HydraNF: Accelerating Service Function Chains with Parallelism

Yang Zhang† *Bo Han** *Zhi-Li Zhang*† *Vijay Gopalakrishnan**
*Bilal Anwer** *Joshua Reich** *Aman Shaikh**
†*University of Minnesota* **AT&T Labs – Research*

Abstract

Network function virtualization (NFV) coupled with software defined network (SDN) creates new opportunities as well as substantial challenges such as increased Service Function Chain (SFC) latency and reduced throughput. As deployment of NFV becomes prevalent, the need to efficiently process traffic and run in optimized SFC will increase. HydraNF exploits the opportunities of parallel packet processing at both traffic level and network function (NF) level. It achieves this goal by designing innovative control- and data-plane protocols that intelligently convert a sequential chain into a hybrid one and balance traffic with affinity accordingly. By carefully analyzing the order dependency of NFs and taking NF configurations/rules into consideration, HydraNF dynamically performs fine-granularity parallelism (*e.g.,* at flow level). HydraNF is practical in that it can handle NFs *spanning multiple servers* and requires no modifications to existing NFs for *incremental deployment*. We implement a prototype of HydraNF on top of a combination of software and hardware switches and conduct extensive experiments to evaluate its correctness and performance.

1 Introduction

Network Function Virtualization (NFV), coupled with Software Defined Networking (SDN), promises to revolutionize networking by allowing network operators to dynamically modify and manage networks. Operators can create, update, remove or scale out/in network functions (NFs) on demand *on demand* [44, 49, 65, 64], construct a sequence of NFs to form a service function chain [32] and steer traffic through it to meet service requirements [45, 51, 52]. However, virtualization and "softwarization" of NFs have also posed many new challenges [33]. In particular, traffic traversing virtualized network functions (vNFs) suffer from reduced throughput and increased latency, compared to physical NFs [38, 45, 51, 52]. As the length of a service chain (*i.e.,* number of vNFs) increases, so does its latency. The flexibility offered by SDN and NFV will enable more complex network services to be deployed, which will likely lead to longer SFC chains.

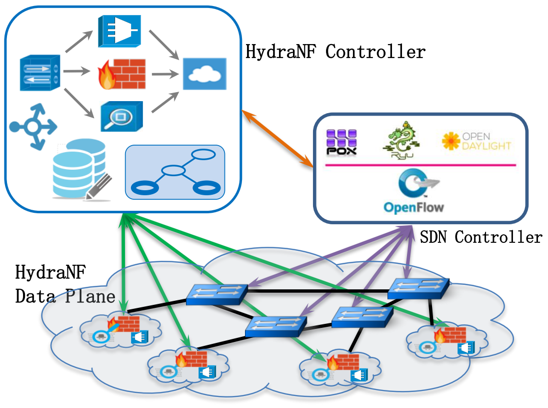

Figure 1: System Architecture of HydraNF.

Exploiting parallelism to reduce processing latency and increase the overall system throughput is a classical approach that has been widely used in networked systems. For example, most of today's web-based cloud computing applications take advantage of the "state-less" HTTP protocols fo parallel HTTP transaction processing. Data analytics frameworks such as Map-Reduce and Spark utilize task level parallelism to speed up massive compute jobs using clusters of servers. In terms of VNF, parallelism is first explored in ParaBox [67] and later in NFP [59] by exploiting order (in)dependence of certain NFs for parallel packet processing within a SFC. Both these efforts focus on parallelizing SFC packet processing on a *single* (multi-core) server. Real-world vNFs, on the other hand, will likely be operating in (small or large) edge clouds or data centers with clusers of servers [3]. How to effectively utilize multiple servers to reduce per-packet SFC processing latency and increase the overall system throughput is the main problem we will explore in this paper.

Challenges in accelerating SFC processing in a multi-server environment are multi-fold. First, as many NFs are *stateful*, which introduces *coupling* among flows. In other words, one cannot blindly perform "flow-level parallelism" by routing flows independently to multiple servers for parallel packet processing and load balancing, as we will expound on in Section 2. This state dependency also affects the order dependency of NFs beyond the "read/write" or "insert/remove" header fields considered in [67, 59], thereby further limiting what sequences of NFs can be executed in parallel. Second, availability of multiple servers not only affords more opportunities for parallelism, but also imposes additional constraints. For example, one can run multiple instances of a SFC in different servers to increase the overall system throughput; however, it will hinge on our ability to load balance the traffic among these servers. The flow coupling of stateful NFs makes this a nontrivial task; in addition, different types of traffic going through the same SFC may incur varying processing overheads. Alternatively, one can parallelize a SFC across multiple servers (e.g., with each server running multiple instances of the same NF) to increase the system throughput. However, the additional network latency for steering traffic through a sequence of servers may mitigate the gain in parallel packet processing among multiple servers. A third and closely related challenge is that in practical operating environments some NFs may be required to place in certain servers, e.g., due to performance or security isolation requirements. This further constrains the options of SFC parallelism. Last but not the least, in designing mechanisms for accelerating SFC processing in a multi-server environment, we must not only explore opportunities and constraints in parallelizing SFC processing along both the *traffic* and the *(network) function*-levels, but also "co-design" the rules needed (in hardware and software switches) for spliting and routing traffic to appropriately steer and load balance among multiple servers while minimizing the network latency incurred.

In this paper, we present HydraNF, a high performance framework, that supports parallel packet processing across vNFs running on different servers. HydraNF applies parallelism concept to both vNF function level and traffic level.

HydraNF identifies the opportunities for fine-granularity parallelism (*e.g.,* at the flow level) by carefully analyzing the configurations and operational rules of vNFs in a chain. Since it might not be possible to parallelize processing across an entire SFC, HydraNF identifies portions of the chain where parallel processing is feasible. This *hybrid chain* consists of both parallel segments (with vNFs processing packets in parallel) and sequential segments (with vNFs processing packets sequentially). Over each hybrid SFC (may contain parallel segment or just a pure sequential segment), we further partition traffic for generating rules in both hardware and software switch to balance vNF loads while taking vNF function level parallelism into consideration.

While conceptually simple, realizing parallel packet processing is by no means straightforward. First, HydraNF must guarantee the correctness of the generated chain by carefully analyzing the *order dependency* of NFs in a chain. The dependency relies on not only the semantics of NFs, but also their configurations and operational rules. Second, HydraNF needs to automatically program both virtual switches in servers and hardware switches connecting the servers by creating appropriate data-plane forwarding rules to perform parallel packet processing across multiple servers. Third, HydraNF controller need to map traffic to existing vNF deployment with balanced traffic load and try to collocate parallel segment in a hybrid chain onto the same machine. Forth, the data plane functions of HydraNF that enable parallel processing should be lightweight, avoiding adding too much processing delay and compromising the primary goal of reducing SFC latency. Finally, to enable incremental deployment, HydraNF should not require any modifications to the existing vNFs.

To address these requirements, HydraNF employs a controller that analyzes a sequential chain and converts it into a hybrid chain if it is *beneficial* to reduce SFC latency. The controller automatically programs both the virtual- and hardware switches to enable parallelism across vNFs spanning multiple physical servers, and balance traffic load across vNFs. HydraNF also employ's a custom data plane to support the dynamic hybrid chains. Based on the instructions from the controller, HydraNF data plane *mirrors* packets to parallelizable vNFs and then intelligently *merges* their outputs to ensure correctness, *i.e.,* traffic emitted from the hybrid chain must be identical to what would have been emitted by the traditional sequential SFC. The use of the custom data plane allows HydraNF to support existing vNFs without modification.

We make the following key contributions in the rest of this paper:

• We argue that the awareness of NF behavior affords additional opportunities for dynamic and fine-grained SFC parallelism. We establish a principled framework for analyzing NF order-dependency, in order to determine which portions of an SFC can be safely parallelized, and provide correctness validation for our approach. (§ 3).

• We present the system architecture of HydraNF (§ 4). We design its controller to enable both vNF level and traffic level parallelism. It effectively converts a sequential SFC into a hybrid one and balance traffic load across multiple vNF instances on multiple servers by automatically programming software and hardware switches (§ 5). In addition, we explore the design choices for the placement of key HydraNF building blocks for a high-performance data plane (§ 6).

• To demonstrate the practicality of HydraNF, we implement a prototype (§7) on top of BESS [6]. We use off-the-shelf open-source and production-grade vNFs to run experiments that examine different SFC combinations of these NFs both on a single as well as multiple servers. Our experiments (§ 8) show that HydraNF reduces latency up to 50.98% with 7% CPU overhead, and a ∼1.47x improvement in overall system throughput through traffic distribution approach in HydraNF.

2 Background and Motivation

In this section we present a few examples that highlight the challenges in designing a system for accelerating SFC with parallelism. We also articulate the comparison with the works trying to achieve parallelism in SFC.

2.1 Motivating Scenarios

it depends on the space to putting motivating examples or not in this section.

Traffic Affinity Figure **??** shows two IDS instances running on different machines for reliability. Each IDS instance keeps counting concurrent TCP sessions and blocks new sessions if a threshold is reached. Ideally, we want traffic to be distributed adhering to policy rules while balancing load across these two IDS instances. Unfortunately, traffic may not be distributed correctly, since a traffic scatter is independent from IDS, and is not aware of IDS policies, *i.e.,*, traffic affinity tenet is violated. Further, if multiple instances of the same vNF run on one machine, traffic scatter in software switch needs to keep traffic affinity as well.

Parallelism in Stateful Network In Figure **??**, a cache server is used in conjunction with an IDS. Suppose NF-level parallelism is applied, and thus requests are processed by these two NFs simultaneously and output is merged [67, 59]. However, the requests hit in the cache server are not supposed to touch IDS, which pollutes IDS internal states and may trigger unexpected behaviors.

