LightNF: Simplifying Network Function Offloading in Programmable Networks

Xiang Chen[1 2 3], Qun Huang[1], Peiqiao Wang[3], Zili Meng[4], Hongyan Liu[5], Yuxin Chen[6],
Dong Zhang[3], Haifeng Zhou[5], Boyang Zhou[7], Chunming Wu[5]
[1]Peking University [2]Pengcheng Lab [3]Fuzhou University [4]Tsinghua University
[5]Zhejiang University [6]University of Science and Technology of China [7]Zhejiang Lab

***Abstract*—In network function virtualization (NFV), network functions (NFs) are chained as a service function chain (SFC) to enhance NF management with high flexibility. Recent solutions indicate that the processing performance of SFCs can be significantly improved by offloading NFs to programmable switches. However, such offloading requires a deep understanding of NF properties to achieve the maximum SFC performance, which brings non-trivial burdens to network administrators. In this paper, we propose LightNF, a novel system that simplifies NF offloading in programmable networks. LightNF automatically dissects comprehensive NF properties (e.g., NF performance behaviors) via code analysis and performance profiling while eliminating manual efforts. It then leverages the analyzed NF properties in its SFC placement so as to produce the performance-optimal offloading. We have implemented a LightNF prototype. 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.**

I. Introduction

In network function virtualization (NFV), network administrators often deploy a sequential chain of network functions (NFs), commonly referred as a *service function chain* (SFC), to provide network services. Compared to 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 performance, offloading could bring significant SFC performance improvement.

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 this goal. Specifically, administrators face four questions when offloading NFs based on their own experiences.

- **Suitability**. Given an arbitrary NF, whether the operations of the NF can be realized in a programmable switch or not?
- **Resource consumption**. Given an arbitrary NF, how many memory and computational resources does the NF need?
- **Performance behaviors**. Given an arbitrary NF, how much performance benefits can offloading the NF to the programmable switch bring?
- **NF dependencies**. How to preserve the inherent execution dependencies between NFs when placing NFs across programmable switches and commodity servers?

To answer these questions, administrators have to conduct a comprehensive analysis on NF properties, which is extremely time-consuming and laborious. Even worse, the analysis results are heterogeneous. The integration of these heterogeneous results in SFC placement remains complicated and challenging for administrators to make offloading decisions. In a nutshell, simplifying the analysis and integration of NF properties is still a challenging yet critical issue for practical NF offloading. To the best of our knowledge, none of existing solutions are capable of addressing the above problem.

In this paper, we propose LightNF, a novel system that alleviates the burdens of NF offloading. LightNF provides a suite of primitives for the description of NFs and SFCs. It further employs a *LightNF analyzer* that dissects NF properties that are indicated by the primitives. Specifically, it (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, it builds an optimization framework that formulates SFC placement with NF properties as constraints. The framework determines how to place the SFC on a hybrid network with several programmable switches and servers while maximizing SFC performance.

We have implemented a LightNF prototype that supports Tofino switches [10]. We conduct extensive experiments to evaluate LightNF. The results show that LightNF offers two orders of magnitude latency reduction and 9.5× throughput improvement compared to state-of-the-art solutions.

II. Background and Motivation

This paper targets a hybrid network that comprises several programmable switches and servers. Programmable switches allow administrators to customize packet processing logics with domain-specific languages such as P4 [11]. Recent studies [2, 3, 7] have demonstrated the feasibility of realizing NFs

Qun Huang is the corresponding author.

978-0-7381-3207-5/21/$31.00

on programmable switches. With the high-throughput (several Tbps) and low-latency ($<1\mu$s) processing capability of switch ASICs, offloading NFs onto programmable switches improves the performance of traditional software-based NFs by several orders of magnitude [7]. However, we cannot offload all NFs of SFCs due to the limited capacity of switch resources. Thus, our goal is to find out the *performance-optimal placement* that places each NF on either a programmable switch or a server to maximize SFC performance.

Challenges. However, it is non-trivial to make 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 so as 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 [12]. 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 [12]. 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 [12], 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 [13, 7] 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 [14], 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.

