Automatic Performance-Optimal Offloading of Network Functions on Programmable Switches

Xiang Chen, *Member, IEEE*, Hongyan Liu, Dong Zhang, Zili Meng, Qun Huang, Haifeng Zhou, Chunming Wu, Xuan Liu, *Senior Member, IEEE*, and Qiang Yang, *Senior Member, IEEE*

Abstract—In network function virtualization (NFV), network functions (NFs) are chained as a service function chain (SFC) to enhance NF management with low cost and high flexibility. Recent NFV solutions indicate that the packet processing performance of SFCs can be significantly improved by offloading NFs to programmable switches. However, such offloading requires a deep understanding of heterogeneous NF properties (e.g., NF resource consumption and NF performance behaviors) to achieve the maximum SFC performance. Unfortunately, none of existing solutions provide automatic analysis of these NF properties. Thus, network administrators have to manually examine the source codes of NFs and profile various NF properties by hand, which is extremely time-consuming and laborious. In this paper, we propose LightNF, a novel system that simplifies NF offloading in programmable networks. LightNF automatically dissects comprehensive NF properties by means of code analysis and performance profiling while eliminating manual efforts. It then leverages its analysis results of NF properties in its SFC placement so as to make the performance-optimal offloading decisions. We have implemented LightNF on Tofino-based hardware programmable switches. We perform extensive experiments to evaluate LightNF with a real-world testbed and large-scale simulation. Our experiments show that LightNF outperforms state-of-the-art solutions with an orders-of-magnitude reduction in per-packet processing latency and 9.5× improvement in SFC throughput.

Index Terms—Network functions, service function chains, packet processing performance, programmable switches.

1 Introduction

In network function virtualization (NFV), network administrators often deploy several sequential chains of network functions (NFs), commonly referred as *service function chains* (SFCs), to provide network services. Compared to traditional consolidated middleboxes, the SFC achieves higher flexibility in NF management and reduces overall costs. However, software-based SFCs suffer from poor performance due to their limited processing capability [1]. For example, Ananta Muxes incurs a latency from 200 μs to 1 ms [1], which violates the requirements of latency-sensitive applications like web search. To improve SFC performance, recent researches [2, 3, 4, 5, 6, 7, 8, 9] offload NFs onto programmable switches. As programmable switches guarantee line-rate packet processing performance, such offloading could bring significant SFC performance improvement. For example, according to our experiments in §2, offloading NFs to programmable switches can improve the throughput by up to 2× and reduce the per-packet processing latency by around 88% compared to software-based NFs.

In particular, network administrators are supposed to make the *performance-optimal* offloading decisions to maximize the benefits of NF offloading in SFC performance. However, it remains non-trivial and burdensome for administrators to achieve the performance-optimal offloading. Specifically, administrators face four non-trivial questions when offloading NFs based on their own experiences.

- **Suitability**. Given a set of NFs, administrators need to determine whether the operations of these NFs can be realized in a programmable switch or not. If an NF uses some complicated operations (e.g., payload encryption) that are forbidden by programmable switches, the NF is unsuitable for offloading on programmable switches.
- **Resource consumption**. Given a set of NFs, administrators need to understand how many memory and computational resources do these NFs need. Otherwise, their offloading decisions may exhaust limited switch resources, leading to SFC deployment failures.
- **Performance behaviors**. Given a set of NFs, administrators need to determine the performance benefits of offloading each NF to programmable switches. Only by understanding accurate performance behaviors of NFs can administrators make the performance-optimal offloading decisions on programmable networks.
- **NF dependencies**. Given an SFC, administrators need to identify all the inherent execution dependencies between the NFs resided in the SFC. They also need to preserve these dependencies when placing NFs across

Xiang Chen, Hongyan Liu, and Chunming Wu are with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310007, China. E-mail: wasdnsxchen@gmail.com, hyliu20,wuchunming@zju.edu.cn.
Dong Zhang is with the College of Computer Science and Big Data, Fuzhou University, Fuzhou 350116, China. E-mail: zhangdong@fzu.edu.cn.
Zili Meng is with the Institute for Network Science and Cyberspace, Tsinghua University, Beijing 100084, China. Email: zilim@ieee.org.
Qun Huang is with the School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China. Email: huangqun@pku.edu.cn.
Haifeng Zhou is with the College of Control Science and Engineering, Zhejiang University, Hangzhou 310007, China. Email: zhouhaifeng@zju.edu.cn.
Xuan Liu is with the College of Information Engineering (College of Artificial Intelligence), Yangzhou University, China, and the School of Computer Science and Engineering, Southeast University, China. Email: yusuf@yzu.edu.cn.
Qiang Yang is with the College of Electrical Engineering, Zhejiang University, Hangzhou 310007, China. Email: qyang@zju.edu.cn.
Manuscript received August 19, 2021; revised: January 30, 2022
Corresponding authors: Dong Zhang, Qun Huang, and Chunming Wu.

programmable switches and commodity servers. Otherwise, their offloading will violate these dependencies, leading to incorrect packet processing results.

To address the aforementioned questions, administrators have to manually analyze the source codes of NFs to obtain NF properties. However, such manual analysis is extremely time-consuming and laborious. Even worse, the analysis results of NF properties are heterogeneous. The integration of these heterogeneous results in SFC placement remains complicated and challenging for administrators to make the performance-optimal offloading decisions. Moreover, to the best of our knowledge, none of existing solutions support automatic and comprehensive analysis on various NF properties, not to mention integrating these properties for SFC placement. In a nutshell, simplifying the analysis and integration of heterogeneous NF properties is still a challenging yet critical issue for practical NF offloading on programmable switches.

In this paper, we propose LightNF, an NFV system that achieves automatic and performance-optimal NF offloading on programmable switches. The goal of LightNF is to automatically analyze NF properties and make the performance-optimal NF offloading decisions based on its analysis results of NF properties. To alleviate the burdens of NF offloading for administrators, LightNF provides a suite of high-level primitives for the description of NFs and SFCs. With these primitives, administrators only need to describe their demanded NFs and SFCs based on their intent, and submit their requests to LightNF. Then LightNF will automatically complete the remaining steps to achieve the performance-optimal NF offloading. By that means, administrators do not need to manually examine NF source codes and consider low-level offloading details in SFC deployment, which significantly reduces their burdens.

Specifically, LightNF employs an *SFC analyzer* to automatically dissect NF properties from input LightNF primitives. The SFC analyzer (1) analyzes the suitability of each NF by determining whether all the NF operations can be offloaded to the programmable switch via code analysis, (2) obtains resource consumption of NFs in both the programmable switch and the commodity server by leveraging static compiling and dynamic runtime analysis, (3) accurately and timely profiles performance behaviors of NFs, and (4) infers execution dependencies among NFs via code analysis. With the analysis results of NF properties, LightNF builds an optimization framework that formulates SFC placement with NF properties as constraints. The optimization framework determines how to place the SFC on a hybrid network with several programmable switches and servers while maximizing SFC performance.

We have implemented LightNF on Tofino-based hardware programmable switches [10]. We conduct extensive experiments on a real-world testbed to evaluate LightNF. The experimental results indicate that LightNF offers two orders of magnitude latency reduction and 9.5× throughput improvement compared to state-of-the-art solutions.

Contributions. This paper makes five main contributions.

- We identify the user burdens of offloading NFs to programmable switches (§2), and propose LightNF, a novel NFV system, to alleviate these burdens (§3).
- We design a suite of LightNF primitives for administrators to describe NFs and SFCs based on their intent (§3.2).
- We design an SFC analyzer to dissect various NF properties, including NF suitability, resource consumption, performance behaviors, and dependencies (§3.3). The analysis is completely done automatically, without requiring any domain-specific knowledge and parameters.
- We design an optimization framework that leverages NF properties to obtain the performance-optimal SFC placement that maximizes SFC performance (§3.4).
- We have implemented LightNF on Tofino-based hardware programmable switches [10]. We conduct extensive testbed experiments to evaluate LightNF. The experimental results show that LightNF offers two orders of magnitude latency reduction and 9.5× throughput improvement compared to state-of-the-art solutions (§4).

A preliminary version of this paper appears as a conference paper published by IEEE/ACM IWQoS 2021 conference [11]. In the preliminary version, we present the high-level idea of LightNF and conduct preliminary experiments to evaluate LightNF. Compared to that paper, we have made substantive enhancements in this manuscript. First, we demonstrate the performance benefits of offloading NFs to programmable switches via testbed experiments (§2.1). Second, we comprehensively survey existing solutions for SFC placement and discuss their limitations (§2.3). Third, we present the detailed description of LightNF primitives and illustrate the approaches of integrating new NFs into LightNF (§3.2). Fourth, we present more technical details of the SFC analyzer (§3.3). In particular, we justify our observation of NF latency with detailed micro-benchmark experiments (§3.3.3). Fifth, we show how LightNF performs SFC deployment and runtime management on the substrate network (§3.5). Finally, we present more experimental results to demonstrate the benefits of LightNF (§4).

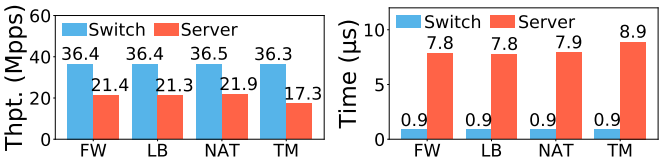
2 Background and Motivation

In this section, we first demonstrate the performance benefits of offloading NFs to programmable switches. Then we discuss the challenges of such offloading and elaborate the limitations of existing solutions.

2.1 Offloading NFs to Programmable Switches

To date, there are four approaches of realizing NFs. (1) The traditional approach leverages proprietary hardware, i.e., standalone middleboxes, to realize each NF with a dedicated appliance [12]. However, middleboxes have been proved as rigid and inflexible, which is discarded by modern networks seeking for elastic network management. (2) The NFV-based approach runs NFs on top of off-the-shelf servers as virtualized NFs so as to enhance flexible NF management at runtime [13, 14]. However, due to the performance overhead incurred by virtualization, this approach sacrifices NF performance by several orders of magnitude, making NFs fail to keep up with high-speed traffic [1, 15]. (3) Many solutions [15, 16, 17] offload NFs to SmartNICs, which provide significant performance benefits to NFV. (4) Motivated by the SmartNIC offloading and the emergence of programmable switches, researchers propose to transform NFs into resource-efficient and hardware-compatible

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2022.3149817, IEEE Transactions on Cloud Computing



(a) Throughput (b) Latency

Fig. 1: Packet processing performance of NFs running on programmable switches vs. that of software-based NFs running on commodity servers. With NF offloading, the performance of NFs is significantly improved.

ones while fully offloading them to programmable switches [7, 8, 18, 19, 20]. According to latest studies [19, 20, 21, 22, 23], programmable switches yield Tbps-level throughput and sub-millisecond per-packet processing latency while outperforming SmartNICs with more resources and lower energy consumption. Given those benefits, offloading NFs to switches has become one of the top choices in production networks [18, 24, 25].

The existing approach of offloading NFs to switches mainly comprises two steps [7, 8, 18]. First, administrators build the NFs with network programming languages such as P4 [26] based on their intent. Examples include (1) traditional NFs such as NAT [18] and load balancers [2, 27], (2) NFs for distributed systems, including in-network key-value caches [3], distributed coordination [5], VXLAN gateways [25], lock manager [28], (3) NFs for network measurement, including sampling-based NFs [29] and sketch-based NFs [30, 31, 32], and (4) NFs for security such as DDoS attack defense [19, 20]. Second, administrators make their offloading decisions that determine which NFs to be offloaded to which devices with the objective of maximizing the overall performance of NFs. Since programmable switches provide superior performance, the decisions often prefer to offload as many NFs to switches as possible. Note that each NF typically consumes a portion of switch resources and can be entirely offloaded to a single switch. Thus, it is common that multiple NFs are offloaded to the same switch [7, 8, 18, 19, 20]. For example, in the defense of distributed denial-of-service (DDoS) attacks, administrators usually deploy the NFs defending against various types of DDoS attacks on the single switch at the same time [19, 20].