Fine-grained NF-level Parallelism The exact semantics of an NF depends on its configurations and operational rules. For example, a firewall can be configured as a layer 3, 4, or 7 firewall, each of which examines different parts of data packets. Layer 3 firewalls check only, e.g., source and destination IP addresses in the packet header; whereas a layer 4 firewall also tracks the session state of a flow, and layer 7 firewalls inspect the payload of data packets (e.g., in order to deny HTTP requests to a given website) [1]. As shown in the top part of Figure **??**, traffic can be parallelized for an WANX → L3-FW chain, but cannot for an WANX → L7-FW chain (as the WANX compresses the packet payload which will be checked by L7-FW, but not by L3-FW). Unfortunately, existing work [67, 59] overlook this important detail.

Traffic Steering to Deployed NFs Figure **??** shows a topology where NAT, IDS, WANX are loaded on server 1, and another IDS instance, WANX instance, and Monitoring are running on server 2. Both SFCs (NAT and IDS; IDS and Monitoring) can run in parallel, since they do not have order dependency. Given a burst of incoming flows going through various SFCs, it is non-trivial to come up with a polynomial-time algorithm to steer them on an optimized path which not only takes load balancing into account, but also consider NF-level parallelism.

2.2 SFC Parallelism

Although parallelism to accelerate an SFC has been explored, *e.g.,* in ParaBox [67] and NFP [59], these solutions are severely limited in terms of real-world applicability. There are two main reasons for this.

First, these solutions only work in deployments where all vNFs of a chain are on the same physical machine. While running vNFs on the same server reduces bandwidth consumption [53], this is quite limiting in real-world networks since it is difficult to always allocate the required resources using a single server for all vNFs in a chain. Present-day vNFs are quite resource intensive when handling large volumes of traffic [3] and a dedicated server for NFV can support up to a very limited number of concurrent vNFs [5]. Dynamics of network services and elastic scalability offered by NFV also prevent single server deployments. For example, cellular networks utilize tens of elements in the virtualized Evolved Packet Core (EPC) of LTE networks [22, 56]. These vNFs are usually distributed over multiple servers, even across different data centers.

Second, ParaBox [67] and NFP [59] parallelize packet processing at the SFC-level and overlook the impact of NF configurations and operational rules. The premise of SFC parallelism is that the operations of NFs originally in tandem are independent and do not conflict with each other. In contrast, we argue that the order dependency of NFs relies on their configurations and operational rules. Hence, for correctness, we must take into account the NF semantics, as an NF may be configured to operate in different modes (*e.g.,* a firewall can be configured to drop flows by checking packet header fields at different layers). Furthermore, we demonstrate in § 2.1 that the awareness of NF semantics also affords additional opportunities for SFC parallelism via (dynamic) decompo-

Features	ParaBox	NFP	HydraNF
NF Level Parallelism	✓	✓	✓
Traffic Level Parallelism	✗	✗	✓
Intelligent Control Plane	✗	✓	✓
Inter-server Parallelism	✗	✗	✓
Granularity of Parallelism	✗	✗	✓
Incremental Deployment	✓	✗	✓

Table 1: Comparison of HydraNF with ParaBox [67] and NFP [59] on the desirable features of parallel packet processing for NFs.

sition of a sequential SFC into multiple *finer-grained* hybrid (sub-)chains, sometimes aided by operational rule transformation. The fact that ParaBox and NFP treat all flows equally limits opportunities for parallelism. Additionally, vNF semantics offer us better context for performing traffic-level parallelism, *i.e.,* "partitioning" and distributing traffic while balancing load and keeping traffic affinity.

Third, it is beneficial to use state-of-art vNF scaling techniques [57, 30]. However, it requires great amount of codebase modifications or even vNFs need to be rebuilt from the ground up to adapt new systems [43]. However, enterprise-grade vNFs are constructed with great engineering efforts, and most of them do design customized operating systems to support vNF functionalities [14, 8]. It is non-trivial for these vNFs to be adapted to such scaling techniques in near future. We focus on legacy vNFs because they are reliable and currently dominate enterprise-grade vNFs. Even if legacy vNF codebase is rewritten to adapt to these scaling techniques, HydraNF is compatible and can still gain benefits.

We compare HydraNF with ParaBox [67] and NFP [59] in Table 1 against the key features desired for SFC parallelism. In a nutshell, HydraNF performs both SFC-level and traffic-level parallelism with fine-granularity and NFs distributed over multiple servers. Furthermore, HydraNF is flexible and can support naturally both Virtual Machine (VM) and Linux Container based vNFs by leveraging an extensible software switch [6].

3 Constraints, Opportunities, and Validation in Parallelism

In this section we discuss the constraints and opportunities for performing parallel packet processing for NFs that impact the ability of HydraNF to parallelize a service chain. Before we do that, however, we describe the operational model and assumptions that we make.

3.1 Operational Model and Assumptions

We assume that (virtualized) NFs under consideration will be executed in a cluster of servers connected with SDN-capable switches (*e.g.,* OpenFlow switches). The NFs/vNFs are mainly provided by third-party vendors, using either containers or VMs (*e.g.,* for performance isolation or security requirements). We will focus on NFs that are commonly deployed for fixed networks and use them as examples for our study; however, our techniques are equally applicable to other NFs (*e.g.,* in cellular networks). These NFs include Firewall (FW), Load Balancer (LB), Network Address Translator (NAT), Traffic Shaper (TS), traffic probe, web proxy, Intrusion Detection/Prevention System (IDS/IPS), WAN optimizer (WANX), VPN gateway, *etc.* We do not assume the knowledge of the internal logic or implementation of these NFs, as many of them are proprietary solutions provided by vendors. On the other hand, network operators are responsible for devising the policies, constructing SFCs and configuring individual NFs. We assume that network operators *do* have certain knowledge regarding the "behavior" of each NF, either explicitly via NF guidelines/specifications or through inferences.

3.2 NF Semantics

At a high level, HydraNF can parallelize packet processing for NFs appearing consecutively in a sequence only if they are independent of each other. OthePolicy rule analysisrwise, we may break the correctness of the intended network and service policy. This *order-dependency* constraint is mainly determined by the operations of NFs which in turn rely on their configurations and operational rules.

NF Operations affect order dependency of NFs based on the modifications they perform on data packets. We list the common operations below and illustrate how these operations affect pair-wise NF order dependency. It is, however, straightforward to extend the analysis to more than two NFs, as we will demonstrate in §5.

• *Read/write of packets:* The read-write conflict is a well-known phenomenon in areas such as Operating Systems [36] and Databases [23]. If one NF writes to a packet field and the subsequent one reads from the same field (*e.g.,* NAT writes source IP address and IDS reads the same field), or two NFs write to the same portion of packets, HydraNF cannot parallelize packet processing by these two NFs. For other cases, it should be safe to send a packet to two NFs in parallel.

• *Reconstruction of packets*: The feasibility and efficiency of parallel packet processing relies on the fact that normally NFs only modify a small portion of data packets (*e.g.,* a few bits in the header). Thus, HydraNF should avoid parallel packet processing for NFs that substantially change the packets. For example, a WAN accelerator (WANX) may compress/decompress packets, a VPN gateway will encrypt/decrypt packets; these create an order dependency with other NFs.

• *Redirection or re-establishment of flows*: Flow redirection (*e.g.,* by an LB) introduces uncertainty about the next-hop. For example, HTTP requests balanced by an LB are directed to different servers based on run time state of the LB. Similarly, certain proxies split and re-establish TCP connections. This makes it hard to parallelize an LB or a proxy processing with the subsequent NF.

• *Termination of flows*: We use firewall as an example to illustrate the problems caused by flow termination. When an IDS exists after a firewall, parallel processing will affect *correctness* by forcing IDS to potentially alert for flows that would have been dropped the firewall. Similarly, with a firewall preceding an LB, parallelism will interfere with the load balancing algorithm as the LB will have to handle dropped flows. For other cases, such as when a firewall appears before a NAT, parallelism may increase *resource utilization* on the NAT by resulting in processing of dropped flows. While a naive solution would be to continue with sequential processing when a firewall drops a large number of flows, we propose a more effective solution later in this section.

One can categorize NFs that are commonly used by service providers based on these operations. In the interest of space, however, we refer the reader to Table 1 in ParaBox [67] and Table 2 in NFP [59]. Instead, based on their operations, we present pairs of NFs that can be parallelized in Table **??**. The first and the second NFs within a chain are in the leftmost column and the top row respectively. We mark the parallelizable chains with a ✓and sequential ones with a ✗. We mark the chains that do not have a deployment case with a **?**, while ρ indicates that the order dependency of the chain is not determinstic and relies on NF configurations. For example, if a proxy does not split a connection by re-establishing another connection with the server, we can parallelize packet processing of the proxy with other NFs. The main take-away from this table is that *a number of chains are parallelizable*.

NF Configurations and Rules. The exact behavior of an NF depends on its configurations and operational rules. For example, we can configure a firewall as a layer 3, 4, or 7 firewall each of which examines different parts of data packets. Layer 3 firewalls check only, *e.g.,* source and destination IP addresses in the packet header; whereas a layer 4 firewall also tracks the session state of a flow. Finally, layer 7 firewalls inspect the payload of data packets (*e.g.,* in order to deny HTTP requests to a given website) [1]. As shown in the top part of Figure **??**, we can parallelize packet processing for an WANX $\rightarrow$ L3-FW chain, but cannot for an WANX $\rightarrow$ L7-FW chain (as the WANX compresses the packet payload which will be checked by L7-FW, but not by L3-FW). Unfortunately, existing work [59, 67] overlook this important detail.

