks.com/features-aFleX.asp

[45] F5 Networks. 2017. Local traffic manager. (2017). https://f5.com/products/modules/local-traffic-manager

[46] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. E2: a framework for NFV applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 121–136.

[47] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), USENIX OSDI*, Vol. 16.

[48] David A Patterson and John L Hennessy. 2013. *Computer organization and design: the hardware/software interface*. Newnes.

[49] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. 2015. Pga: Using graphs to express and automatically reconcile network policies. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 29–42.

[50] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. 2013. SIMPLE-fying middlebox policy enforcement using SDN. In *Proceedings of the ACM SIGCOMM 2013 conference (SIGCOMM'13)*. ACM.

[51] Paul Quinn and Uri Elzur. 2014. Network service header. *draft-quinnsfc-nsh-01* (2014).

[52] P Quinn and T Nadeau. 2014. Service function chaining problem statement. *draft-ietf-sfc-problem-statement-10 (work in progress)* (2014).

[53] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. 2013. Split/merge: System support for elastic execution in virtual middleboxes. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 227–240.

[54] Will Reese. 2008. Nginx: the high-performance web server and reverse proxy. *Linux Journal* 2008, 173 (2008), 2.

[55] Martin Roesch and others. 1999. Snort: Lightweight Intrusion Detection for Networks.. In *LISA*, Vol. 99. 229–238.

[56] Alex Rousskov and Valery Soloviev. 1999. A performance study of the Squid proxy on HTTP/1.0. *World Wide Web* 2, 1-2 (1999), 47–67.

[57] Erik Rubow, Rick McGeer, Jeff Mogul, and Amin Vahdat. 2010. Chimpp: A Click-based programming and simulation environment for reconfigurable networking hardware. In *Architectures for Networking and Communications Systems (ANCS), 2010 ACM/IEEE Symposium on*. IEEE, 1–10.

[58] Eder J Scheid, Cristian C Machado, Ricardo L dos Santos, Alberto E Schaeffer-Filho, and Lisandro Z Granville. 2016. Policy-based dynamic service chaining in Network Functions Virtualization. In *Computers and Communication (ISCC), 2016 IEEE Symposium on*. IEEE, 340–345.

[59] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. 2012. OpenStack: toward an open-source solution for cloud computing. *International Journal of Computer Applications* 55, 3 (2012), 38–42.

[60] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K Reiter, and Guangyu Shi. 2012. Design and implementation of a consolidated middlebox architecture. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 24–24.

[61] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. 2012. Making middleboxes someone else's problem: network processing as a cloud service. *ACM SIGCOMM Computer Communication Review* 42, 4 (2012), 13–24.

[62] Matthias Vallentin, Robin Sommer, Jason Lee, Craig Leres, Vern Paxson, and Brian Tierney. 2007. The NIDS cluster: Scalable, stateful network intrusion detection on commodity hardware. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 107–126.

[63] Wei Zhang, Jinho Hwang, Shriram Rajagopalan, KK Ramakrishnan, and Timothy Wood. 2016. Flurries: Countless Fine-Grained NFs for Flexible Per-Flow Customization. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. ACM, 3–17.

[64] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phil Lopreiato, Gregoire Todeschi, KK Ramakrishnan, and Timothy Wood. 2016. OpenNetVM: A Platform for High Performance Network Service Chains. In *Proceedings of the 2016 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization. ACM*.

[65] Yang Zhang, Bilal Anwer, Vijay Gopalakrishnan, Bo Han, Joshua Reich, Aman Shaikh, and Zhi-Li Zhang. 2017. ParaBox: Exploiting Parallelism for Virtual Network Functions in Service Chaining. In *Proceedings of the Symposium on SDN Research*. ACM, 143–149.