We conduct experiments to validate the performance benefits of NF offloading on programmable switches. We select four widely-used NFs: firewall (FW), load balancer (LB), network address translator (NAT), and traffic monitor (TM). We implement these NFs and deploy them on a Tofino-based hardware switch [10]. We compare these NFs with their corresponding software implementations. The software-based NFs are implemented based on E2 [33], a mature NFV framework. We deploy each software-based NF in a server with twelve 2.3GHz CPU cores. We configure each NF with 100 rules. Also, we dedicate the entire capacity of server resources (e.g., CPU cores) to the NF to maximize its performance. We generate 64-byte packets to stress-test each NF at 40 Gbps. Figure 1 presents the results. For throughput (Figure 1(a)), all P4-based NFs achieve a throughput of 36 Mpps, while the throughput of software-based NFs is much lower. For per-packet processing latency (Figure 1(b)), NF offloading reduces latency by around 88% for all NFs. Such performance benefits are necessary for realtime applications. For example, distributed memory cache [34] can only be tolerant of a few microseconds.

TABLE 1: Commonly-used NFs and their suitability of offloading on programmable switches.

Network Function	Suitable?	Unsuitable Reason
Caching	✓[3]	-
Congestion control	✓[35]	-
Congestion detection	✓[36]	-
Coordination	✓[5]	-
Firewall, ACL	✓[7]	-
Key-value store	✓[3]	-
L2/L3 switch, router	✓[7]	-
Load balancer	✓[2]	-
Mobile serving gateway	✓[37]	-
NAT, CGNAT, NAT64	✓[7]	-
QoS mapping	✓[38]	-
String search	✓[39]	-
Traffic monitor	✓[7]	-
Traffic shaper	✗	Buffer packets
Deep packet inspection	✗	Read or write payload
IDS / IPS	✗	Read or write payload
Packet compression	✗	Read or write payload
Trojan detector	✗	Stateful computation
Top-K ranker	✗	Stateful computation
Virtual private network	✗	Read or write payload

We further analyze twenty commonly-used NFs and determine whether these NFs are suitable for offloading on programmable switches. We present more details of analyzing NF suitability in §3.3.1. Table 1 presents our analysis results. The results indicate that a substantial portion (65%) of NFs are offloadable. Thus, it is totally worthy for administrators to exploit the opportunities of offloading NFs to enhance the packet processing performance of NFV.

2.2 Challenges of Offloading NFs

Given an input SFC, our goal is to find out the *performance-optimal placement* that places each NF in the SFC on either a programmable switch or a server with the objective of maximizing SFC performance. However, it is non-trivial to make the performance-optimal offloading decisions, which requires deep understanding of NF properties. We summarize two main challenges in what follows.

① **Dissecting NF properties.** It is necessary for administrators to carefully analyze NF properties to make reasonable offloading decisions. However, analyzing properties is not straightforward since these properties are complicated and vary across user-specified NFs. We present the challenge of identifying each type of NF properties in what follows.

- **NF suitability**. Existing programmable switches adopt a pipeline-based paradigm for packet processing [40]. The paradigm partitions packet processing into several stages. For realizability, each stage can only perform a limited number of simple actions on each arrival packet within a small time budget [40]. Such a limitation prohibits many complicated NF operations, including loop, payload encryption, and buffering, from being deployed on programmable switches. However, identifying whether an NF includes unoffloadable operations, which make the NF unsuitable for offloading, requires domain knowledge. Existing solutions fail to address this challenge due to the lack of accurate analysis towards the limitations of programmable switches.

- **Resource limitation**. Every NF occupies a portion of memory to store its rules and processing status. Further, it also consumes several types of computational resources (e.g., action units) for packet processing. However, both memory and computational resources in a device are scarce. For example, a PISA-based switch only provides the capacity of around 3 M rules [40], which is far less than the amount required by a commodity load balancer to maintain millions of connections [2]. Fulfilling the resource limitations requires careful resource estimation and scheduling, incurring the burdens of NF offloading. Existing solutions [7, 35] not only lack a comprehensive and accurate analysis of NF resource consumption, but also overlook the limitation of computational resources in programmable switches.
- **Performance variance**. The suitability and resource limitations imply that only partial NFs in an SFC are offloadable in practice. On the other hand, to maximize performance gain, administrators need to carefully offload NFs with a deep understanding of the performance behaviors of each NF in both programmable switches and commodity servers. In detail, the performance analysis requires both high accuracy and timeliness. However, none of these requirements is easy to achieve due to two reasons. First, NF performance varies as traffic patterns or NF configurations change [41], making it difficult to achieve high accuracy. Second, exhaustively testing NF performance under every possible situation is time-consuming and laborious.
- **Dependency preserving**. Offloading NFs may change the execution order among NFs. Consider an example SFC "FW⇒IDS⇒NAT", in which FW and NAT are offloadable but IDS is not. If we offload FW and NAT onto the same programmable switch, the resulting SFC becomes "FW⇒NAT⇒IDS". However, since NAT modifies IP addresses matched by IDS, such offloading violates the original dependency between IDS and NAT. Thus, we need to assign NFs in a right order to preserve NF dependencies. However, for a complex SFC, identifying all dependencies is even a hard problem, not to mention preserving these dependencies during SFC placement.

② **Integrating NF properties.** Even when NF properties are well dissected, it remains challenging to integrate these heterogeneous properties to make offloading decisions. For example, the offloading should deal with both deterministic (and strict) resource limitations and stochastic performance variances within a single decision framework. Further, it is non-trivial to formulate NF dependencies as constraints, to say nothing of integrating it with other properties.

2.3 Related Works

However, none of existing solutions are capable of resolving the above problems. Specifically, we classify existing solutions into two main categories. We illustrate each category of existing solutions and discuss its limitations as follows.

NFV frameworks for offloading NFs. In recent years, many solutions are proposed to offload various types of NFs, such as load balancers [2], key-value caches [3], data aggregation [4], coordination [5], and neural network training [6], to programmable switches to improve NF performance. However, these solutions focus on offloading a specific NF rather than SFCs. On the other hand, P4SC [7] and Dejavu [8] offload entire SFCs to programmable switches. They overlook NF suitability on programmable switches, such that their placement may be failed. SPEED [42] and Lyra [24] support the network-wide deployment of data plane programs on several programmable switches. However, these solutions do not target high-performance NFV so that they cannot perform comprehensive analysis on NF properties. Moreover, Metron [35] only offloads stateless NFs while deploying other NFs on servers. Gallium [9] offers a compiler-oriented approach to partition an SFC between a programmable switch and a server. P4NFV [43] deploys NFs on either servers or programmable switches while enabling switch management at runtime. P4-SFC [44] adopts programmable switches as flow classifiers to direct traffic among different software-based NFs. HyperVDP [45] virtualizes programmable networks to enable multi-tenancy. However, none of these solutions conduct a deep analysis of NF properties, leading to failed or sub-optimal SFC placement. For example, according to our experiments in §4, Metron incurs deployment failure when deploying several concurrent SFCs due to its lack of NF analysis.

In addition to the switch-based NF offloading, previous systems [15, 16, 17] choose to offload NFs to SmartNICs. For example, SoftNIC [16] offers a hardware-abstraction layer to enable the programming of NFs atop the architecture of SmartNICs; ClickNP [15] provides a modular approach for administrators to assemble a number of operations into SmartNIC-compatible NFs; UNO [17] presents a central-controlled NF offloading architecture for making the best use of SmartNICs and host packet processing capabilities. Our work focuses on NF offloading on programmable switches, which is complementary to the above solutions.

NFV frameworks for high-performance SFCs. Some existing NFV frameworks [33, 46, 47] present trade-offs between SFC performance and available resources in SFC placement. However, they only deploy SFCs on servers, while offloading NFs to programmable switches remains non-trivial. Cohen *et al*. [48] formulates the problem of SFC placement with the objective of minimizing both the deployment and connection cost of NFs. It proposes heuristics to solve the problem with guarantees on SFC performance. However, this work does not take the resource constraints of network devices into account. It also lacks of analyzing NF properties, leading to sub-optimal SFC placement. Moreover, the authors of [49] optimizes SFC placement on the hybrid network scenario in which NFs can be realized using either traditional middleboxes or virtualization techniques. The work [50] designs a heuristic based on dynamic programming to efficiently place SFCs. Xia *et al*. [51] models the SFC placement in optical inter-datacenter networks as a binary integer programming problem with the objective of minimizing the cost of optical-electronic-optical conversions. The work [52] predicates future resource requirements of NFs based on graph neural networks so as to achieve elastic NF resource allocation. Nevertheless, none of the aforementioned solutions take some essential NF properties such as NF performance behaviors into account. Thus, their SFC placement is still sub-optimal in packet processing

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2022.3149817, IEEE Transactions on Cloud Computing

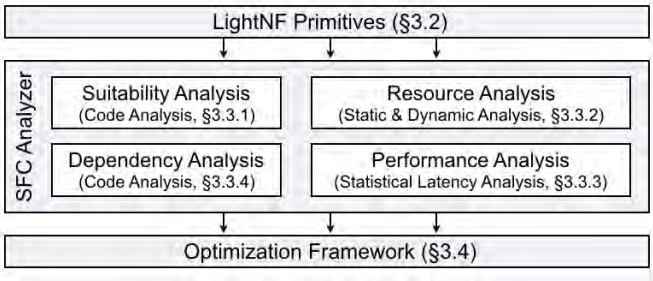


Fig. 2: Overview of LightNF.

performance. In addition, some solutions leverage packet delivery acceleration [53], NF modularization [54], and NF parallelism [1] to accelerate SFCs while high-performance network I/O engines such as DPDK [55] and PF_RING [56] boost up SFC performance via kernel bypassing. However, these studies are still based on software, thus suffering from limited CPU capability and highly variable latency.

Summary. None of existing NFV frameworks can provide automatic analysis of comprehensive NF properties, not to mention making the performance-optimal offloading decisions. In response, we aim to design a system that provides an in-depth analysis of comprehensive NF properties while eliminating manual efforts to reduce user burdens. With the analysis results of properties, the system will automatically formulate these results as constraints to address challenges and achieve the performance-optimal SFC placement.

3 LightNF Design

In this section, we present an overview of LightNF architecture (§3.1). Then we elaborate the components of LightNF, including LightNF primitives (§3.2), the SFC analyzer (§3.3), and the optimization framework (§3.4).

3.1 Overview of LightNF

In this paper, we propose a novel NFV framework, namely LightNF, to handle the challenges of offloading NFs (§2.2). LightNF provides both the automatic analysis of NF properties and the performance-optimal SFC placement. It comprises three main components, including LightNF primitives, the SFC analyzer, and the optimization framework.

As shown in Figure 2, LightNF first offers a set of intent-based primitives for administrators to describe NFs and SFCs in a high-level manner (§3.2). Next, the SFC analyzer dissects the properties of the NFs realized by LightNF primitives (Section 3.3). According to the characteristics of NF properties, it classifies NF properties into four types, including (1) NF suitability, (2) NF resource consumption, (3) NF performance behaviors, and (4) NF dependencies. For each type of NF properties, it proposes a specific analysis strategy, which is built on the techniques of static code analysis and dynamic performance profiling. Such analysis addresses the challenge of dissecting NF properties. Finally, LightNF formulates an optimization framework to make the best offloading decisions (Section 3.4). The framework carefully integrates the analysis results of NF properties into its optimization objective and constraints. By this means, it makes the performance-optimal NF offloading decisions with respect to NF properties while preserving all the relevant constraints such as NF dependencies. Thus, it addresses the challenge of property integration.

TABLE 2: LightNF primitives ("O" refers to "Offloadable").