Inferring NF Semantics. In order to leverage the above observations, we need to determine the behavior of an NF. In general, there are two approaches: offline modeling and online monitoring. Modeling NFs has been extensively investigated in the literature. We can roughly divide the existing work into three categories: manual creation [28, 39], program analysis [63], and symbolic execution [58].

The manually created model is known to be error-prone. NFactor [63] is a tool that automatically synthesizes the model of an NF by performing code refactoring and program slicing the source code of the NF. SymNet [58] creates NF models via its proposed Symbolic Execution Friendly Language. We can extract the behavior of an NF from these models directly. The limitation of this approach is that certain network state is implicit and is difficult to infer [**?**], and the rules of an NF may be proprietary (*e.g.,* IDS signatures).

On the other hand, online monitoring can infer the modifications made by an NF through the comparison of its incoming and outgoing packets. For example, SIMPLE [55] uses a similarity-based correlation algorithm over the payload of incoming and outgoing packets to infer the behavior of NFs that merge existing sessions or create new sessions. It is known, however, that this solution increases performance overhead (*i.e.,* user-perceived latency increase) and affects correctness of inferences. We are primarily interested in how packets are modified by NFs, as the order dependency of NFs relies on this information. Hence, in this paper, we infer the NF behavior using handcrafted models [28, 39] and leave the exploration of other solutions to future work.

3.3 Opportunities for SFC Parallelization

Leveraging the knowledge of NF configurations and operational rules, we can create more opportunities for SFC parallelization by performing *fine-grained* and *dynamic* SFC decomposition (*e.g.,* at flow level).

NF operational rules facilitate fine-granularity parallelism at flow level. Both ParaBox [67] and NFP [59] treat all flows traversing an SFC equally. Parallel packet processing will be either enabled or disabled for all of them. Consider the example shown in the middle part of Figure **??**. Suppose the operational rules of a firewall require to forward flows from IP address 1.1.1.1, but drop those from 2.2.2.2. Based on these rules, we can enable parallelism for only flows from 1.1.1.1, and avoid sending flows from 2.2.2.2 to the NAT for unnecessary parallel processing. This fine-granularity parallelism also solves the flow termination issue discussed previously.

Making the operational rules of an NF HydraNF**-aware increases the opportunity of parallelism**. The reason that we cannot parallelize packet processing for the NAT $\rightarrow$ L3-FW chain is because rules of the L3-FW are configured while accounting for the fact that the source IP address of a packet is modified by the NAT. If HydraNF controller can transform the rules of the L3-FW appropriately when parallelizing it with a NAT, we can make more chains parallelizable. For the example in the bottom part of Figure **??**, the NAT translates the source IP address 1.1.1.1 to 2.2.2.2. If we change the rules of the L3-FW from dropping flows from IP address 2.2.2.2 to dropping those from 1.1.1.1, we can safely enable parallel packet processing for the NAT $\rightarrow$ L3-FW chain.

3.4 Merge Correctness

We model a given NF X as a *pure* function that takes in a packet and outputs a packet $X : pkt \rightarrow pkt$. We model each packet p (including the payload) as a vector of fields of fixed length: $p = [f_1, f_2, \ldots, f_k]$ where p_f to references the contents of field f in packet p.

3.4.1 Formal Definition of Sequential Execution

We define sequential application of a packet first through NF A, then through NF B) is $B(A(p))$ and equivalent to:

$$B(A(p)) = [B(A(p) - A(p)] + [[A(p) - p] + p] \qquad (1)$$

With respect to any given field f, we can refine Eq 1 to

$$B(A(p))_f = [B(A(p)_f - A(p)_f] + [[A(p)_f - p_f] + p_f] \qquad (2)$$

Note: when B is not a pure function of just the packet (e.g., $B : state \rightarrow pkt \rightarrow (pkt, state)$), the proofs below can be modified to hold so long as the additional condition holds that the state evolves equivalently when B sees either packet p or $A(p)$. See our online appendix for the full proofs.

3.4.2 Proofs

For any field f there are four possibilities:

1. neither: $B(p) = A(p) = p$
2. only A: $B(p) = A(p), A(p) \neq p$
3. only B: $B(p) \neq A(p), A(p) = p$
4. both A and B: $B(p) \neq A(p), A(p) \neq p$

For case (1)

$$merge_1(f, p, p', p'') := p = p' = p'' \qquad (3)$$

is rather obviously correct, as substituting for Eq. 2 yields

$$\begin{aligned} B(A(p))_f &= [p_f - p_f] + [[p_f - p_f] + p_f] && (4)\\ &= p_f && (5)\\ &= merge_1(f, p, A(p), B(p)) \quad \text{[by (3)]} && (6) \end{aligned}$$

.

For case (2)

$$merge_2(f, p, p', p'') := p' \qquad (7)$$

$$\begin{aligned} B(A(p))_f &= [A(p)_f - A(p)_f] + [[A(p)_f - p_f] + p_f] && (8)\\ &= A(p)_f && (9)\\ &= merge_2(f, p, A(p), B(p)) \quad \text{[by (7)]} && (10) \end{aligned}$$

.

For cases (3) and (4), we use the merge rule

$$merge_3(f, p, p', p'') := p'' \qquad (11)$$

when (a) the value written to field f by B only depends on fields $\{f_i\}$

$$B(p)_f = B(p')_f \quad \forall p \text{ when } p_{f_i} = p'_{f_i} \qquad (12)$$

and (b) these fields are not written by A

$$A(p)_{f_i} = p_{f_i} \quad \forall \{f_i\}, \forall p \qquad (13)$$

(If these conditions do not hold, applying merge rule 11 will not yield the correct sequential output.)

Consequently, for for field f

$$\begin{aligned} B(A(p))_f &= B(p)_f \quad \text{[by (12,13)]} && (14)\\ &= merge_3(f, p, A(p), B(p)) \quad \text{[by (11)]} && (15) \end{aligned}$$

.

3.5 Reduction to Bitwise Merge Algorithm

The merge algorithm used in our implementation is defined as a bitwise merge on the entire packet

$$merge_b(p, p', p'') := ((p' \oplus p) \wedge p'' \oplus p)) \oplus p \qquad (16)$$

Since the $\oplus$ and $\wedge$ bitwise operators applied to the i^{th} bits of the input only effect the i^{th} bit of the output (unlike the operators such as shift << and >>) and defining

$$merge_b(f, p, p', p'') := ((p'_f \oplus p_f) \wedge (p''_f \oplus p_f)) \oplus p_f \qquad (17)$$

we have

$$merge_b(p, p', p'') := \bigcup_{f \in p} merge(f, p, p', p'') \qquad (18)$$

Trivially, in case (1) the bitwise merge outputs the original packet and is thus correct. In case (2)

$$\begin{aligned} merge_b(f, p, p', p'') &= merge_b(f, p, p', p) && (19) \\ &= ((p'_f \oplus p_f) \wedge (p_f \oplus p_f)) \oplus p_f && (20) \\ &= ((p'_f \oplus p_f) \oplus p_f) && (21) \\ &= p' && (22) \\ &= merge_2(f, p, p', p'') && (23) \end{aligned}$$

Likewise, for case (3) we have

$$merge_b(f, p, p', p'') = p'' = merge_3(f, p, p', p'') \qquad (24)$$

by symmetry.

In case (4), however, $merge_b \neq merge_3$, so our current implementation cannot take advantage of the parallelization opportunity available when both A and B modify a given packet f when conditions (a) and (b) hold.

4 An Overview of HydraNF

In this section, we present the system architecture and introduce the key components of HydraNF.

As mentioned before, the design of HydraNF is guided by several operational considerations and principles. In particular, (1) HydraNF should guarantee the *correctness* and *safety* of packet processing of SFCs by preserving the semantics of sequential chaining, maintaining consistent system state and enforcing the intended network policies. (2) HydraNF should support the practical scenarios where NFs of a chain can either run as containers or VMs, within the same server or span *multiple physical machines*. (3) HydraNF should make *no modifications* to existing NFs in terms of their implementation. (4) The components added by HydraNF into the data plane should be *lightweight* without introducing noticeable latency and require *minimal knowledge* of the internal logics of NFs for high-performance packet processing and scalability.

The overall system architecture of HydraNF is schematically depicted in Figure 1. We assume that HydraNF has knowledge about the underlying network topology and switch/link capabilities, collections of NFs and their semantics (*i.e.,* configurations and operational rules), NF placement and other relevant requirements. Such information will be specified and provided by network operators as part of the input to HydraNF controller. In the following we provide an overview of the key system components and their functionality.

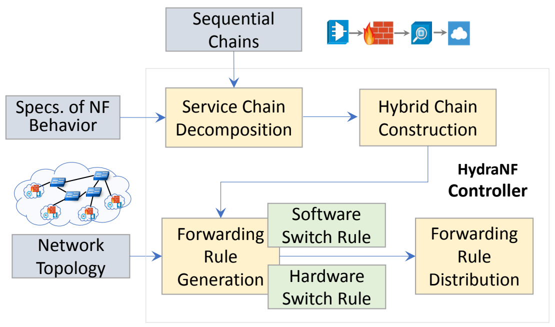

Figure 2: HydraNF Controller with four building blocks and three inputs.

HydraNF **Controller** (HRC) analyzes the order dependency of NFs in a sequential SFC and leverages the configurations and operational rules of NFs to determine whether it is beneficial to decompose the SFC into multiple *finer-grained* sub-SFCs, and whether it is possible and safe to convert each sequential sub-SFCs into a hybrid chain. One sub-SFC maps to one or more than one flow. We propose a greedy conversion algorithm for SFC parallelism (§ 5.2). We propose a dynamic-programming based algorithm (§ 5.3) to map incoming traffic to existing vNF deployment, and in the meantime, balance traffic distribution among multiple machines. The algorithm can correctly migrate loads while preserving traffic affinity after server/vNF scaling. Based on the output of these two algorithms, the run-time system of HRC automatically generates appropriate processing rules and pushes them to software switches that connect vNFs on a server. It also interacts with an SDN controller to push forwarding rules in hardware switches that connect multiple servers.