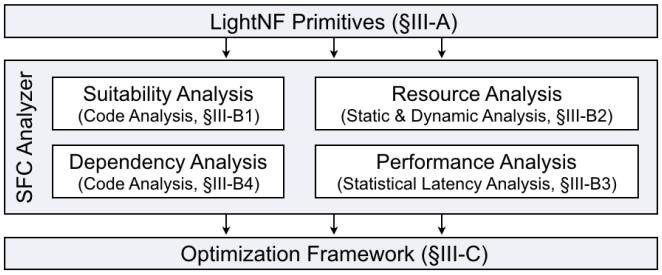


Fig. 1: Overview of LightNF.

(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.

III. LightNF Design

To address these challenges, we propose LightNF, a system that simplifies NF offloading on programmable switches. As shown in Figure 1, LightNF offers a set of intent-based primitives for administrators to describe NFs and SFCs in a high-level manner (§III-A). Next, an SFC analyzer is invoked to dissect the properties of the NFs realized by LightNF primitives (§III-B). By combining both static code analysis and dynamic performance profiling, it infers four types of NF properties, including (1) NF suitability, (2) NF resource consumption, (3) NF performance behaviors, and (4) NF dependencies. Such analysis addresses the challenge of dissecting NF properties in §II. Finally, LightNF formulates an optimization framework to make the best offloading decisions (§III-C). The framework collectively takes the analysis results (i.e., all types of NF properties) into account, thereby addressing the challenge of property integration. After acquiring the offloading decisions, LightNF deploys NFs on the substrate network (§III-D).

A. LightNF Primitives

LightNF provides three classes of intent-based primitives.

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

Name	O?	Description
Atomic NF operations		
`add_header`	✓	Add a header (e.g., VLAN tag) to the packet.
`alert`	✓	Raise an alert with the flow ID (e.g., 5-tuple).
`broadcast`	✓	Broadcast a packet to multiple ports.
`copy`	✓	Copy a packet.
`count`	✓	Count packets based on flow IDs.
`decrypt`	✗	Decrypt a packet based on AES.
`dequeue`	✗	Pop a packet from a queue.
`drop`	✓	Drop packets.
`encrypt`	✗	Encrypt a packet based on AES.
`enqueue`	✗	Send a packet to a queue.
`header_classify`	✓	Classify packets based on header fields.
`modify_field`	✓	Modify a packet field.
`modify_payload`	✗	Modify the packet payload.
`output`	✓	Send a packet to a destination port.
`payload_classify`	✗	Classify incoming packets based on payloads.
`rate_limit`	✗	Limit traffic rate.
`read_state`	✗	Read the state associated with a flow.
`recirculate`	✓	Recirculate a packet.
`remove_header`	✓	Remove a specific header from the packet.
`update_state`	✗	Update the state associated with a flow.
Primitive for building NFs (u, v are NF operations)		
`Connect`(u, c, v)	-	Execute v if the execution result of u is c (c can be `hit`, or `miss`, or `all`)
Primitive for building SFCs (u, v are NFs)		
`Connect`(u, v)	-	Create an edge from u to v

Atomic NF operations. We observe that even though NFs are diverse, they use many atomic operations such as matching headers and forwarding packets [15]. Thus, we design a set of atomic NF operations, as shown in Table I, that enables administrators to compose NFs on their demands.

Primitive for building NFs. We design a primitive to assemble atomic operations into NFs. The primitive, `Connect`(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 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 2(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 2(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.

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 them in §III-B.

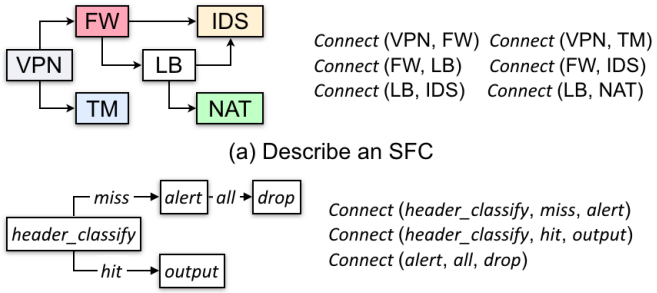


Fig. 2: Describing an SFC with LightNF primitives.

B. SFC Analyzer

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

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. If an NF contains any restricted operations, the NF is unsuitable to be offloaded. Currently, LightNF identifies the following types of restricted operations.