Name	O?	Description
Atomic NF operations		
`add_header`	✓	Add a new packet header (e.g., VLAN tag) to the packet. The new header can reside in an arbitrary location on the packet.
`alert`	✓	Raise an alert to the control plane. The alert contains the flow ID (e.g., 5-tuple), which is used to identify a potential malicious flow.
`broadcast`	✓	Broadcast a packet (or the five-tuple of the packet) to multiple output ports.
`copy`	✓	Copy a packet. The packet and its replica can be processed by different subsequent operations (e.g., sending to different ports).
`count`	✓	Count packets. It can be used to create per-flow counters or hash tables.
`decrypt`	✗	Decrypt a packet. It uses the AES algorithm by default. Users can change the algorithm.
`dequeue`	✗	Pop a packet from a first-in-first-out queue.
`drop`	✓	Drop packets.
`encrypt`	✗	Encrypt a packet. It uses the AES algorithm by default. Users can change the algorithm.
`enqueue`	✗	Send a packet to a first-in-first-out queue.
`header_classify`	✓	Classify packets based on the user rules that match specific values of packet header fields.
`modify_field`	✓	Modify the value of a packet field.
`modify_payload`	✗	Modify the value of the packet payload.
`output`	✓	Send a packet to a destination port.
`payload_classify`	✗	Classify packets based on the user rules that match specific values of payloads.
`rate_limit`	✗	Limit traffic rate by buffering packets.
`read_state`	✗	Use the flow ID of a specific flow to lookup the state (e.g., packet count) associated with the flow.
`recirculate`	✓	Recirculate a packet. The packet will be processed by previous operations again.
`remove_header`	✓	Remove a specific header from the packet.
`update_state`	✗	Update the value of the state associated with a flow, which is identified by a specific ID.
Primitive for building NFs (u, v are NF operations)		
`Link`(u, c, v)	-	Execute v after u if c happens. c is `hit` when the packet matches one rule of u; c is `miss` if the packet does not match any rules of u; c is `all` if the packet has been processed by u.
Primitive for building SFCs (u, v are NFs)		
`Connect`(u, v)	-	Create an edge from u to v, such that u will be invoked before v.

3.2 LightNF Primitives

In LightNF, we design three classes of intent-based primitives to support administrators to describe NFs and SFCs on demand. With LightNF primitives, administrators only need to care about their intent of NF and SFC design without considering unnecessary offloading details. Table 2 presents a non-exhaustive list of LightNF primitives.

Atomic NF operations. The first class of LightNF primitives comprises basic and atomic NF operations. In particular, we observe that even though NFs are diverse, they use many atomic operations such as matching headers and forwarding packets [54]. For example, both firewall and layer-3 forwarding invoke `header_classify` to match incoming packets and process packets based on matching results. Thus, LightNF offers a set of atomic NF operations to allow administrators to compose NFs on their demands.

Primitive for building NFs. The second class contains a primitive used to assemble atomic operations into NFs. The primitive, `Link`(u, c, v), specifies that the operation v is invoked if the execution result of the pre-visit operation u equals c. Here, c can be `hit` (i.e., the packet hits one of u

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2022.3149817, IEEE Transactions on Cloud Computing

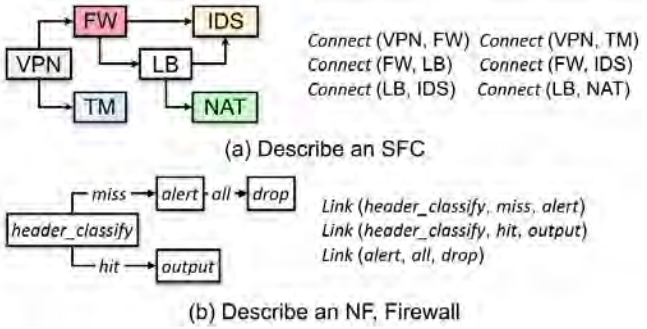


Fig. 3: Describing an SFC with LightNF primitives.

rules), or `miss` (i.e., the packet misses all u rules), or `all` (i.e., arbitrary results are acceptable). LightNF also offers tens of ready-to-use NFs that can be directly invoked to ease development burdens.

Primitive for building SFCs. The other primitives are used to assemble NFs into a complete SFC. We abstract an SFC as a directed acyclic graph, where each node corresponds to an NF. A directed edge implies that the packets sent by the source NF are consumed as input by the destination NF. We design a primitive to build the SFC. Specifically, we offer `Connect` to build a directed edge connecting two NFs.

Example. Figure 3(a) plots an example of using LightNF primitives to describe an SFC. The SFC has six NFs and six edges, such that we use six `Connect` primitives to establish the edges between NFs in the SFC. Moreover, Figure 3(b) details the assembling of four operations of the NF Firewall. For each packet, Firewall first invokes `header_classify`. If the execution result of `header_classify` is `miss`, implying that the packet may be malicious, Firewall raises an `alert` and `drop` the packet. Otherwise, the packet is considered as innocent so that Firewall `output` it.

Integration of new NFs. LightNF offers two approaches for administrators to integrate new NFs. First, administrators can directly invoke existing atomic NF operations offered by LightNF to compose new NFs. Second, if their NFs invoke new NF operations, LightNF offers an intuitive interface to administrators to submit the source codes of these new operations. These operations will be automatically registered into LightNF. For example, administrators may adopt sketch algorithms (e.g., Count-Min sketch) in their NFs to perform accurate and resource-efficient traffic measurement in the scenario of software-defined measurement. In this context, since sketch algorithms have not been incorporated into LightNF, administrators can submit the codes of their desired algorithms via the interface offered by LightNF. LightNF will register each algorithm as an individual NF operation in its primitives. Then administrators can exploit the first approach to invoke sketch algorithms to compose their NFs. In our current implementation, we provide both a P4-based interface that receives the offloadable NF operations written in P4 and a DPDK-based interface that takes the operations built by DPDK as input. The two interfaces automate the workflow of NF integration in LightNF, which reduces user burdens.

Benefits. LightNF primitives offer benefits to both administrators and system design. For *administrators*, these primitives are intent-based and enable practical NF offloading. With these primitives, administrators only need to care about their intents: operations of individual NFs and their expected assembling as an SFC. LightNF hides all details on resource consumption, performance behaviors, and execution dependencies by automatically dissecting them. For *system design*, LightNF primitives abstract NF behaviors at the granularity of operations. This enables further operation-level analysis of NF suitability and dependencies. We illustrate this benefit in §3.3.

3.3 SFC Analyzer

The SFC analyzer of LightNF performs four types of analysis to dissect NF properties mentioned in §2.2. We illustrate each type of analysis as follows.

3.3.1 Suitability Analysis

For each NF, the SFC analyzer dissects whether this NF can be offloaded to the programmable switch via code analysis. It first identifies general restrictions of programmable switches, and examines every NF whose operations are specified in LightNF primitives. Then its suitability analysis detects the NF operations that violate switch restrictions. Specifically, if an NF contains any restricted operations, the NF is identified as unsuitable for offloading. Otherwise, the NF is suitable for offloading. Currently, LightNF identifies the following types of restricted operations.

- **Buffering.** Packet buffers are invisible for users in programmable switches due to high manufacturing cost for realizing customizable buffers. Thus, controlling packet buffers is prohibited by most programmable switches. For example, the NF operations, including `rate_limit`, `enqueue`, and `dequeue`, that conduct traffic shaping by queuing packets and scheduling packet queues cannot be offloaded to programmable switches.
- **Manipulating packet payload.** Some NFs process packets by reading or writing packet payload: intrusion detection system (IDS) reads payload based on regular expression matching to filter malicious packets; packet compression compresses payload to reduce packet size; virtual private network (VPN) encrypts and decrypts payload to enhance the privacy of end-to-end communication. However, programmable switches restrict the number of parsed header bytes for each packet due to the limitation of memory. Thus, the payload-oriented operations, such as `modify_payload`, `payload_classify`, `encrypt`, and `decrypt`, are not offloadable yet.
- **Stateful computation.** Many NFs involve stateful computation. For example, top-K ranker finds the k-most largest flows based on historical processing states such as per-flow packet counts; Trojan detector [57] performs stateful packet inspection. However, programmable switches adopt a pipeline-based paradigm to achieve line-rate packet processing [40]. The paradigm only supports one-shot packet processing, which restricts stateful operations, i.e., `read_state` and `write_state`.

The results of suitability analysis identify all the offloadable NFs. With these results, LightNF will choose as many offloadable NFs as possible and offload them to programmable switches while meeting the resource and performance requirements. The remaining NFs will be deployed on commodity servers. Note that our suitability

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2022.3149817, IEEE Transactions on Cloud Computing

analysis can be applied to *arbitrary* NFs as long as these NFs are implemented with our primitives (§3.2).

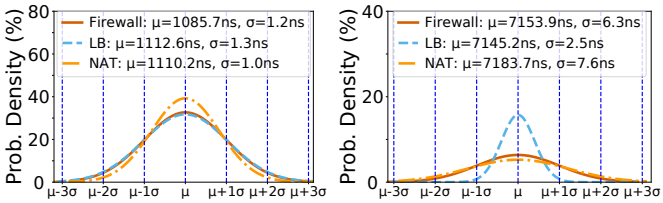
3.3.2 Resource Analysis

The second type of analysis infers the resource usage of each NF. Specifically, each NF contains two properties specified by administrators, i.e., the maximum number of rules, and a set of NF operations. Thus, LightNF considers two types of resources, including memory resources for storing NF rules and computational resources for realizing NF operations. Since an NF can be deployed on either a programmable switch (if offloadable) or a commodity server, the SFC analyzer examines the resource requirements for running each NF on a programmable switch and a server, respectively. Note that there are two execution models of running NFs on a server: (1) the run-to-complete (RTC) model [35] that executes all NFs within a single core, and (2) the pipeline model [33, 53] that assigns an NF to at least one CPU core. Since the pipeline model offers more flexibility in NF management than RTC [41], we design LightNF to analyze in-server NFs under the pipeline model.

- **In-switch resources**. LightNF targets PISA [40], a general programmable switch architecture. For memory resources, it examines both *SRAM* and *TCAM*, which store exact rules and rules with wildcards, respectively. SRAM also maintains stateful information during packet processing. For computational resources, it considers *action units* used to perform actual NF operations, and *packet header vectors* (PHVs) that store packet headers and metadata for exchanging processing status across action units.
- **In-server resources**. For memory resources, LightNF monitors the usage of RAM, which stores rules and computational states. For computational resources, it inspects CPU resources consumed by NFs for packet processing. As each NF is pinned with one or more CPU cores, LightNF quantifies CPU resources in terms of the number of CPU cores instead of CPU cycles.

LightNF employs different strategies to infer in-switch and in-server resource usage, respectively. For in-switch resources, it employs a static method by compiling LightNF primitives via the compiler for programmable switches [58] since the resource usage remains stable after compilation. After compilation, the switch compiler will return a report that presents the detailed switch resource consumption. LightNF parses the report to acquire the consumption of SRAM, TCAM, action units and PHVs. By that means, it obtains the in-switch resource consumption of each NF.

For in-server resources, LightNF employs a dynamic method as the usage varies at runtime. It first runs each NF on a commodity server and injects a high-speed traffic workload to the server. For the traffic workload, it sets the speed of injecting traffic to 40 Gbps while setting the size of each packet to 64 bytes. Such settings simulate the peak workload each server receives at runtime. Next, LightNF incrementally assigns CPU cores to the NF when testing the NF with the workload. It records the number of CPU cores when NF throughput reaches the maximum as the desired number of CPU cores for the NF. Meanwhile, it invokes system tools, e.g., `pmap` [59], to measure the maximum RAM consumption of each NF during its execution. Thereby, it obtains the worst-case in-server resource consumption of the NF, including occupied CPU cores and RAM.



(a) Latency on switch. (b) Latency on server.

Fig. 4: Distribution of NF latency.