HydraNF **Data Plane** consists of two key modules, *mirror* and *merge*, which are employed as packets move through the service chain. For a single-machine scenario, both mirror and merge modules run on the same physical server. Regarding the multi-machine scenario, there are several design choices as far as placement of these module goes. For example, we can place them on different servers, or implement either one or both of them on programmable hardware switches, by considering the performance requirements, bandwidth utilization, implementation challenges, *etc.* (as we will discuss in §6).

5 HydraNF Controller

In this section, we describe the main functions of HRC and describe the hybrid chain construction algorithm. Figure 2 shows the design of HRC which consists of four building blocks.

5.1 Service Chain Decomposition

Given a sequential chain, HRC analyzes it to decide whether and how to decompose it into multiple finer-grained sub-chains. To do this, it leverages NF-behavior specifications (*i.e.,* configurations and operational rules). In particular, for each NF, it parses the operational rules to extract packet/flow classification information based on L2-L4 headers. It then steers a flow through a sub-chain based on the classification results. We employ a similar technique as used in Header Space Analysis [42, 41] to keep track of the possible header transformation made by NFs. This allows us to expand the sequential chain into a graph of (labeled) sub-chains. A sub-chain can safely bypasses NFs in the original chain that perform no operations on the packets (*e.g.,* the rest of NFs after a FW for the dropped flows), leading to a shorter sub-chain. In summary, given a sequential chain, the Service Chain Decomposition module produces a set of finer-grained sub-SFCs at flow level.

5.2 Construction of Hybrid Chains

Given each (sub-)SFC produced by the Service Chain Decomposition module, HRC applies a greedy algorithm to convert it into a hybrid chain by considering NF order-dependency while also leveraging NF-behavior awareness to ensure correctness and safety. We show the baseline algorithm in Algorithm 1. The inputs of the algorithm are a sequential chain *SC* and the operations of its NFs *NF_Ops* extracted from the specifications of NF behavior.

Algorithm 1 Construction of Hybrid Chain

Variable Definition: (a) *SC*: sequential chain (input); (b) *NF_Ops*: operations of NF (input); (c) *NF_Seg*: current NF segment; (d) *Agg_Ops*: aggregated operations of current segment; (e) *HC*: hybrid chain (output)

```
 1: procedure CONSTRUCT_HYBRID_CHAIN
 2:     initiate NF_Seg, Agg_Ops, HC
 3:     while $NF_i$ in SC do
 4:         NF_Ops ← Fetch_Ops($NF_i$)
 5:         if Independent(NF_Ops, Agg_Ops) then
 6:             NF_Seg.push($NF_i$)
 7:             Agg_Ops.push(NF_Ops)
 8:         else
 9:             HC.push(NF_Seg)
10:             NF_Seg.clear()
11:             Agg_Ops.clear()
12:             NF_Seg.push($NF_i$)
13:             Agg_Ops.push(NF_Ops)
14:     HC.push(NF_Seg)
```

The main idea of Algorithm 1 is to parallelize two consecutive NFs based on the order-dependency constraints, aggregate their operations, and take their combination as a bigger one for further processing. If NF_i is parallelizable with the current NF segment *NF_Seg* (*i.e.,* having independent ordering which is derived from their operations), we push it into *NF_Seg* and aggregate its operations into *Agg_Ops* (lines 6-7). Otherwise, we push *NF_Seg* into the output hybrid chain *HC* as a completed segment, clear *NF_Seg* and *Agg_Ops*, and then push NF_i into *NF_Seg* and its operations into *Agg_Ops* for the order-dependency check with the next NF (lines 9-13).

We use an example to illustrate the execution of Algorithm 1. Suppose we have a chain with six NFs: A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D $\rightarrow$ E $\rightarrow$ F. Let's assume this chain consists of two parallel segments: {A, B} and {D, E, F}. Given this, *Agg_Ops* is empty when the algorithm begins. As the *Independent*() function returns TRUE, we push A into *NF_Seg* and its operations into *Agg_Ops*. Since A and B are parallelizable, the algorithm pushes B and its operations into *NF_Seg* and *Agg_Ops* respectively. As a result, *NF_Seg* becomes {A, B}. Since C is not parallelizable with A and B, we push the current segment {A, B} into the output *HC* before handling C and its operations. The next NF is D which is not parallelizable with C. The algorithm thus pushes the current segment with a single component C into *HC*, turning it into {{A, B}, C}. Continuing in this fashion with D, E and F, the algorithm pushes the last parallel segment {D, E, F} into *HC*. Thus, the output of our algorithm in this case turns out to be {{A, B}, C, {D, E, F}}.

5.3 Traffic Distribution over Multiple Servers

5.3.1 Mapping Hybrid Chains to Servers

After analyzing vNF configurations and operational rules affecting forwarding, a superclass is generated as a basic unit for traffic affinity in load balance. For example, IDS may maintain host-based state (*e.g.,*, number of concurrent sessions of a host should not be larger than 200), while a NAT maintains flow-based state. In this case, we choose *host* as a basic unit because it contains multiple flows. [1]

To solve traffic distribution problem, our algorithm is based on dynamic programming. The algorithm is divided into four steps: 1) extracting all common vNF be-

[1]It is a very rare case in vNF that a specific flow forwarding is determined by a variable updated by all flows/packets. Such a variable typically serves as statistic purpose. Otherwise, it requires all vNF instances to work as a whole for scaling [57, 30], which requires great vNF code changes. Such scaling solutions are compatible with HydraNF, because multiple vNF instances are taken as one instance for NF-level parallelism.

tween target chain and vNF instances running on each machine; 2) calculating all possible combinations to constitute the target chain; 3) filter results with minimal machines numbers; 4) filter results with parallel segments on the same machines. Figure **??** shows an example of how traffic distribution algorithm works. For the interest of space, we put the implementation of the algorithm in our website [4].

Here we dive into the details of step 1 and 2, while step 3 and 4 are straight-forward.

Step 1: extracting all common vNF between target chain and vNF instances running on each machine. SFC has order, but vNF instances running on each machine do not have order. Thus, we can only use vNF instances which is consecutive in target SFC. The gist of this algorithm is based on the following formula.

Step 2: calculating all possible combinations to constitute the target chain. The gist of this algorithm is based on the following formula.

```
if SFC[i..j] can run in a single machine:
SFC[i][j] = True
else: SFC[i][j] = True, if SFC[i][k-1] and SFC[k][j] are
true for any k between i to j.
```

The output of traffic distribution algorithm is the chain instances which can be run in parallel for flows associated with the SFC in existing topology. How load balancing can be achieved in Openflow Switch and its optimization have been widely exploited in existing works [61, 50, 54]. Our implementation borrows the idea from paper [61], the partition idea of which is similar to consistent hashing [10], to perform traffic load balance in a proactive way. In our current implementation, the system only support basic unit as host. We leave other basic units like cluster affinity, application affinity as future work.

5.3.2 Heavy Load and NF Scaling

With traffic load goes up, the initial traffic distribution may not be able to handle. There are two options. One is to steer traffic in a sub-optimal vNF instance path, while the other one is to launch more vNF instances on servers.

In term of the first solution, the topology does not change. The intermediate results generated by traffic distribution algorithm can be utilized to pursue a sub-optimal solutions quickly. For example, step 4 can be released to parallelize traffic across multiple machines. Even step 3 can be released that traffic goes through more physical machines to achieve SFC. Similar to paper [61], the system monitors rule counter in Openflow switch as indicator for unbalanced load, and transitions traffic with connection affinity. Note that the way of partitioning incoming traffic across the new and old instances avoids state migration [30, 57] in vNF instances.

The initial placement decisions are informed by network operator, which may change over time. When NF instance is overloaded, new instance or servers loaded with NF instances will be added into the cluster. Every time the topology is changed (e.g., a new NF instance is added), traffic distribution algorithm needs to be run again. HydraNF needs to adapt to new instance when scattering traffic.

5.4 Generation of Forwarding Rules

To accommodate the multi-server scenario, HRC automatically generates the forwarding rules for both software and hardware switches and enforces them appropriately. We should note that achieving this is more complicated and challenging than existing work on traffic steering for SFC, such as SIMPLE [55] and StEERING [68], which mainly consider the configurations of hardware switches only.

As shown in Figure 2, the inputs to the Forwarding Rule Generation module are the hybrid chain and the network topology which specifies the placement of NFs and how the servers that host these NFs are connected via hardware switches. Note that we are assuming here that that NF placement is provided as an input to HRC. However, if HRC can control NF placement, we can potentially improve the SFC latency by integrating the state-of-the-art proposals (*e.g.,* Slick [21] and E2 [53]) into HydraNF. We leave this as part of future work though.

The forwarding rule generation module, pushes the processing and forwarding rules to appropriate hardware and software switches. The module configures hardware switches through an SDN controller (*e.g.,* OpenDaylight [17] and Ryu [20] for OpenFlow switches). HydraNF does not rely on a specific SDN technology, such as OpenFlow, and can operate on top of any programmable switches with their corresponding controllers. For software switches, we use a local daemon on each physical server that communicates with HRC, and pushes the received forwarding rules to the switches. This daemon also reports run-time state of the data plane and system load information based on HRC's queries.

We deploy server-level forwarding rules for hardware switches, as they do not need to know the fine-grained traffic forwarding details. The reason is that the NF-level traffic steering is performed by software switches running on the servers. Thus, the rules installed by HRC steer the traffic among servers loaded with NFs. The creation of hardware-switch rules can benefit from technologies such as SwitchTunnels, which is used in SIMPLE [55], to compact the forwarding table. We leave this optimization to future work. We will discuss in § 6

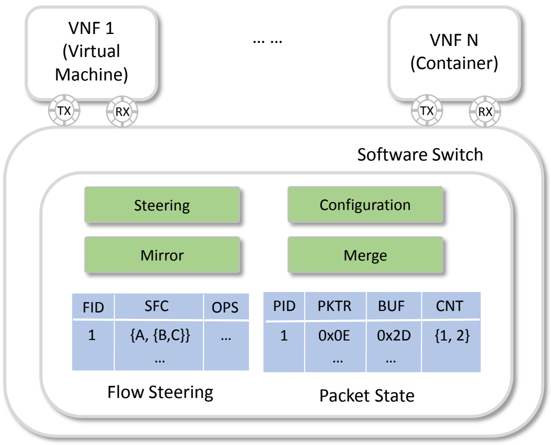

Figure 3: Building blocks and data structures of HydraNF data plane.

how to improve the efficiency of bandwidth utilization by configuring the hardware switches to support the mirror function.

HRC also computes and installs forwarding rules in software switches running on servers to enable parallel packet processing for NFs. Besides the traffic steering rules, HRC informs the data-plane modules the operations of NFs hosted on each physical server for two purposes. First, certain NFs add/delete bytes in data packets (*e.g.,* VLAN ID). This information is needed by the data plane to merge the outputs of parallelizable NFs correctly. Second, for NFs that do not modify packets (*e.g.,* traffic shaper), this information can help the merge function to optimize its performance. We will present the details in the next section.

6 HydraNF Data Plane

In this section, we present the data-plane design of HydraNF, focusing in particular on its scatter module, mirror and merge modules and their placement.

6.1 Mirror and Merge Modules

We show the main building blocks and data structures of the HydraNF data plane in Figure 3. The data-plane execution engine enforces SFC parallelism based on the forwarding and processing rules installed by HRC. We first present design of the data plane within a single server as an extension of software switches. We will present the support for multiple server case in § 6.2. Before presenting the details of the mirror and merge modules, we first describe structure of two tables, *Flow Steering* table and *Packet State* table, that these modules operate on.

The Flow Steering table contains information for packet processing of SFC segments consisting of NFs residing within a given server. Each entry represents an SFC segment, *e.g.,* {A,{B,C}}, along with the corresponding NF operations denoted as OPS (see § 5.2), and FID which denotes the flow (in terms of a layer 2–4 header match rule) to which the SFC segment applies. HRC installs these entries in the software switch. The mirroring module uses the Steering Flow entries to steer packets among NFs, duplicating them if needed. For an example SFC segment {A,{B,C}, mirroring module would duplicate packets processed by A, sending them to both B and C for parallel processing.

The Packet State table is primarily used by the merge module and contains four fields: 1) PID (packet ID), 2) PKTR (reference pointer to the memory address of the original packet), 3) BUF (packet buffer for saving the intermediate results), and 4) CNT (counter array for parallel SFC segments). The unique PID keeps track of duplicate packets that are processed by parallelizable NFs. The CNT records the number of NFs in each segment of a local hybrid SFC. For instance, CNT for {A, {B, C}} is {1, 2}. The count decrements by one after a packet goes through an NF in the segment. HydraNF performs the merge operation when the count reaches zero.

In the merge operation, we treat a data packet as a sequence of bits, namely a $\{0|1\}^*$ string. If an NF in a parallel adds L extra bits into packets, we insert a string of L zeros at the corresponding location in the outputs from other NFs before the merge. In a similar vein, if an NF removes L bits from packets, we delete the same bits from the outputs of other NFs. Let's assume P_O is the original packet and there are two NFs A and B in the chain. Assume further that P_A and P_B are their respective outputs. In order to obtain the modified bits by an NF, we *xor* its output with the original packet. We obtain the modified bits of multiple NFs by combining (*or*) the modifications performed by each of them. For example, the output of the merge function {A, B} parallel segment would be $[(P_O \oplus P_A)|(P_O \oplus P_B)] \oplus P_O$. Note that correctness is guaranteed as parallelizable NFs do not modify the same portion of a packet. Finally, we recalculate the checksum before steering/mirroring the packet to the next NF(s) – either residing within the same server or different server.

Load Balanced Merging

6.2 Placement of Mirror and Merge

When parallelizing packet processing for NFs spanning multiple machines, a natural question is where to place the mirror and merge functions. For SFC parallelism within a server, the mirror and merge functions are placed with software switches, as in ParaBox [67]. However, when considering an SFC spanning multiple servers, if we naively place mirror and merge modules on one of the servers, we may not only waste bandwidth,

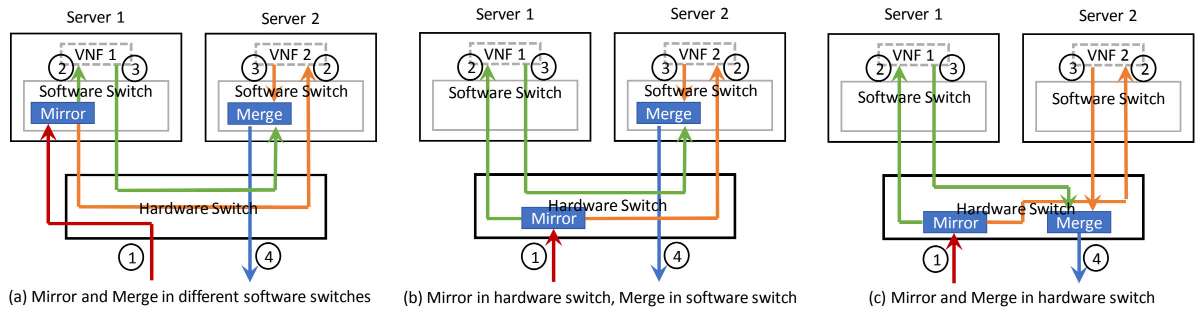
Server 1
Server 2
VNF 1
VNF 2
Software Switch
Mirror
Merge
Hardware Switch
1
2
3
4
(a) Mirror and Merge in different software switches
(b) Mirror in hardware switch, Merge in software switch
(c) Mirror and Merge in hardware switch

Figure 4: Various placements of the mirror and merge modules for the multi-server scenario. In the first step, the mirror module receives the packets. It then duplicates them to two NFs in the second step. In the third step, the NFs send back the packets to the merge module which generates the final output.

but also potentially increase the SFC latency.

For example, consider an SFC: NAT → FW → IPS → WANX, and suppose we can parallelize the packet processing for NAT, FW and IPS, but not the WANX. Further, assume that NAT and FW are placed on server 1, while IPS and WANX are on server 2. If we place the mirror and merge functions on server 1, the mirror function will have to duplicate packets to IPS on server 2. After IPS examines the packets, it will have to send them back to server 1 for the merge, and then back to server 2 again for the last-hop WANX processing. As a result, placing the mirror and merge functions on server 1 ends up increasing latency instead of reducing it.

Hence, when parallelizing SFC processing across multiple servers (with NF placement constraints), the location of the mirror and merge functions is also crucial. Using the two parallel NFs placed on two servers as an example, Figure 4 illustrates several design choices for the placement. (a) *mirror and merge in software switches (on servers) only*: This placement strategy can, to certain extent, avoid sending packets back-and-forth between the two servers, but there will still be two outgoing flows for server 1 and two incoming flows for server 2 (in contrast, processing two NFs sequentially generates only one incoming and outgoing flow for each server). (b) *mirror on hardware switches and merge on software switches*: This placement can reduce the number of outgoing flows for server 1 from two to one; it still cannot improve the situations for server 2. This design choice is what is currently implemented in HydraNF. (c) *both mirror and merge on hardware switches*: This is the ideal case which achieves the same bandwidth utilization as the sequential chain for this example.

Although it is feasible to place the mirror function on hardware switches (given its simple functionality), it is more challenging to design and implement the merge function even on programmable hardware switches. The reason is that the merge function requires relatively complex operations (*e.g., xor*) and needs extra memory to store the intermediate results. We will discuss the possibility of implementing the merge function in P4 [24] switches in § 10.

Note that if we want to realize SFC parallelism for multiple NFs within a server, we will still need to place the mirror and merge functions in the software switch on the server. In the above discussion, we focus on multiple servers connected to the same hardware switch. We can easily extend HydraNF to the multi-machine multi-switch scenario.

7 Implementation

We have implemented a proof-of-concept for HydraNF. To handle the single-server scenario, the implementations of both the mirror and merge modules are in a software switch. We offload the mirror module to the programmable switches for improving the efficiency of the multi-server scenario.

Our current implementation uses BESS (Berkeley Extensible Software Switch) [6], a modular framework integrated natively with Intel DPDK [11]. We can also use other software switches, such as Open vSwitch [12], mSwitch [37], Vector Packet Processing [9], *etc.*, with minor modifications to our implementation. We choose BESS mainly due to its high performance and flexibility. BESS allows friendly extensions from developers by defining their own customized logic. We have implemented HydraNF's data plane as several extensible BESS modules with around 1100 lines of C code (*i.e.,* 200 lines for the mirror function and 800 for the merge function). For other functions such as input and output ports and VLAN operations, we reuse BESS's native modules with extensions (200 lines overall) for supporting metadata transfer among the BESS modules.

To optimize the bandwidth utilization for the multi-server scenario, we have implemented the mirror module inside OpenFlow switches. Due to the more complex operations in the merge module, it is challenging to implement it using OpenFlow switches. We are now exploring the feasibility of offloading the merge module to P4 [24] switches. As recommended by Henke *et al.* [35], the packet ID is the one-at-a-time hash value of selected high entropy bytes of a packet, which depend on the protocols (*e.g.*, the IP ID field, TCP sequence and acknowledgment numbers, *etc.*).