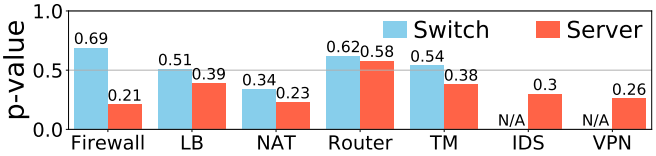


Fig. 5: p-values of NF latency produced by Shapiro-Wilk test (N/A indicates that the NF is unoffloadable).

Note that our strategies in LightNF analyze the worst-case resource consumption of each NF. The reason is that with the worst-case results, LightNF can formulate the per-device resource constraints that prevent NFs from exhausting per-device resource capacity. We illustrate this in §3.4.

3.3.3 Performance Analysis

The third analysis profiles in-switch and in-server performance behaviors of each NF. The measured results are provisioned to the optimization framework for SFC placement.

Latency profiling. LightNF performs accurate and timely profiling of the tail latency of each NF. Here, we choose to quantify the tail NF latency as many applications are latency-sensitive. These applications impose service-level objectives defined by the bounds on tail latency (e.g., the 99-th percentile of latency should be below 10μs) in order to achieve predictable performance [60].

However, characterizing latency is hard because the latency of NFs exhibits significant variance as traffic patterns and NF configurations change. Exploring all possible combinations of traffic patterns and NF configurations is infeasible. In response, LightNF measures NF latency with the worst-case workloads. More precisely, it measures the latency of each NF when processing 1500-byte packets because NFs take longer to process large packets longer than small ones [1, 15] while the maximum size of a packet is typically 1500 bytes due to the Ethernet maximum transmission unit (MTU). It injects these packets to each NF to reach the maximum processing rate. For NF configurations, LightNF only installs the configurations specified by administrators to target NFs. Thus, LightNF achieves accurate analysis since its profiling is specific to the worst-case settings.

In particular, we observe that the latency of an NF on a single device (a programmable switch or a server) follows intrinsic Gaussian distribution under fixed experimental settings, as empirically demonstrated in [61]. To justify our observation, we deploy seven NFs, including Firewall, LB, NAT, Router, TM, IDS, and VPN on our testbed. The details of NF settings and our testbed are illustrated in §4.1. We measure the per-packet processing latency of each NF on

Algorithm 1 Performance analysis of NF latency.

Input: threshold ε
Output: the mean latency μ, the latency variance σ
Variables: the i-th mean latency μ_i, the i-th latency variance σ_i, the set $\mathcal{S}$ of measured latency statistics.

```
1: function MEASURE_NF_LATENCY(ε)
2:     μ_{i-1} ← 0; σ_{i-1} ← 0; S ← ∅                ▷ Initialization
3:     while True do
4:         x_i ← Test(); Add x_i to S;
5:         μ_i ← Mean(S); σ_i ← Variance(S);
6:         if |μ_i − μ_{i-1}| < ε and |σ_i − σ_{i-1}| < ε then
7:             Return μ_i, σ_i;
8:         else
9:             μ_{i-1} ← μ_i; σ_{i-1} ← σ_i;
10:        end if
11:    end while
12: end function
```

both a programmable switch and a commodity server for 1000 times. In particular, we leverage in-switch timestamps to measure the latency of NFs running on programmable switches. More precisely, when a packet arrives a switch, we set the switch to record its current system time (in nanoseconds) in a 48-bit metadata field, i.e., the timestamp t_{in}. When the switch completes its packet processing, it records the current time as t_{out} and calculates the per-packet processing latency as $t_{out} - t_{in}$. It piggybacks the latency data on the per-packet header space so that we can directly extract the data from each packet. We plot the latency distributions of Firewall, LB, and NAT in Figure 4. The results indicate that the latency of these NFs exactly fits Gaussian distribution. Moreover, we use Shapiro-Wilk test [62] to validate the normality of measured data. Take Firewall as an example. Only 0.2% and 0.1% points deviate from the mean by more than three times standard deviation on the programmable switch and the commodity server, respectively. Also, as shown in Figure 5, the p-values produced by Shapiro-Wilk test are higher than 0.05 in all cases. Thus, the null hypothesis that the NF latency follows Gaussian distribution is accepted. Note that Guassian distribution can take negative values of latency. However, this case happens with a near-zero probability so that LightNF chooses to elide this situation.

According to the above observation, we design LightNF to estimate the mean and variance of Gaussian distribution of NF latency as analysis results. By the law of large numbers, LightNF repeats measurements until the mean and variance of processing latency converge. As shown in Algorithm 1, for each NF, LightNF repeatedly stress-tests it and measures the latency under fixed settings. It records the latency for the i-th run as x_i (line 4). After each experiment, it calculates existing mean μ_i and variance σ_i (line 5). It examines two absolute differences $|\mu_i - \mu_{i-1}|$ and $|\sigma_i - \sigma_{i-1}|$. When both differences are less than a small threshold $\varepsilon > 0$, the profiling terminates and returns both μ_i and σ_i (lines 6-7). Otherwise, LightNF updates μ_{i-1} and σ_{i-1} and continues profiling (line 9).

With the mean μ and variance σ as the analysis results of NF latency, LightNF can select NFs to be offloaded with the objective of minimizing the per-packet processing latency of the entire SFC. We discuss how LightNF leverages this type of NF properties in its placement in §3.4.

Throughput profiling. In addition to the measurement of NF latency, LightNF also profiles NF throughput. It allocates the required amount of resources (e.g., CPU cores), which is obtained during resource analysis (§3.3.2), to the target NF. Then it injects the workload comprising 1024-byte packets to the NF at 40 Gbps, and obtains the average throughput (in pps) after 1000 runs. In particular, we use the average throughput as the analysis result. This is because NF throughput is much more stable than latency statistics under fixed experimental settings.

Algorithm 2 Calculate maximum stage number.

Input: set of NF dependencies $R_E(n)$
Output: stage number $L(n)$
Variable: linked list $c(u)$, height h

```
1: function CALC_STAGE(R_E(n))
2:     for (u, v) ∈ R_E(n) do              ▷ Create dependency trees
3:         Add v to c(u)
4:     end for
5:     L(n) ← 0
6:     for (u, v) ∈ R_E(n) do      ▷ Obtain max height among all trees
7:         h ← Max_Height(u, c(u))
8:         L(n) ← Max(L(n), h)
9:     end for
10:    Return L(n)
11: end function
```

3.3.4 Dependency Analysis

The SFC analyzer inspects NF dependencies via code analysis. Previous studies have identified three types of operations that can alter NF dependencies [1, 40]: (1) read-based operations, (2) write-based operations, and (3) delete-based operations. The three types of operations form three types of dependencies: (1) *Read-after-Write dependency*. NF u writes/deletes a packet field that a subsequent NF v reads; (2) *Write-after-Write dependency*. Two NFs u and v write/delete the same packet field; (3) *Write-after-Read dependency*. NF u reads a packet field that a subsequent NF v writes/deletes. Note that we do not consider the *successor* dependency in [40] because it exhibits the same pattern as the *Read-after-Write* dependency.

Inspecting NF dependencies. Specifically, the SFC analyzer enumerates every pair of NFs. For the pair (u, v), it examines every pair of operations used by the two NFs. If two operations exhibit one of the above dependencies, it regards u and v are interdependent. After enumerating all pairs of NFs, it records identified NF dependencies in a dependency matrix D: $D(u, v) = 1$ if v depends on u; $D(u, v) = 0$ otherwise. D will be used by the placement in §3.4 to preserve NF dependencies within a single device.

Measuring switch stage consumption. When placing interdependent NFs on a programmable switch, these NFs must be separated in different switch stages due to the switch restriction [58]. Thus, it is essential to guarantee that the number of stages occupied by interdependent NFs should not exceed the total number of switch stages. However, given a set of NF dependencies $R_E(n)$, how to determine the number of stages required to preserve NF dependencies is complex and uncertain.

To this end, the SFC analyzer offers Algorithm 2 to calculate the number $L(n)$ of switch stages occupied by interdependent NFs running on a programmable switch n. It takes a set of NF dependencies $R_E(n)$ as input. Here, $R_E(n)$ is a subset of dependency matrix D (see Equation 14), which records the dependencies of NFs running on n. First of all, for an arbitrary NFs u recorded in $R_E(n)$, the SFC analyzer

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2022.3149817, IEEE Transactions on Cloud Computing

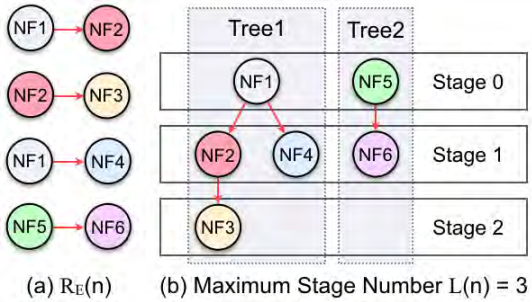


Fig. 6: Calculating maximum stage number $L(n)$.

creates a *dependency tree* and represents the tree in a linked list $c(u)$ (lines 1-3). Specifically, a dependency tree is a tree-like data structure, where the NF u is the root node of the tree, and all the subsequent NFs that depend on u are the child nodes of u. In particular, the nodes (i.e., NFs) located in different levels of a dependency tree must be placed in different switch stages with respect to switch restrictions. Next, for each dependency tree $c(u)$, the SFC analyzer obtains its maximum height using depth-first search (line 7). After all, since each tree level needs an individual switch stage, the maximum tree height among all dependency trees (denoted by $L(n)$) corresponds to the number of switch stages occupied by the NF dependencies in $R_E(n)$.

We illustrate the above algorithm via an example shown in Figure 6(a). The input $R_E(n)$ records four NF dependencies. For each NF, the SFC analyzer creates a corresponding dependency tree. It enumerates each tree to acquire its height. Figure 6(b) plots the two tallest dependency trees, *Tree1* and *Tree2*, in this example. The height of *Tree1* is 3, which is larger than that of *Tree2*. Thus, the maximum number of stages used to maintain NF dependencies is 3.

3.4 Optimization Framework

LightNF offers an optimization framework that performs the performance-optimal SFC placement based on the analysis results of NF properties offered by the SFC analyzer.

Problem statement of SFC placement. Given an SFC, we aim to place the SFC on the substrate network by offloading some NFs to programmable switches while deploying other NFs to servers. Our goal is to maximize SFC performance. The input of this problem of SFC placement includes a network comprising a set N^P of programmable switches and a set N^S of servers, a matrix B of link bandwidth, a matrix M of link latency, an SFC, and a dependency matrix D. Here, B and M are obtained by periodically measuring the latency and bandwidth of the shortest path (i.e., the path with the minimal number of hops) between each pair of devices. The output is a set of binary decision variables, $\{x_n^u\}$, indicating the mapping between NFs and network devices: $x_n^u = 1$ if the NF u is deployed on the device n; $x_n^u = 0$ otherwise. We summarize our notations in Table 3.

Solution. We design the optimization framework to formulate and solve the above problem.

① Novelty. *Unlike previous works, LightNF leverages its analysis results to make the performance-optimal offloading decisions.* Specifically, its analysis results guide SFC placement in three aspects. First, the suitability analysis classifies NFs into two sets, i.e., R_O for offloadable NFs and R_U for remaining NFs. The performance analysis characterizes NF performance behaviors, including mean μ^P and standard deviation σ^P of latency, and throughput ϕ^P in the programmable switch, as well as mean μ^S and standard deviation σ^S of latency, and throughput ϕ^S in the server. Intuitively, the above two types of analysis suggest offloading NFs with the maximum performance gain. Second, the resource analysis quantifies the NF demands for device resources. It restricts NF offloading based on the capacity of device resources. Finally, the impact of dependency analysis can be classified into intra-device dependencies and inter-device dependencies. Intra-device dependencies impose constraints on the arrangement of NFs in a single device, while inter-device dependencies determine how the traffic is directed among devices.

TABLE 3: Notation of symbols used in this paper.

Symbol	Description
N^P	Set of programmable switches.
N^S	Set of commodity servers.
$B(n,m)$	Maximum bandwidth between two devices n, m.
$M(n,m)$	Link latency between two devices n, m.
$P(n)$	Resource capacity of a programmable switch n.
$S(n)$	Resource capacity of a commodity server n.
R_O	Set of offloadable NFs.
R_U	Set of unoffloadable NFs.
$R(u)$	Resource requirement of an NF u.
$\mu(u)$	Mean of latency of NF u.
$\sigma(u)$	Standard deviation of latency of NF u.
$\phi(u)$	Throughput of NF u.
T_P	Sum of processing latency of the NFs running on switches.
T_S	Sum of processing latency of the NFs running on servers.
T_L	Sum of link latency between devices.
Φ_P	Minimal throughput of NF u on the switch.
Φ_S	Minimal throughput of NF u on the server.
Φ_B	Minimal bandwidth of links between used devices.
$D(u,v)$	Variable indicating whether two NFs are interdependent.
$R_E(n)$	Set of intra-device NF dependencies within a switch.
$L(n)$	Number of stages used by intra-device NF dependencies.
x_n^u	Variable indicating if an NF is deployed on a device.

② Overview. The optimization framework executes a three-step procedure. First, it sets the objective to maximize SFC performance. Second, it formulates the analysis results of LightNF as three types of constraints to restrict NF offloading. Finally, it inputs the objective and constraints to Gurobi [63], an integer programming solver, for problem solving.

③ Objective. The optimization objective can be either (1) minimizing latency, or (2) maximizing throughput. For minimizing latency, we consider (1) the sum of total latency in all programmable switches T_P, (2) that in all commodity servers T_S, and (3) the sum of link latency between devices T_L. Thus:

$$\min\ (T_P + T_S + T_L) \tag{1}$$

$$T_P = \sum_{n \in N^P} \sum_{u \in R_O \cup R_U} (x_n^u \cdot (\mu^P(u) + \beta \times \sigma^P(u))) \tag{2}$$

$$T_S = \sum_{n \in N^S} \sum_{u \in R_O \cup R_U} (x_n^u \cdot (\mu^S(u) + \beta \times \sigma^S(u))) \tag{3}$$

$$T_L = \sum_{n,m \in N^S \cup N^P} \sum_{u,v \in R_O \cup R_U} (x_n^u \cdot x_m^v \cdot M(n,m)) \tag{4}$$

Here, u and v are two adjacent NFs in the SFC, and u is executed before v. Our framework takes the latency variance into account by incorporating both the mean μ and variance

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2022.3149817, IEEE Transactions on Cloud Computing

σ in the calculation of T_P and T_S. Specifically, we calculate the processing latency of an NF as $\mu + \beta \times \sigma$, where β is a tunable parameter assigned by administrators. Recall that the latency of an NF follows Gaussian distribution as §3.3.3. According to the properties of Gaussian distributions, we can set β to 0 or 3: when $\beta = 0$, the latency is the average latency μ, while $\beta = 3$ implies the tail NF latency.

Moreover, for maximizing throughput, we abstract the SFC as an end-to-end system, which throughput is determined by the NF or the link with the *minimal* throughput. Thus, we consider (1) the minimal throughput of the NFs deployed on programmable switches Φ_P, (2) that of the NFs deployed on servers Φ_S, and (3) the minimal bandwidth among mapped links Φ_B. We have:

$$\max\left(\min\left(\Phi_P, \Phi_S, \Phi_B\right)\right) \tag{5}$$

$$\Phi_P = \min_{n \in N^P, u \in R_O} \left(x_n^u \cdot \phi^P(u)\right) \tag{6}$$

$$\Phi_S = \min_{n \in N^S, u \in R_O \cup R_U} \left(x_n^u \cdot \phi^S(u)\right) \tag{7}$$

$$\Phi_B = \min_{n,m \in N^S \cup N^P, u,v \in R_O \cup R_U} \left(x_n^u \cdot x_m^v \cdot B(n,m)\right) \tag{8}$$

$$\Phi_P \cdot \Phi_S \cdot \Phi_B > 0 \tag{9}$$

④ Suitability constraints. The suitability analysis derives two constraints. First, an arbitrary NF u can only be assigned to one target device $n \in N^P \cup N^S$, i.e.

$$\sum_{n \in N^P \cup N^S} x_n^u = 1,\ \forall u \in R_O \cup R_U \tag{10}$$

Second, if an NF v is unoffloadable (i.e., $v \in R_U$), it cannot be deployed on any programmable switch $n \in N^P$. Therefore:

$$x_n^v = 0,\ \forall v \in R_U, \forall n \in N^P \tag{11}$$

⑤ Per-device resource constraints. NFs in a device should not consume more resources than the resource capacity of the device. Here, we separately consider the resources of programmable switches and that of commodity servers. For a programmable switch $n \in N^P$, we have:

$$\sum_{u \in R_O} \left(x_n^u \cdot R(u)\right) \le P(n),\ \forall n \in N^P \tag{12}$$

where $R(u)$ represents the requirements of TCAM, SRAM, action units, and PHVs. Next, for a server $n \in N^S$, we have:

$$\sum_{u \in R_O \cup R_U} \left(x_n^u \cdot R(u)\right) \le S(n),\ \forall n \in N^S \tag{13}$$

where $R(u)$ represents the requirements of RAM and cores.
⑥ Intra-device dependencies. The number of stages is limited in a programmable switch. Thus, the number of stages $L(n)$ occupied by interdependent NFs running on a programmable switch n should be less than the maximum number of switch stages $P^{ST}(n)$:

$$R_E(n) = \left\{(u,v) | x_n^u \cdot x_n^v = 1, D(u,v) = 1\right\}, \quad \forall u, v \in R_O, \forall n \in N^P \tag{14}$$

$$L(n) = Calc\_Stage(R_E(n)), \forall n \in N^P \tag{15}$$

$$L(n) \le P^{ST}(n),\ \forall n \in N^P \tag{16}$$

Here, $L(n)$ is determined by the set $R_E(n)$ of NF dependencies among the NFs running on a switch n. We input $R_E(n)$ to Algorithm 2 to calculate $L(n)$.

3.5 SFC Deployment and Runtime Management

LightNF adopts existing approaches to perform the remaining steps of SFC deployment and runtime management.

NF deployment. After solving the problem of SFC placement, LightNF obtains the set of binary decision variables, $\{x_n^u\}$, from its optimization framework. Recall that if $x_n^u = 1$, LightNF needs to deploy the NF u on the device n. Thus, given an SFC to be deployed, LightNF enumerates every NF defined in the SFC. For the NF u, if $x_n^u = 1$, LightNF deploys u on the target device n. If n is a programmable switch, LightNF inputs the LightNF primitives that implement u to the switch compiler. It obtains corresponding switch configurations from the switch compiler and installs these configurations on the target device n. Otherwise, n is a server so that LightNF directly runs u on n. Then it allocates the required amount of in-server resources, which is profiled by the SFC analyzer (§3.3.2), to u.

Moreover, LightNF needs to configure the switch parser to support NF processing. Specifically, since switches forbid the processing of packet payloads (see Section 3.3.1), the NFs offloading to switches only process a limited number of packet headers varying from the layer-2 headers such as Ethernet to the layer-4 headers such as TCP. Thus, LightNF configures each switch with a default parser that is capable of handling all types of layer 2-4 headers. More precisely, the parser is a finite state machine, in which each state represents the logic of parsing a specific packet header (e.g., IPv4) while each transition between two states denotes the dependency between two headers (e.g., the transition "Ethernet→IPv4" is activated iff the value of EtherType equals 0x0800). It is pre-defined with all the states processing layer 2-4 headers and corresponding state transitions. By this means, LightNF does not need to change the switch parser as the NF types change.

Traffic routing. After NF deployment, the interdependent NFs defined in the SFC may be placed on different network devices. Thus, LightNF needs to correctly route traffic between interdependent NFs at runtime. Otherwise, inter-device NF dependencies will be violated. To this end, we design LightNF to generate the routing rules that correctly forward traffic with respect to NF dependencies. Specifically, if two NFs, u and v, are deployed on different devices but have a dependency (indicated by the LightNF primitive, `Connect`(u, v)), LightNF will first locate the target devices, n and m, that run u and v, respectively. Next, it finds out the network paths connecting n and m from the network topology. Then for each device located in these paths, it generates a routing rule that correctly routes traffic from n to m, and populates this rule to the device. As a result, the traffic will be correctly routed from u to v at runtime, which guarantees inter-device NF dependencies.

Incremental SFC deployment. LightNF applies two strategies to support incremental SFC deployment at runtime. (1) When administrators need to deploy a new SFC at runtime,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2022.3149817, IEEE Transactions on Cloud Computing

LightNF re-runs its optimization framework to place the SFC on residual network resources. To avoid disturbing existing placement, it prioritizes the devices that have not run any NFs yet to deploy the new SFC. (2) Administrators often need to deploy several SFCs at the same time to improve resource usage [7]. To this end, LightNF adopts P4SC [7] to merge SFCs iteratively by eliminating redundant operations among SFCs. Specifically, given a set $\mathcal{A}$ of n input SFCs, LightNF executes $n-1$ iterations. In each iteration, it picks up two SFCs from the input set. Then it inputs the two SFCs to P4SC. P4SC merges the two SFCs by merging redundant NF operations among the two SFCs while observing NF dependencies. Next, LightNF obtains a merged SFC from P4SC and adds the SFC to $\mathcal{A}$. Thus, after $n-1$ iterations, there is only one SFC left in $\mathcal{A}$, which is the resulting merged SFC. Finally, LightNF places the merged SFC on the network using its first strategy.

4 Evaluation

In this section, we conduct extensive experiments to evaluate LightNF. We repeat each experiment for 100 times and take the average. Our experimental results indicate that:

- LightNF achieves higher scalability than Metron [35] by fulfilling all SFC placement requests (Exp#1).
- For real SFCs, LightNF reduces SFC latency by one order of magnitude compared to E2 [33] (Exp#2).
- For complex SFCs, LightNF reduces SFC latency by two orders of magnitude, and achieves 9.5× higher throughput than E2 and Metron (Exp#3).
- In testbed, LightNF reduces SFC latency by around 93% and achieves 176% throughput increase than E2 (Exp#4).
- Even for the complex SFC with 100 NFs, LightNF can solve the SFC placement problem within 60 ms (Exp#5).

4.1 Setup

Prototype. We implement the LightNF primitives that build NFs in both P4 [26] (if offloadable) and DPDK 17.11 [55]. We realize other primitives that build SFCs in C++. We also implement a primitive compiler, an SFC analyzer, and an optimization framework in Python. The compiler parses input primitives and delivers parsed results to the SFC analyzer, which performs subsequent analysis on parsed results. Next, the optimization framework uses Gurobi [63] to execute SFC placement. Then LightNF identifies which NFs are offloaded and connects the P4 source codes of these NFs to generate the switch configuration. Currently, LightNF installs the switch configuration on PISA-based switches [40]. For other NFs, it assembles the DPDK codes of these NFs to the server configuration. Also, we use LightNF primitives to build seven NFs, including Firewall (FW), load balancer (LB), network address translator (NAT), Router, traffic monitor (TM), IDS, and VPN. Table 4 shows the details of these NFs. We configure these NFs as suggested by [1], and use these NFs to compose the real SFCs [1, 7, 33] in Table 5.

Testbed. We build a testbed with a 32×100 Gbps Barefoot Tofino switch [10] and four servers. Each server has twelve 2.3 GHz CPU cores, and directly connects to the switch via a 40 Gbps link. We run LightNF on an individual server that controls the testbed. To evaluate LightNF, we generate a 40-Gbps traffic workload based on a CAIDA trace [64]. We also implement a simulator in C++ to evaluate LightNF at scale.

TABLE 4: NFs used by our experiments. "Suitability" indicates whether an NF is suitable for offloading while "# of primitives" indicates the number of LightNF primitives used to implement the NF.