Regarding the control plane, we have implemented HRC in Python with around 700 lines of code using the Flask framework [13]. We store the table information as discussed in Section 5 using pickleDB [19] which is a light-weight key-value store. HRC communicates with the Ryu [20] OpenFlow controller through its RESTful APIs. HRC includes a configuration module for hardware switches that installs and removes forwarding rules via Ryu. We have implemented the local daemon (150 lines of Python code) to exchange control information with HRC and report system usage, also using the Flask framework [13].

8 Evaluation

In this section, we evaluate HydraNF through a prototype implementation. In order to demonstrate the key features of HydraNF, we conduct extensive experiments for the following scenarios.

• In benchmarking experiments, HydraNF performance is better compared to existing solutions (Figure 6).

• In realistic chains, HydraNF reduces SFC latency in various setups (Figure 7)

• HydraNF improves packet processing performance in multiple-server scenarios. (Figure 8, Figure 9, Figure 10).

• The overhead introduced by HydraNF is manageable. (Figure 11, Figure 12)

• HydraNF achieves fine-grained parallelism by analyzing NF configurations and operational rules. (Figure 13, Figure **??**)

Prototype. HydraNF prototype uses an Openflow-enabled Switch with 48 x 10Gbps ports. We connect six servers to the switch, each with two 10Gbps links. Each server uses Intel(R) Xeon(R) CPU E5-2620 with 6 cores, for a total of 36 cores. On each server, one core is dedicated to HydraNF data plane. The HydraNF controller runs on a standalone server that connects to each server and to the management port of the switch on a separate 1Gbps control network.

vNF used in Experiments. The experimental setup is shown in Figure 5. Each NF is running in either a Docker container or a KVM-based VM. We dedicate a CPU core to each container or VM. We use seven types of vNFs: Layer 2 forwarder (L2FWD), NAT, FW, IDS, Monitor, Load Balancer, and VPN gateway. L2 Forwarder is used as vNF baseline. For NAT and FW running in VM, we use product-level vNFs with real operational rules from a carrier network. We also use open-source iptables running in containers as NAT and FW. We use BRO [7] as IDS, Nload [16] as Monitor, Linux Network Load Balancing [15] as Load Balancer, and OpenVPN [18] as

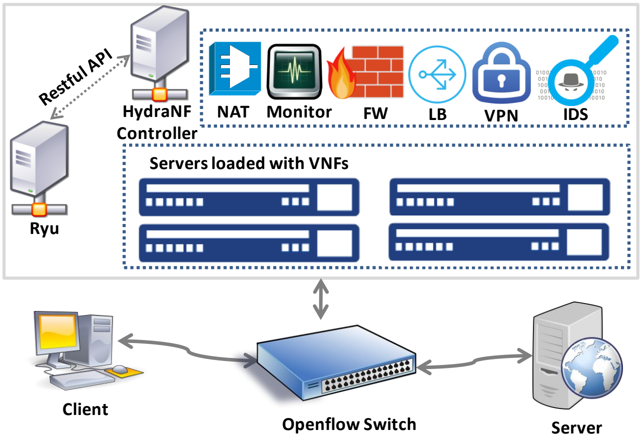

Figure 5: Experiment Setup

VPN gateway in the experiments. Moreover, we create customized L2 forwarder, NAT, and FW to call BESS zero-copy API and OpenNetVM zero-copy API respectively.

8.1 HydraNF Data Plane Performance

We evaluate HydraNF data plane by first performing benchmarking results and comparing with OpenNetVM. Then, the system over several realistic service chains are evaluated. After that, in multiple-machine scenarios, we evaluate the performance benefits by both NF-level parallelism and traffic-level parallelism, and placement of HydraNF core modules. Finally, data plane overhead in term of CPU cycles is measured. The results are presented for traffic workflow of all minimal-sized 64B Ethernet packets.

As a benchmarking experiment, we compare the SFC latency for both sequential and hybrid chains in HydraNF with OpenNetVM [66] and show the results in Figure 6. We use the L2FWD included in OpenNetVM as the NF and run it in a container for HydraNF. We increase the SFC length from 1 to 6. Figure 6 shows the latency comparison between OpenNetVM and HydraNF.

In theory, the more NFs are processed in parallel, the more processing latency can be reduced. We observe from Figure 7 that the processing latency in parallel processing is significantly reduced compared to sequential processing. Moreover, the latency reduction rises from 13.04% to 50.98% with the length of a chain. However, the latency benefits do not reach a theoretical value for a chain with 6 vNFs to be parallelized simultaneously due to overhead of the mirror and merge modules, which we will explore later.

Table 2 presents sequential and hybrid chains generated by Algorithm 1. Latency reduction is demonstrated in Figure 7. Since HydraNF provides unified virtual interfaces to both Docker containers and KVM-based VMs, mixed-technique vNFs can be integrated into a chain and processed by HydraNF. As we can from this figure, HydraNF can reduce the SFC latency by up to 31.7%.

Supporting vNFs across multiple machines is challenging. Not only software switch, but also physical switch needs to be programed to enable vNF parallelism. We also takes efforts to optimize vNF parallelism under multiple machines scenario by putting core modules in different places. In Figure 8, HydraNF Parallel Processing (MM) means that mirror and merge functions are put together into the same physical switch; HydraNF Parallel Processing (M/M) means that mirror and merge functions can be decoupled into different physical machines. HydraNF Parallel Processing (MH/M) means that mirror functions are implemented into physical switch. Figure 8 shows latency gains achieved by vNF parallelism in various SFCs. We observe that HydraNF controller can correctly distribute forwarding rules into hardware and software to enable parallelism, and vNF parallelism among multiple machines can still achieve latency gains. Among three system implementation choices for vNF parallelism, putting mirror functions into physical switch achieves the best results in most cases.

However, it is not always the case that putting mirror functions into physical switch can achieve the best performance results. For example, in the chain that VPN and monitor are in the same machine, while FW and LB are on the other machine, Monitor and FW can be processed in parallel. The reason why putting mirror function into physical switch is not as good as putting it into the physical switch because the traffic processed by VPN needs to be mirrored into Monitor and FW. Thus, the traffic needs to be detoured back to physical switch for mirror. In order to further investigate the performance difference among these three scenarios, we use iperf to generate traffic and last for 60 seconds. From the bandwidth utilization on the switch, there are around 40GB traffic exchanged in putting mirror functions in physical machines, but for putting mirror functions in physical switch, the traffic exchanged through physical switch is just 35GB. In sequential chain, we just observe around 30GB traffic exchanged. We expect further reduction in traffic caused by NF parallelism by putting merge functions into hardware switch as well.

To understand the overhead introduced by HydraNF, especially the mirror and merge modules, we measure the CPU cycles to process each packets. Figure 11 indicates the CPU cycles per packet increase with the SFC length for both sequential and hybrid chains. HydraNF introduces at most 7% more CPU cycles per packet than the sequential chains.

Furthermore, in order to verify the correctness of hybrid-chain construction, we always do the experiments to send traffic through sequential chains and reply the same traffic through the corresponding hybrid chains. We then compare the packets at the receiver side and network state maintained in vNFs using log information, in order to confirm that the hybrid chains generate the same output as the sequential ones.

8.2 Fine-Grained Parallelism

Fine-grained parallelism not only saves vNF resource utilization, but also guarantees the correctness of vNF parallelism under stateful network functions.

8.2.1 Resource Utlization

Without fine-grained parallelism, resource utlization of some vNFs suffers. We craft a SFC consisting with FW and NAT. There are N FW instances before NAT, and on average 1/N flows will be allowed by FW instances. Figure 13 indicates packets receiving rate at receiver. With the number of FW instance increases, the receiving rate decreases dramatically without fine-grained parallelism.

8.2.2 Stateful Network Function

HydraNF is transparent to stateful network function, given the assumption that fine-grained parallelism is performed by HydraNF. Let us take a look at a chain in which traffic goes through stateless FW and then a stateful FW. We assume that thestateful FW limits the number of active TCP flows to be no more than 100, while stateless FW randomly terminates 1/2 TCP flows it recieves. Experimental results in Figure **??** show that fine-grained parallelism produces the same execution results with sequential execution, but not coarse-grained parallelism. Moreover, the traffic load going through stateful FW in coarse-grained parallelism is larger than fine-grained parallelism.