Name	Match	Operations	Suitability	# of primitives
FW	5-tuple	Drop malicious packets	✓	3
LB	5-tuple	Hash 5-tuple; forward packets	✓	5
NAT	5-tuple	Modify IP addresses	✓	4
Router	srcIP	Forward packets	✓	3
TM	5-tuple	Hash 5-tuple; report	✓	4
IDS	Payload	Drop packets	✗	5
VPN	5-tuple	AES-based encryption	✗	6

TABLE 5: SFCs used by our experiments.

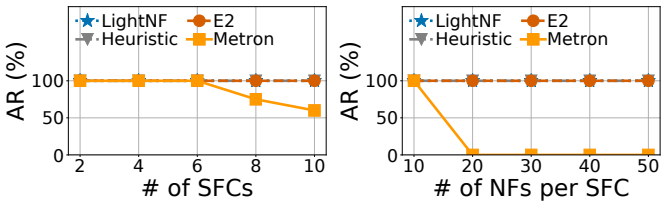
Name	LightNF primitives for building SFCs
SFC1 [7]	`Connect`(FW, LB)
SFC2 [7]	`Connect`(IDS, FW); `Connect`(FW, NAT); `Connect`(NAT, Router)
SFC3 [1]	`Connect`(VPN, TM); `Connect`(TM, FW); `Connect`(FW, LB)
SFC4 [1]	`Connect`(IDS, TM); `Connect`(TM, LB);
SFC5 [33]	`Connect`(NAT, FW); `Connect`(FW, IDS); `Connect`(FW, VPN); `Connect`(IDS, VPN)

Comparison solutions. We compare LightNF with two representative state-of-the-art solutions, including a software-based method, E2 [33], and a hardware-assisted method, Metron [35]. E2 deploys SFCs on multi-core servers with the objective of minimizing inter-server transfers. It assigns each NF to a specific CPU core and uses an additional core for traffic dispatching and steering among NFs. We base E2 on DPDK. Moreover, Metron selects a programmable switch and a server from the network to deploy an SFC. Specifically, its selection comprises two steps. First, for unoffloadable NFs, it randomly selects two servers and determines if none of them has enough resources to deploy unoffloadable NFs. If so, it abandons the two servers and repeats the process until a server is found. Otherwise, it chooses the server with more resources. For the chosen server, it executes all the unoffloadable NFs within a single core to avoid inter-core packet transferring. Second, it deploys offloadable NFs on the programmable switch that is closest to the selected server. Next, we implement a heuristic that randomly selects programmable switches for NF offloading. Thus, we can demonstrate the benefits of considering NF properties when making offloading decisions in LightNF by comparing it with the heuristic.

4.2 Experimental Results

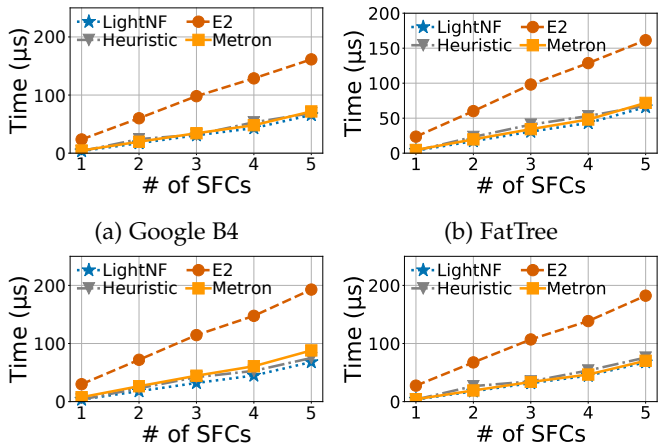
(Exp#1) Scalability of LightNF. We evaluate the scalability of LightNF, in terms of the ability of simultaneously deploying multiple SFCs. We measure the SFC acceptance rate (AR), which is the ratio of the number of accepted SFCs to that of total SFCs. Here, the accepted SFCs are the SFCs that are successfully deployed. We simulate a 4-ary FatTree data center network to deploy SFCs. We randomly set half of the devices as programmable switches, while using the remaining devices as servers [42]. The link latency is uniformly distributed from 0.1 μs to 0.5 μs. We set the resource capacity of devices based on real settings [40, 58]: each switch has 46.25 MB SRAM, 10 MB TCAM, 512 B PHV, 6400 action units, and 32 stages, while each server has 12 CPU cores and 128 GB RAM. Moreover, the resource usage of an NF is uniformly distributed. Specifically, an NF

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2022.3149817, IEEE Transactions on Cloud Computing



(a) Normal SFCs (b) Complex SFCs

Fig. 7: (Exp#1) Scalability of LightNF.



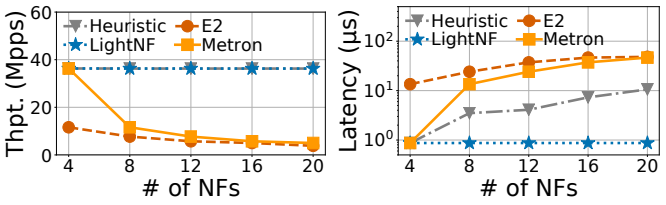
(a) Google B4 (b) FatTree

(c) Internet2 (d) Stanford

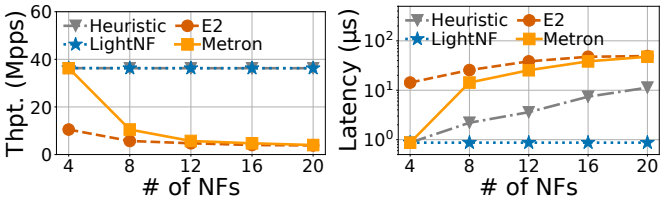
Fig. 8: (Exp#2) Performance benefits for real SFCs.

consumes 6∼12 MB TCAM and SRAM, 12∼24 MB RAM, 1000∼1500 action units, 10∼50 B PHV and 1∼5 CPU cores. We generate two types of synthetic SFCs: (1) normal SFCs, each of which has ten NFs; (2) complex SFCs with massive NFs. For normal SFCs, we vary the number of SFCs to be deployed from 2 to 10. For complex SFCs, we deploy an SFC at a time but vary the number of NFs resided in an SFC from 10 to 50. In each SFC, we set half of NFs to offloadable NFs. In Figure 7, we see that both E2 and LightNF accept all SFCs in all cases. In contrast, Metron supports at most six normal SFCs to be simultaneously deployed, and fails to deploy complex SFCs with more than 20 NFs. This is because Metron does not take per-device resource constraints into consideration. For example, for the complex SFC with twenty NFs, Metron tries to find a switch that is capable of maintaining ten offloadable NFs, which is infeasible given limited switch resources.

(Exp#2) Performance benefits for real SFCs. We evaluate the performance benefits incurred by LightNF for real SFCs. We simulate four large-scale topologies, Google B4 [65], 4-ary FatTree, Internet2 [66], and Stanford campus network [67], to deploy SFCs. We consider the five SFCs in Table 5 and use the same settings as Exp#1. We incrementally deploy these SFCs and direct the traffic workload to go through every SFC. As shown in Figure 8, LightNF achieves the optimal placement that minimizes SFC latency. Compared to E2, LightNF reduces the latency by orders of magnitude. The improvement comes from the line-rate capability of programmable switches in which a large portion of NFs is offloaded. Moreover, other solutions achieve higher performance than E2 via NF offloading. However, due to their random selection mechanism, they suffer from a latency overhead of up to 23 μs compared to LightNF,



(a) Throughput in FatTree (b) Latency in FatTree



(c) Throughput in Internet2 (d) Latency in Internet2

Fig. 9: (Exp#3) Performance benefits for complex SFCs.

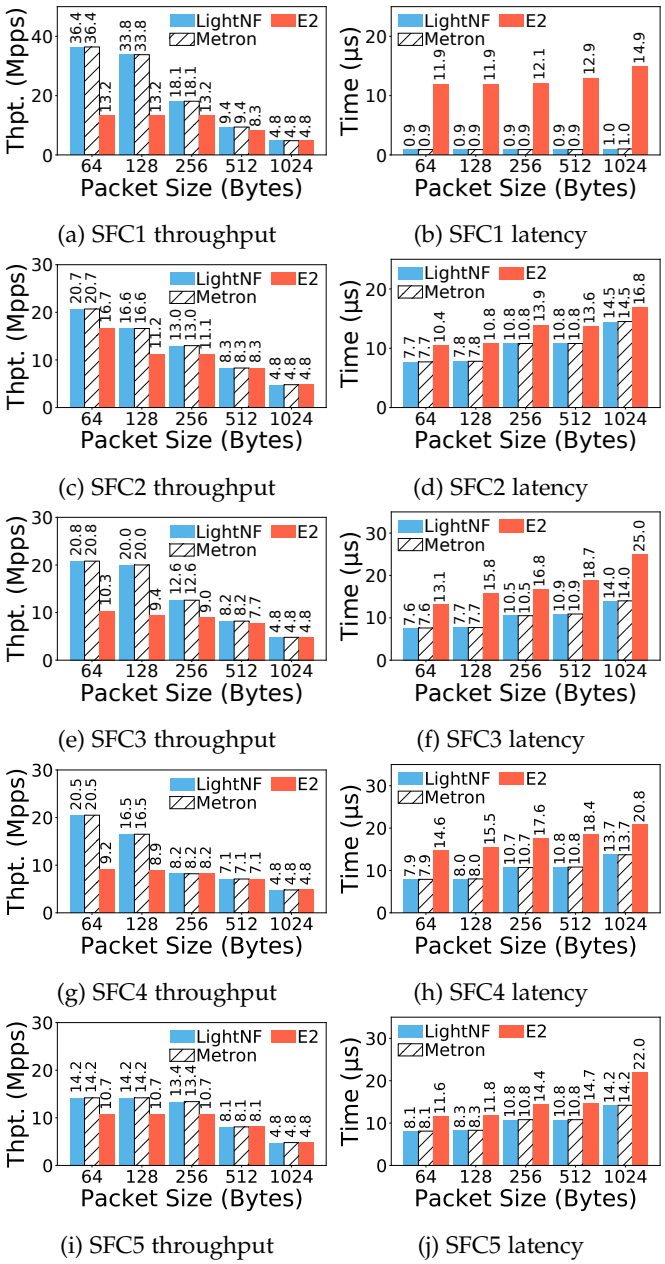
which inevitably affects latency-sensitive applications.

(Exp#3) Performance benefits for complex SFCs. We evaluate the performance benefits incurred by LightNF for complex SFCs. We generate five SFCs by varying the number of NFs from four to twenty while randomly setting the number of edges between NFs. We inject the traffic workload to each SFC and steer it to travel all the NFs in the SFC. We compare LightNF with E2, an enhanced version of Metron that can use multiple servers for SFC deployment, and the random heuristic. We deploy SFCs on the 4-ary FatTree and Internet2 [66], respectively. We apply the same settings as Exp#1. Figure 9 shows that compared to other solutions, LightNF reduces SFC latency by two orders of magnitude, and increases throughput by up to 9.5×. Note that Metron dramatically decreases SFC performance when the number of NFs increases. This is because Metron overlooks per-device resource constraints, leading to inefficient placement for complex SFCs. Moreover, the random heuristic randomly selects switches for NF offloading, inevitably leading to sub-optimal decisions compared to LightNF.

(Exp#4) Testbed performance. We deploy real SFCs on our testbed via LightNF and measure SFC performance. Figure 10 shows that LightNF achieves at most 176% throughput increase than E2. The throughput gain is small for large packets because both approaches have reach link capacity. In particular, as shown in Figure 10(b), LightNF reduces the latency in E2 by around 93%. This is because all the NFs in SFC1 are offloadable. Thus, LightNF deploys the entire SFC on the programmable switch, leading to remarkably low latency. Note that Metron achieves similar results compared to LightNF. This is because the number of devices in our testbed is limited, such that the placement offered by Metron is the same as that of LightNF. Due to the same reason, the heuristic makes the same decisions as LightNF, which is elided here.

(Exp#5) Execution time of LightNF. We measure the execution time of LightNF. We first measure the time of dissecting NF properties at scale. We generate artificial NFs by randomly combining the operations used by real NFs. Each NF stores 10^4 rules and requires 0.32 MB RAM. We manually generate the LightNF primitives that construct SFCs based on artificial NFs. The SFCs contain various

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2022.3149817, IEEE Transactions on Cloud Computing



(a) SFC1 throughput (b) SFC1 latency

(c) SFC2 throughput (d) SFC2 latency

(e) SFC3 throughput (f) SFC3 latency

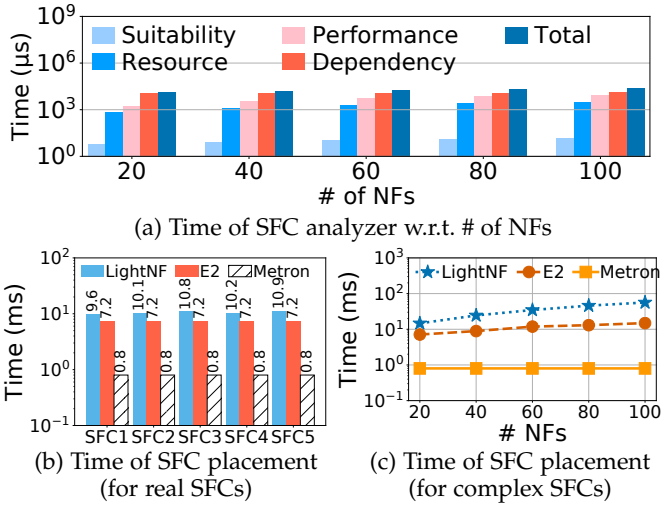
(g) SFC4 throughput (h) SFC4 latency

(i) SFC5 throughput (j) SFC5 latency

Fig. 10: (Exp#4) Testbed performance.

number of NFs ranging from 20 to 100. Figure 11(a) shows that the SFC analyzer takes around 23.5 ms to analyze 100 NFs. It is acceptable due to two reasons. First, the SFC analyzer performs its tasks in offline mode. Second, the properties of an NF are reusable. Thus, the SFC analyzer can omit the analysis of an NF if this NF has been dissected, which decreases analysis time. We also evaluate the time of SFC analyzer for the real SFCs used in our experiments. Our results indicate that the analysis completes within 20 ms in all cases, which is acceptable.

Next, we evaluate the time of optimization framework for real SFCs and complex SFCs, respectively. We apply the same settings as Exp#1. We compare our framework with E2 and Metron. As shown in Figure 11(b)-(c), our framework takes less than 60 ms, which is a bit slower than E2. However, recall that a real SFC generally uses less than 20 NFs. Thus, for real SFCs, the time of our framework is a few milliseconds. Since our framework is invoked offline, its



(a) Time of SFC analyzer w.r.t. # of NFs

(b) Time of SFC placement (for real SFCs)

(c) Time of SFC placement (for complex SFCs)

Fig. 11: (Exp#5) Execution time of LightNF.

execution time is acceptable compared to the minute-level device configuration time. Note that Metron spends less than 1 ms across all cases because its simple heuristic only selects a switch and a server to deploy an SFC. Although its time remains stable and small, its scalability is limited as indicated by Exp#1.

5 Discussion

Integration of sketch algorithms. To date, many solutions have leveraged various hardware-compatible sketch algorithms to realize their desired NFs of on-demand traffic measurement such as heavy hitter detection entirely in the data plane. However, the current list of LightNF primitives (see Table 2) does not contain such algorithms. In response, we plan to integrate existing sketch algorithms into LightNF in the near future. More precisely, we plan to implement each sketch algorithm as an individual NF operation and register it as a specific primitive supported by LightNF by means of the approaches mentioned in Section 3.2. In this way, administrators can directly invoke the LightNF primitives corresponding to their desired types of sketch algorithms to compose their NFs.

Analysis of stateful NFs. Recall from Section 3.3.1 that existing NFs may adopt stateful computation that requires to store the historical packet processing information. However, modern programmable switches only provide limited support of such stateful processing due to their limited capacity of resources [68, 69, 70]. In this context, our current design of LightNF choose to entirely deploy the NFs that involve stateful operations to commodity servers. It guarantees that all the NFs can be successfully deployed but leaves room for further performance optimization. In the future, we plan to enhance LightNF with the suitability of analyzing stateful NFs to address this limitation.

Overhead analysis. We analyze three-fold overheads of the deployment made by LightNF. (1) *Per-packet byte overhead*. Some solutions [71, 72, 73] piggyback essential packet processing results such as packet-in timestamps on the per-packet header space so as to deliver information among distributed NFs. This incurs the per-packet byte overhead, which reduces the available size of payload in each

packet and decreases end-to-end performance [29]. However, LightNF does not add information to packet headers. It preserves inter-device NF dependencies by correctly routing packets among devices, avoiding the per-packet byte overhead. (2) *Switch resource consumption*. LightNF aims to make the performance-optimal decisions in NF offloading. Its deployment does not add additional logic to switches. Thus, it does not incur switch resource consumption. (3) *Network-wide overhead*. LightNF populates additional routing rules to network devices so as to correctly deliver packets between the devices running NFs. Nevertheless, its strategy of static routing only adds 2-3 rules to each device, making network-wide overheads small and acceptable.

Failure recovery. Recall from Section 3.5 that our current design of LightNF routes traffic among different devices with pre-defined routing rules to preserve inter-device NF dependencies. However, such static routing fails to mitigate the impact of network failures. In response, we plan to enhance LightNF with fault-tolerant traffic routing. More precisely, we plan to integrate existing solutions of in-network failure detection and dynamic traffic rerouting [73, 74] into LightNF.

6 Conclusion

With the flexibility of programmable switches, recent researches enable network administrators to offload NFs from commodity servers to programmable switches. However, to achieve the performance-optimal SFC placement, administrators still have to manually analyze the source codes of NFs to understand various NF properties, which imposes significant user burdens. To address this problem, we propose LightNF, a novel NFV system that performs the comprehensive analysis of NF properties, and leverages the analysis results to achieve performance-optimal SFC placement. We have implemented LightNF on Tofino-based hardware programmable switches. Extensive experiments show that LightNF outperforms existing solutions with orders-of-magnitude latency reduction, up to 9.5× throughput increase, and higher scalability in SFC placement.

Acknowledgement

We sincerely thank our editors and reviewers for their constructive and insightful comments. This work is supported by the National Key R&D Program of China (2020YFB1804705), the Key-Area Research and Development Program of Guangdong Province (2020B0101390001), the National Natural Science Foundation of China (61802365, 61902362), the Joint Funds of the National Natural Science Foundation of China (U20A20179), and the Key R&D Program of Zhejiang Province (2021C01036).

Reference

[1] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu, "Nfp: Enabling network function parallelism in nfv," in *ACM SIGCOMM*, 2017, pp. 43–56.
[2] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *ACM SIGCOMM*, 2017, pp. 15–28.
[3] X. Jin *et al.*, "Netcache: Balancing key-value stores with fast in-network caching," in *ACM SOSP*, 2017, pp. 121–136.
[4] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, "In-network computation is a dumb idea whose time has come," in *HotNet*, 2017, pp. 150–156.
[5] X. Jin, X. Li *et al.*, "Netchain: Scale-free sub-rtt coordination," in *USENIX NSDI*, 2018, pp. 35–49.
[6] D. Sanvito, G. Siracusano, and R. Bifulco, "Can the network be the ai accelerator?" in *Proceedings of the Morning Workshop on In-Network Computing*, 2018, pp. 20–25.
[7] X. Chen, D. Zhang, X. Wang, K. Zhu, and H. Zhou, "P4sc: Towards high-performance service function chain implementation on the p4-capable device," in *IEEE/IFIP IM*, 2019, pp. 1–9.
[8] D. Wu, A. Chen, T. E. Ng, G. Wang, and H. Wang, "Accelerated service chaining on a single switch asic," in *HotNet*. ACM, 2019.
[9] K. Zhang, D. Zhuo, and A. Krishnamurthy, "Gallium: Automated software middlebox offloading to programmable switches," in *ACM SIGCOMM*, 2020, pp. 283–295.
[10] Tofino-based hardware programmable switches. "https://www.barefootnetworks.com/technology/#tofino".
[11] X. Chen, Q. Huang, P. Wang, M. Zili, H. Liu, Y. Chen, D. Zhang, H. Zhou, Z. Boyang, and C. Wu, "Lightnf: Simplifying network function offloading in programmable networks," in *IEEE/ACM IWQoS*, 2021, pp. 1–10.
[12] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: network processing as a cloud service," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 13–24, 2012.
[13] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "Opennf: Enabling innovation in network function control," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 163–174.
[14] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *USENIX NSDI*. USENIX, 2014, pp. 459–473.
[15] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen, "Clicknp: Highly flexible and high performance network processing with reconfigurable hardware," in *ACM SIGCOMM*, 2016, pp. 1–14.
[16] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "Softnic: A software nic to augment hardware," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-155*, 2015.
[17] Y. Le, H. Chang, S. Mukherjee, L. Wang, A. Akella, M. M. Swift, and T. Lakshman, "Uno: uniflying host and smart nic offload for flexible packet processing," in *ACM SOCC*, 2017, pp. 506–519.
[18] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, and S. Seshan, "Tea: Enabling state-intensive network functions on programmable switches," in *ACM SIGCOMM*, 2020, pp. 90–106.
[19] M. Zhang, G. Li, S. Wang, C. Liu, A. Chen, H. Hu, G. Gu, Q. Li, M. Xu, and J. Wu, "Poseidon: Mitigating volumetric ddos attacks with programmable switches," in *NDSS*, 2020.
[20] Z. Liu, H. Namkung, G. Nikolaidis *et al.*, "Jaqen: A high-performance switch-native approach for detecting and mitigating volumetric ddos attacks with programmable switches," in *USENIX Security*, 2021.
[21] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith, "Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow," in *USENIX ATC*, 2018, pp. 823–835.
[22] J. Sonchack *et al.*, "Turboflow: Information rich flow record generation on commodity switches," in *ACM EuroSys*, 2018, p. 11.
[23] Y. Zhou, Z. Xi, D. Zhang, Y. Wang, J. Wang, M. Xu, and J. Wu, "Hypertester: high-performance network testing driven by programmable switches," in *ACM CoNEXT*, 2019, pp. 30–43.
[24] J. Gao, E. Zhai, H. H. Liu *et al.*, "Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics," in *ACM SIGCOMM*, 2020, pp. 435–450.
[25] T. Pan, N. Yu, C. Jia *et al.*, "Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches," in *ACM SIGCOMM*, 2021, pp. 194–206.
[26] P. Bosshart, D. Daly, G. Gibb *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.
[27] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in *ACM SOSR*, 2016, pp. 1–12.
[28] Z. Yu, Y. Zhang, V. Braverman, M. Chowdhury, and X. Jin, "Netlock: Fast, centralized lock management using programmable