9 Related Work

Parallelism in Computer Networks. Parallel processing has also been widely investigated in other areas of computer networks. P4 [24] is a domain-specific, protocol-independent language for programming packet forwarding dataplanes with support for parallel table lookups. ClickNP [46] utilizes parallelism in FPGA to optimize packet processing. MapReduce [27] is a programming model (plus an associated implementation) for large data-set processing, which parallelizes jobs on a cluster of servers. Graphene [31] is a cluster scheduler for improving job completion time in a directed acyclic graph with many parallel chains. WebProphet [47] and

Table 2: Service Function Chains Used in the Experiments

Index	Deployed Chain in One Machine	Hybrid Chain in One Machine	Deployed Chain in Two Machines	Hybrid Chain in Two Machines
1	{IDS, FW, NAT}_container	{{IDS, FW, NAT}}	{IDS ∥ FW, NAT}_container	{{IDS, ∥ FW, NAT}}
2	{IDS, FW, NAT}_vm	{{IDS, FW, NAT}}	{IDS ∥ FW, NAT}_vm	{{IDS, ∥ FW, NAT}}
3	{VPN, Monitor}_vm, {FW, LB}_container	{{VPN},{Monitor, FW}, {LB}}	{VPN, Monitor}_vm ∥ {FW, LB}_container	{{VPN}, {Monitor ∥ FW}, {LB}}
4	{FW, NAT, IDS, LB}_vm	{{FW, NAT, IDS}, {LB}}	{FW, NAT ∥ IDS, LB}_vm	{{FW, NAT ∥ IDS}, {LB}}

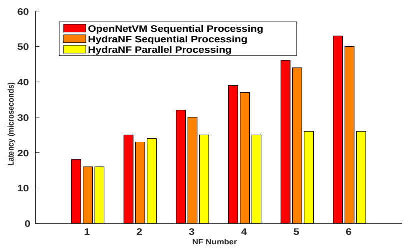

Figure 6: Baseline Performance Results

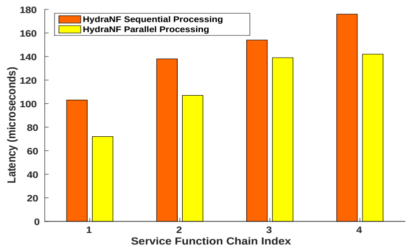

Figure 7: Performance of Different Service Function Chains with Different Complexity

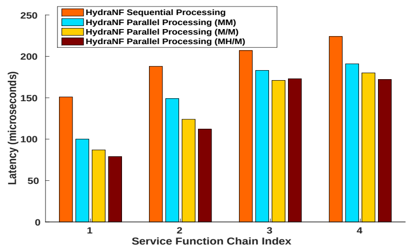

Figure 8: Performance of Different Service Function Chains across Multiple Physical Servers

WProf [62] expedite web page load time by parallelizing object downloading based on dependency graph analysis.

Service Function Chaining. Service function chaining has been extensively studied in the literature. The Delegation-Oriented Architecture [60] facilitates incremental middlebox deployment and chaining by leveraging global identifiers in packets and endpoint-based delegation. PLayer [40] proposes a policy-aware switching layer to explicitly steer traffic through middleboxes. SIMPLE [55] is a policy enforcement platform for efficient middlebox specific traffic steering. STEERING [68] provides a flexible way to route traffic for inline services based on SDN schema. FlowTags [29] offers flow tracking capability by using tags associated with appropriate middlebox context to ensure consistent policy enforcement. PGA [34] proposes a network policy expression and leverages graph structure to resolve service chaining policy conflicts. Slick [21] implements an integrated platform which handles both middlebox placement and traffic steering for efficient use of network resources. The above-mentioned papers about traffic steering and policy enforcement for SFC are orthogonal to HydraNF whose goal is to improve SFC latency.

Network Functions Virtualization. There is a plethora of work on improving the performance and manageability of VNFs. FreeFlow [?] provides a systems-level and state-centric abstraction, called Split/Merge, for elastic execution of vNFs. NetVM [38] allows zero-copy packet delivery across a chain of VMs within a trust boundary on the same physical server. ClickOS [48] proposes a high-performance virtualized software platform to reduce VM's instantiation time. OpenNF [?] is a framework that offers coordinated control of network function state. E2 [53] proposes an NFV framework which makes developers focus on core application logic by automating and consolidating common management tasks. OpenBox [25] decouples control plane from data plane of vNFs and allows reuse of software modules across network functions. Differently from the above work, HydraNF leverages vNF-level parallel packet processing to reduce service chain latency.

10 Discussion

In this section, we discuss how to extend HydraNF and highlight several open issues as future directions.

Implementing Merge Function in Programmable Switches. We have currently implemented the mirror function in hardware switches, but the merge function in software switches. As discussed in § 6.2, placing both mirror and merge functions in hardware switches is the ideal solution which introduces almost no overhead in terms of bandwidth utilization. Even for OpenFlow switches, it is challenging to implement the merge functions, as the required *xor* and *or* operations are not yet supported by the OpenFlow specification version 1.3 which is used widely in switches available today. However, it is possible to implement the merge function with P4 [24] switches. P4 exposes a standard set of primitive actions, including *bit_or* and *bit_xor* [2]. Given this support, we are currently implementing the merge function of HydraNF on P4 switches.

vNF Scaling In/Out. To support vNF scaling, HRC needs to integrate with the vNF application controller/orchestrator. One possible approach is to inform HRC about the creation/deletion of vNF instances and their locations. Based on the notification, the mirror function would be aware of the new instances and start mirroring traffic to them, or stop duplicating packets for the removed vNF instances. This will require adding the support for vNF instance tracking inside the Flow Steer-

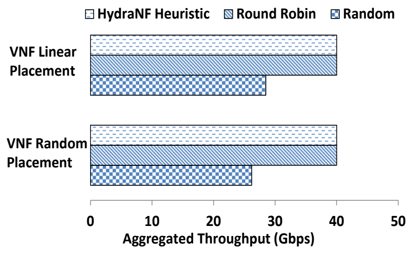

Figure 9: Throughput with different placement solutions

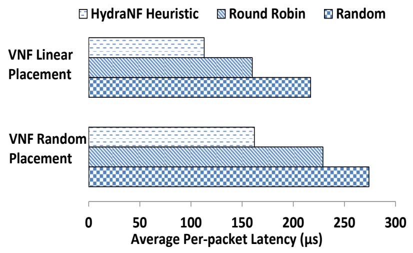

Figure 10: Latency with different placement solutions

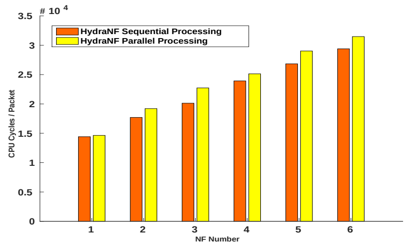

Figure 11: CPU Overhead in Sequential Processing and Parallel Processing

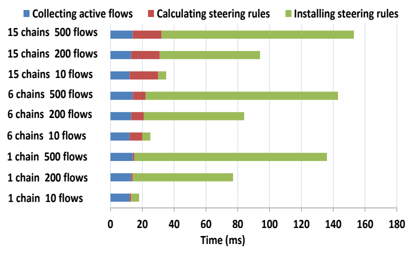

Figure 12: Controller Overhead

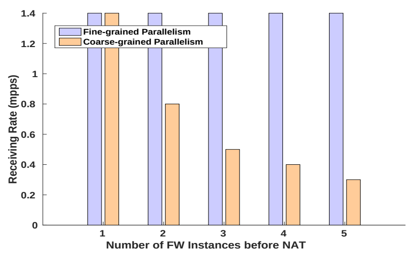

Figure 13: Resource Utilization between Fine-grained Parallelism and Coarse-grained Parallelism

ing table as well as modifications to the merge function. Furthermore, we plan to support asymmetric vNF scaling, *i.e.,* HydraNF can dynamically scale in/out any hop of a chain, instead of scaling the whole chain.

vNF Decomposition. Recent proposals virtualize and decompose NF at different granularity. For example, EdgePlex [26] assigns a single VM for the control plane and a dedicated VM for the data plane of each customer provisioned on a provider edge router. HydraNF should be able to handle this type of decoupling of control and data planes due its ability to work at flow-level. OpenBox [25] performs fine-grained decomposition at the software module level of NFs. In this case, the components of a service chain will be NF modules, instead of NFs. Despite this, it should still be feasible for HydraNF to parallelize packet processing for these NF modules in a chain.

The Gain of Parallelization depends on the dominating NFs in a chain in terms of processing latency. For example, if a traffic shaper adds 100 μs extra latency to a service chain with two other NFs having 10 μs processing latency each, employing parallelize packet processing will reduce the latency from $\sim$ 120 μs to $\sim$ 100 μs (*i.e.,* merely a 16.7% reduction). However, if the traffic shaper adds only 20 μs, HydraNF can reduce the latency by $\sim$ 50% (from 40 μs to 20 μs). In cases where parallelization gain is not high enough, it would be prudent to revert back to the sequential chain.

Other Limitations. We have not considered other types of vNFs, *e.g.,* mobility vNFs which include virtual Packet Data Network Gateway, Serving Gateway, Mobility Management Entity *etc.*, for parallel data processing. We have also not investigated the parallelization for layer-3 virtual routers. These NFs either create various tunnels or determine the next hop, creating order dependency with other NFs. However, HydraNF we can still chain them with other NFs due to its hybrid nature.

11 Conclusions

In this paper, we design, implement and evaluate HydraNF, an SFC parallelization framework to reduce the service chaining latency. In contrast to existing approaches, HydraNF takes into account the configurations and operational rules of NFs (*i.e.,* the NF behavior) to create more parallelization opportunities, while performing fine-grained parallelism for packet processing. HydraNF supports not only NFs within a single servers, but also those spanning multiple machines, by automatically generating processing and forwarding rules for software and hardware switches. The data plane of HydraNF runs as an extension of the BESS virtual switch and thus works naturally with vNFs implemented in both VMs and containers. Our performance evaluation through extensive experimentation demonstrates that HydraNF can reduce SFC latency with manageable overhead for various realistic scenarios.

References

[1] Layer 3 and 7 Firewall Processing Order. https://documentation.meraki.com/MR/Firewall_and_Traffic_Shaping/Layer_3_and_7_Firewall_Processing_Order, 2015.

[2] The P4 Language Specification, version 1.0.3. https://p4lang.github.io/p4-spec/p4-14/v1.0.3/tex/p4.pdf, 2016.

[3] Virtual Network Functions (VNFs). https://www.business.att.com/content/productbrochures/network-functions-on-demand-vnf-brief.pdf, 2016.

[4] Algorithm to Extract All Possible Chains. https://www-users.cs.umn.edu/~zhan3248/materials/AllPossibleChains.py, 2017.

[5] AT&T FlexWare. https://www.business.att.com/solutions/Service/network-services/sdn-nfv/virtual-network-functions, 2017.

[6] Berkeley Extensible Software Switch. http://span.cs.berkeley.edu/bess.html, 2017.

[7] BRO: Network Intrusion Detection System. https://www.bro.org/, 2017.

[8] Cisco IOS. https://www.cisco.com/c/en/us/products/ios-nx-os-software/index.html, 2017.

[9] Cisco's Vector Packet Processing. https://wiki.fd.io/view/VPP, 2017.

[10] Consistent Hashing. https://en.wikipedia.org/wiki/Consistent_hashing, 2017.

[11] DPDK. http://dpdk.org/, 2017.

[12] DPDK OVS. https://clearlinux.org/documentation/ac-ovs-dpdk.html, 2017.

[13] Flask Python Framework. https://www.fullstackpython.com/flask.html, 2017.

[14] Junos OS. https://www.juniper.net/us/en/products-services/nos/junos/, 2017.

[15] Linux Network Load Balancing. http://lnlb.sourceforge.net/, 2017.

[16] Network Monitor. https://linux.die.net/man/1/nload, 2017.

[17] OpenDaylight: Open Source SDN Platform. https://www.opendaylight.org/, 2017.

[18] OpenVPN: VPN Gateway. https://openvpn.net/, 2017.

[19] PickleDB: Python Key Value Store. https://pythonhosted.org/pickleDB/, 2017.

[20] RYU SDN Framework. https://osrg.github.io/ryu/, 2017.

[21] ANWER, B., BENSON, T., FEAMSTER, N., AND LEVIN, D. Programming Slick Network Functions. In *Proc. SOSR* (2015).

[22] BANERJEE, A., MAHINDRA, R., SUNDARESAN, K., KASERA, S., DER MERWE, K. V., AND RANGARAJAN, S. Scaling the LTE Control-Plane for Future Mobile Access. In *Proc. CoNEXT* (2015).

[23] BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. *Concurrency Control and Recovery in Database Systems*. Addison Wesley Publishing Company, 1987.

[24] BOSSHART, P., ET AL. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM CCR* (2014).

[25] BREMLER-BARR, A., HARCHOL, Y., AND HAY, D. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In *Proc. SIGCOMM* (2016).

[26] CHIU, A., GOPALAKRISHNAN, V., HAN, B., KABLAN, M., SPATSCHECK, O., WANG, C., AND XU, Y. EdgePlex: decomposing the provider edge for flexibilty and reliability. In *Proc. SOSR* (2015).

[27] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. *Proc. OSDI* (2004).

[28] FAYAZ, S. K., ET AL. Buzz: Testing context-dependent policies in stateful networks. In *Proc. NSDI* (2016).

[29] FAYAZBAKHSH, S. K., ET AL. Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags. In *Proc. NSDI* (2014).

[30] GEMBER-JACOBSON, A., ET AL. OpenNF: Enabling Innovation in Network Function Control. In *Proc. SIGCOMM* (2014).

[31] GRANDL, R., KANDULA, S., RAO, S., AKELLA, A., AND KULKARNI, J. GRAPHENE: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters. In *Proc. OSDI* (2016).

[32] HALPERN, J. M., AND PIGNATARO, C. Service Function Chaining (SFC) Architecture, 2015.

[33] HAN, B., GOPALAKRISHNAN, V., JI, L., AND LEE, S. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine* (2015).

[34] HARTERT, R., ET AL. PGA: Using Graphs to Express and Automatically Reconcile Network Policies. *Proc. SIGCOMM* (2015).

[35] HENKE, C., SCHMOLL, C., AND ZSEBY, T. Empirical Evaluation of Hash Functions for Multipoint Measurements. *SIGCOMM CCR* (2008).

[36] HENNESSY, J. L., AND PATTERSON, D. A. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[37] HONDA, M., HUICI, F., LETTIERI, G., AND RIZZO, L. mSwitch: A Highly-Scalable, Modular Software Switch. *Proc. SOSR* (2015).

[38] HWANG, J., RAMAKRISHNAN, K. K., AND WOOD, T. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *Proc. NSDI* (2014).

[39] JOSEPH, D., AND STOICA, I. Modeling middleboxes. *IEEE Network: The Magazine of Global Internetworking* (2008).

[40] JOSEPH, D. A., TAVAKOLI, A., AND STOICA, I. A Policy-aware Switching Layer for Data Centers. In *Proc. SIGCOMM* (2008).

[41] KAZEMIAN, P., CHANG, M., ZENG, H., VARGHESE, G., MCKEOWN, N., AND WHYTE, S. Real Time Network Policy Checking using Header Space Analysis. In *Proc. NSDI* (2013).

[42] KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Header space analysis: Static checking for networks. In *Proc. NSDI* (2012).

[43] KHALID, J., GEMBER-JACOBSON, A., MICHAEL, R., ABHASHKUMAR, A., AND NSDI, I. Paving the Way for NFV : Simplifying Middlebox Modifications Using StateAlyzr This paper is included in the Proceedings of the. *Proc. NSDI* (2016).

[44] KULKARNI, S. G., ZHANG, W., HWANG, J., RAJAGOPALAN, S., RAMAKRISHNAN, K., WOOD, T., ARUMAITHURAI, M., AND FU, X. NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains. In *Proc. SIGCOMM* (2017).

[45] KUMAR, S., ET AL. Service Function Chaining Use Cases In Data Centers. Tech. rep., IETF, 2016.

[46] LI, B., ET AL. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proc. SIGCOMM* (2016).

[47] LI, Z., ZHANG, M., ZHU, Z., CHEN, Y., GREENBERG, A., AND WANG, Y.-M. WebProphet: Automating Performance Prediction for Web Services. In *Proc. NSDI* (2010).

[48] MARTINS, J., ET AL. ClickOS and the Art of Network Function Virtualization. In *Proc. NSDI 14* (2014).

[49] MEKKY, H., HAO, F., MUKHERJEE, S., ZHANG, Z.-L., AND LAKSHMAN, T. Network function virtualization enablement within sdn data plane. In *Proc. of INFOCOM* (2017).

[50] MOGUL, J. C., TOURRILHES, J., YALAGANDULA, P., SHARMA, P., CURTIS, A. R., AND BANERJEE, S. Devoflow: Cost-effective flow management for high performance enterprise networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks* (New York, NY, USA, 2010), Hotnets-IX, ACM, pp. 1:1–1:6.

[51] NADEAU, T., AND QUINN, P. Problem Statement for Service Function Chaining. RFC 7498, 2015.

[52] NAPPER, J., ET AL. Service Function Chaining Use Cases in Mobile Networks. Tech. rep., IETF, 2016.

[53] PALKAR, S., ET AL. E2: A Framework for NFV Applications. In *Proc. SOSP* (2015).

[54] PATEL, P., BANSAL, D., YUAN, L., MURTHY, A., GREENBERG, A., MALTZ, D. A., KERN, R., KUMAR, H., ZIKOS, M., WU, H., KIM, C., AND KARRI, N. Ananta: Cloud scale load balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (New York, NY, USA, 2013), SIGCOMM '13, ACM, pp. 207–218.

[55] QAZI, Z. A., ET AL. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proc. SIGCOMM* (2013).

[56] QAZI, Z. A., PENUMARTHI, P. K., SEKAR, V., GOPALAKRISHNAN, V., JOSHI, K., AND DAS, S. R. KLEIN: A Minimally Disruptive Design for an Elastic Cellular Core . In *Proc. SOSR* (2016).

[57] RAJAGOPALAN, S., ET AL. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proc. NSDI* (2013).

[58] STOENESCU, R., POPOVICI, M., NEGREANU, L., AND RAICIU, C. Symnet: Scalable symbolic execution for modern networks. In *Proc. SIGCOMM* (2016), Proc. SIGCOMM.

[59] SUN, C., BI, J., ZHENG, Z., YU, H., AND HU, H. NFP: Enabling Network Function Parallelism in NFV. In *Proc. SIGCOMM* (2017).

[60] WALFISH, M., ET AL. Middleboxes No Longer Considered Harmful. In *Proc. OSDI* (2004).

[61] WANG, R., BUTNARIU, D., AND REXFORD, J. Openflow-based server load balancing gone wild. In *Proceedings of the 11th USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services* (Berkeley, CA, USA, 2011), Hot-ICE'11, USENIX Association, pp. 12–12.

[62] WANG, X. S., BALASUBRAMANIAN, A., KRISHNAMURTHY, A., AND WETHERALL, D. Demystifying Page Load Performance with WProf. In *Proc. NSDI* (2013).

[63] WU, W., ZHANG, Y., AND BANERJEE, S. Automatic synthesis of nf models by program analysis. In *Proc. HotNets* (2016).

[64] ZAVE, P., FERREIRA, R. A., ZOU, X. K., MORIMOTO, M., AND REXFORD, J. Dynamic Service Chaining with Dysco. In *Proc. SIGCOMM* (2017).

[65] ZHANG, W., HWANG, J., RAJAGOPALAN, S., RAMAKRISHNAN, K., AND WOOD, T. Flurries: Countless Fine-Grained NFs for Flexible Per-Flow Customization. In *Proc. CoNEXT* (2016).

[66] ZHANG, W., LIU, G., ZHANG, W., SHAH, N., LOPREIATO, P., TODESCHI, G., RAMAKRISHNAN, K., AND WOOD, T. OpenNetVM: A Platform for High Performance Network Service Chains. In *Proc. of HotMIddlebox* (2016).

[67] ZHANG, Y., ANWER, B., GOPALAKRISHNAN, V., HAN, B., REICH, J., SHAIKH, A., AND ZHANG, Z.-L. ParaBox: Exploiting Parallelism for Virtual Network Functions in Service Chaining. In *Proc. SOSR* (2017).

[68] ZHANG, Y., ET AL. StEERING: A software-defined networking for inline service chaining. In *Proc. ICNP* (2013).