switches," in *ACM SIGCOMM*, 2020, pp. 126–138.
[29] R. Ben Basat *et al.*, "Pint: Probabilistic in-band network telemetry," in *ACM SIGCOMM*, 2020, pp. 662–680.
[30] T. Yang *et al.*, "Elastic sketch: Adaptive and fast network-wide measurements," in *ACM SIGCOMM*, 2018, pp. 561–575.
[31] Q. Huang, P. P. Lee, and Y. Bao, "Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference," in *ACM SIGCOMM*, 2018, pp. 576–590.
[32] Q. Huang, S. Sheng, X. Chen *et al.*, "Toward nearly-zero-error sketching via compressive sensing," in *USENIX NSDI*, 2021, pp. 1027–1044.
[33] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: a framework for nfv applications," in *ACM SOSP*, 2015, pp. 121–136.
[34] R. Gandhi *et al.*, "Duet: Cloud scale load balancing with hardware and software," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 27–38, 2015.
[35] G. P. Katsikas, T. Barbette, D. Kostic, R. Steinert, and G. Q. Maguire Jr, "Metron: Nfv service chains at the true speed of the underlying hardware," in *USENIX NSDI*, 2018, pp. 171–186.
[36] B. Turkovic, F. Kuipers *et al.*, "Fast network congestion detection and avoidance using p4," in *NEAT*, 2018, pp. 45–51.
[37] Onos project: P4 brigade. "https://wiki.onosproject.org/display/ONOS/P4+brigade".
[38] K. Qian, S. Ma, M. Miao, J. Lu, T. Zhang, P. Wang, C. Sun, and F. Ren, "Flexgate: High-performance heterogeneous gateway in data centers," in *APNet*, 2019, pp. 36–42.
[39] T. Jepsen, D. Alvarez, N. Foster, C. Kim, J. Lee, M. Moshref, and R. Soulé, "Fast string searching on pisa," in *ACM SOSR*, 2019, pp. 21–28.
[40] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown *et al.*, "Forwarding metamorphosis: fast programmable match-action processing in hardware for sdn," in *ACM SIGCOMM*, 2013, pp. 99–110.
[41] P. Zheng, A. Narayanan, and Z.-L. Zhang, "A closer look at nfv execution models," in *Proceedings of the Asia-Pacific Workshop on Networking (APNet)*, 2019, pp. 85–91.
[42] X. Chen, H. Liu *et al.*, "Speed: Resource-efficient and high-performance deployment for data plane programs," in *IEEE ICNP*, 2020, pp. 1–12.
[43] M. He, A. Basta, A. Blenk, N. Deric, and W. Kellerer, "P4nfv: An nfv architecture with flexible data plane reconfiguration," in *CNSM*, 2018, pp. 90–98.
[44] A. Stockmayer, S. Hinselmann, M. Häberle, and M. Menth, "Service function chaining based on segment routing using p4 and sr-iov (p4-sfc)," in *International Conference on High Performance Computing*. Springer, 2020, pp. 297–309.
[45] C. Zhang, J. Bi, Y. Zhou, and J. Wu, "Hypervdp: High-performance virtualization of the programmable data plane," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 556–569, 2019.
[46] Y. Li, L. T. X. Phan, and B. T. Loo, "Network functions virtualization with soft real-time guarantees," in *IEEE INFOCOM*, 2016, pp. 1–9.
[47] D. Li, P. Hong, K. Xue *et al.*, "Virtual network function placement considering resource optimization and sfc requests in cloud datacenter," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 7, pp. 1664–1677, 2018.
[48] R. Cohen, L. Lewin-Eytan, J. S. Naor, and D. Raz, "Near optimal placement of virtual network functions," in *IEEE INFOCOM*, 2015, pp. 1346–1354.
[49] H. Moens and F. De Turck, "Customizable function chains: Managing service chain variability in hybrid nfv networks," *IEEE Transactions on Network and Service Management*, vol. 13, no. 4, pp. 711–724, 2016.
[50] F. Bari, S. R. Chowdhury, R. Ahmed, R. Boutaba, and O. C. M. B. Duarte, "Orchestrating virtualized network functions," *IEEE Transactions on Network and Service Management*, vol. 13, no. 4, pp. 725–739, 2016.
[51] M. Xia, M. Shirazipour, Y. Zhang, H. Green, and A. Takacs, "Network function placement for nfv chaining in packet/optical datacenters," *Journal of Lightwave Technology*, vol. 33, no. 8, pp. 1565–1570, 2015.
[52] R. Mijumbi, S. Hasija, S. Davy, A. Davy, B. Jennings, and R. Boutaba, "Topology-aware prediction of virtual network function resource requirements," *IEEE Transactions on Network and Service Management*, vol. 14, no. 1, pp. 106–120, 2017.
[53] J. Hwang, K. K. Ramakrishnan, and T. Wood, "Netvm: high performance and flexible networking using virtualization on commodity platforms," *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, pp. 34–47, 2015.
[54] A. Bremler-Barr, Y. Harchol, and D. Hay, "Openbox: a software-defined framework for developing, deploying, and managing network functions," in *ACM SIGCOMM*, 2016, pp. 511–524.
[55] Intel. Data Plane Development Kit. "http://dpdk.org".
[56] PF_RING. "http://www.ntop.org/products/packet-capture/pf_ring".
[57] L. De Carli, R. Sommer, and S. Jha, "Beyond pattern matching: A concurrency model for stateful deep packet inspection," in *SIGSAC*. ACM, 2014, pp. 1378–1390.
[58] L. Jose, L. Yan, G. Varghese *et al.*, "Compiling packet programs to reconfigurable switches." in *USENIX NSDI*, 2015, pp. 103–115.
[59] Linux command: pmap. "https://linux.die.net/man/1/pmap".
[60] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat, "Chronos: Predictable low latency for data center applications," in *SOCC*, 2012, pp. 1–14.
[61] A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, and R. Ricci, "Taming performance variability," in *USENIX OSDI*, 2018, pp. 409–425.
[62] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, no. 3/4, pp. 591–611, 1965.
[63] Gurobi optimizer. "http://www.gurobi.com".
[64] Caida traces. "http://www.caida.org/data/overview/".
[65] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined wan," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 3–14.
[66] Internet2. "https://www.internet2.edu/media/medialibrary/2018/07/16/Internet2-Network-Infrastructure-Topology-Layers-23.pdf".
[67] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. M. Parulkar, "Can the production network be the testbed?" in *USENIX OSDI*, vol. 10, 2010, pp. 1–6.
[68] V. Sivaraman, S. Narayana, O. Rottenstreich *et al.*, "Heavy-hitter detection entirely in the data plane," in *ACM SOSR*, 2017, pp. 164–176.
[69] A. Gupta, R. Harrison, M. Canini *et al.*, "Sonata: Query-driven streaming network telemetry," in *ACM SIGCOMM*, 2018, pp. 357–371.
[70] N. Gebara, A. Lerner, M. Yang, M. Yu, P. Costa, and M. Ghobadi, "Challenging the stateless quo of programmable switches," in *ACM HotNets*, 2020, pp. 153–159.
[71] S. Luo, H. Yu, and L. Vanbever, "Swing state: Consistent updates for stateful and programmable data planes," in *ACM SOSR*, 2017, pp. 115–121.
[72] Q. Huang, H. Sun, P. P. Lee, W. Bai, F. Zhu, and Y. Bao, "Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy," in *ACM SIGCOMM*, 2020, pp. 404–421.
[73] D. Kim *et al.*, "Redplane: enabling fault-tolerant stateful in-switch applications," in *ACM SIGCOMM*, 2021, pp. 223–244.
[74] M. Chiesa, R. Sedar, G. Antichi, M. Borokhovich, A. Kamisiński, G. Nikolaidis, and S. Schmid, "Fast reroute on programmable switches," *IEEE/ACM Transactions on Networking*, vol. 29, no. 2, pp. 637–650, 2021.

Xiang Chen received the B.Eng. and the M.Eng. degrees from Fuzhou University in 2019 and 2022, respectively. He is currently pursuing the Ph.D. degree with the College of Computer Science and Technology, Zhejiang University, China. He has published papers in IEEE INFOCOM, IEEE ICNP and so on. He received the Best Paper Award from IEEE/ACM IWQoS 2021, and the Best Paper Candidate from IEEE INFOCOM 2021. His research interests include programmable networks and network security.

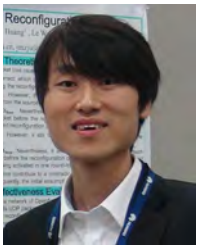
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2022.3149817, IEEE Transactions on Cloud Computing

Dong Zhang received the B.S. and Ph.D. degrees from Zhejiang University, China, in 2005 and 2010, respectively. He visited Alabama University, USA as a visiting scholar from 2018 to 2019. He is currently a professor of the College of Computer Science and Big Data at Fuzhou University, China. His research areas include software defined networking, network virtualization and Internet QoS.

Qun Huang is an Assistant Professor at Department of Computer Science and Technology, Peking University. Before joining PKU, he worked at Institute of Computing Technology, Chinese Academy of Sciences (ICT-CAS) from Septemper 2017 to May 2020, and at Huawei from September 2015 to September 2017. He got his Ph.D. degree in August 2015 in The Chinese University of Hong Kong. He received his bachelor degree in Computer Science from Peking University in 2011.

Zili Meng received the B.Eng. degree in Department of Electronic Engineering at Tsinghua University in 2019. He is currently pursuing the Ph.D. degree with Institute for Network Science and Cyberspace, Tsinghua University. He has published papers in ACM SIGCOMM, ACM Multimedia, IEEE/ACM ToN, IEEE JSAC and so on. His research interests include learning-based networked systems and real-time communications. He was a recipient of Microsoft Research Asia Fellowship in 2020, also the winner of the Student Research Competition in ACM SIGCOMM 2018.

Hongyan Liu received the B.Eng. degree with the College of Mathematics and Computer Science, Fuzhou University. He is currently pursuing the M.Eng. degree with the College of Computer Science, Zhejiang University. His current research interests include network measurement and programmable networks.



Haifeng Zhou received the Ph.D. degree in computer science and technology from Zhejiang University in 2018. He is now an associate research fellow in College of Control Science and Engineering, Zhejiang University. His current research interests include P4 and security, software-defined network and security, cloud security, Industrial Internet and security, intelligent networks and security systems, and innovative network and security technologies.

Chunming Wu received the Ph.D. degree in computer science from Zhejiang University in 1995. He is currently a Professor with the College of Computer Science and Technology, Zhejiang University. He is also the Associate Director of the Research Institute of Computer System Architecture and Network Security, Zhejiang University, and the Director of the NGNT Laboratory. His research fields include software defined networking, reconfigurable networks, proactive network defense, network security, network virtualization, the architecture of next-generation Internet, and intelligent networks.

Xuan Liu (S'11-M'17-SM'21) received Ph.D. degree in Computer Science and Engineering at Southeast University, China, and is currently a lecturer with College of Information Engineering (College of Artificial Intelligence) at Yangzhou University, China, and on-the-job postdoctor with School of Computer Science and Engineering, Southeast University, China. He is also serving as an Editorial Board Member of Computer Communications, an Associate Editor of TELS, IET Smart Cities, IJCA and IJIN, an Area Editor of EAI TIoT. Furthermore, he served(s) as a TPC Member of ACM MobiCom workshop, IEEE INFOCOM workshop, IEEE ICC, IEEE GlobeCom, IEEE WCNC, IFIP/IEEE IM, IEEE NOMS, IEEE PIMRC, IEEE MSN, IEEE VTC, IEEE ICIN, IEEE GIIS, IEEE DASC, APNOMS, AdHoc-Now, FNC, CollaborateCom, and ChinaCom, etc. Besides, he served as a Reviewer for 200+ reputable conferences/journal papers. His main research interests focus on ubiquitous collaborative networking.

Qiang Yang received Ph.D. degree in Electronic Engineering and Computer Science from Queen Mary, University of London, London, U.K., in 2007 and worked in the Department of Electrical and Electronic Engineering at Imperial College London, U.K., from 2007 to 2010. He visited University of British Columbia and University of Victoria Canada as a visiting scholar in 2015 and 2016. He is currently a full Professor at College of Electrical Engineering, Zhejiang University, China, and has published more than 200 technical papers, applied 60 national patents, co-authored 2 books, edited 2 books and several book chapters. His research interests over the years include smart energy systems, large-scale complex network modelling, control and optimization, learning based optimization and control. He is the Follow of British Computer Society (BCS), Senior Member of IEEE, IET and the Senior Member of China Computer Federation (CCF).