- **Buffering.** 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 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.
- **Accessing packet payload.** Some NFs read or write 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 privacy. Since programmable switches restrict the number of header bytes for each packet, the payload-oriented operations, including `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

TABLE II: Commonly-used NFs and their suitability.

Network Function	Suitable?	Unsuitable Reason
Caching	✓[3]	-
Congestion control	✓[13]	-
Congestion detection	✓[16]	-
Coordination	✓[5]	-
Firewall, ACL	✓[7]	-
Key-value store	✓[3]	-
L2/L3 switch, router	✓[7]	-
Load balancer	✓[2]	-
NAT, CGNAT, NAT64	✓[7]	-
QoS mapping	✓[17]	-
Traffic monitor	✓[7]	-
Traffic shaper	✗	Buffering
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

packet counts. However, programmable switches adopt a pipeline-based paradigm to achieve line-rate packet processing [12]. The paradigm only supports one-shot packet processing, which restricts the operations for stateful computation, including `read_state` and `write_state`.

The results of suitability analysis identify all the offloadable NFs. We deploy a subset of them to programmable switches to meet the resource and performance requirements. The remaining NFs are deployed on servers. Note that our suitability analysis can be applied to *arbitrary* NFs as long as these NFs are implemented with our primitives.

Findings. We analyze the suitability of commonly-used NFs, as shown in Table II, and summarize two findings as follows. First, a substantial portion (>50%) of NFs are offloadable. Second, the number of processed flows significantly varies across NFs. For instance, some NFs such as traffic monitor are hot-spot and have rules matching all incoming flows to provide fundamental services, while some NFs like packet compression only serve a small portion of traffic with a few rules. We plan to enhance LightNF to detect and prioritize hot NFs in SFC placement in the future.

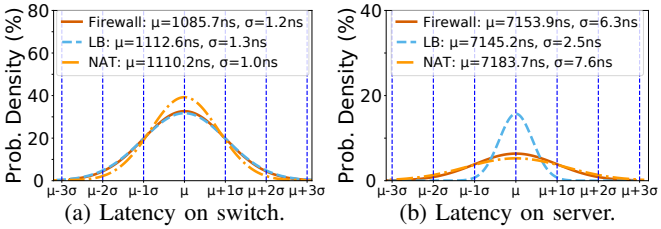
2) Resource Analysis: The second type of analysis infers the resource usage of each NF. Specifically, each NF contains two properties specified by administrators, i.e., the maximum number of rules, and a set of NF operations. Thus, LightNF considers two types of resources, including memory resources 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 [13] that executes all NFs within a single core, and (2) the pipeline model [18, 19] that assigns an NF to at least one CPU core. Since the pipeline model offers more flexibility in NF management than RTC [14], we design LightNF to analyze in-server NFs under the pipeline model.

- **In-switch resources**. LightNF targets PISA [12], 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 stores packet headers and metadatas 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 number of CPU cores.

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 since the resource usage remains stable after compilation. According to compilation results, it obtains the usage of SRAM, TCAM, action units and PHVs. For in-server resources, LightNF employs a dynamic method as the usage varies at runtime. It monitors the maximum memory usage of each NF during its execution via system tools, e.g., `pmap` [20]. For CPU cores, LightNF incrementally assigns cores to an NF until NF throughput reaches the maximum. In this way, LightNF obtains the desired number of CPU cores for each NF.

3) Performance Analysis: The third analysis profiles in-switch and in-server performance behaviors of each NF. It mainly focuses on accurately and timely profiling the tail latency of each NF. The measured results are then provisioned to the optimization framework for SFC placement. 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 fixes experimental settings when measuring NF latency. For traffic pattern, LightNF synthesizes a benchmark workload. The packet size of the benchmark workload is 1500 bytes, which is widely used for SFC benchmarks [1]. Then LightNF injects the workload to each NF to reach its maximum processing rate. For NF configurations, LightNF only installs the configurations specified by administrators to target NFs. Thus, LightNF achieves accurate analysis since its latency profiling is quite specific to fixed settings. 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 [21].

Observation. 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 [22]. We choose three NFs,



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

Fig. 3: Distribution of NF latency.

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: $\mu_{i-1} \leftarrow 0$; $\sigma_{i-1} \leftarrow 0$; $\mathcal{S} \leftarrow \emptyset$ ▷ Initialization
3: **while** True **do**
4: $x_i \leftarrow$ Test(); Add x_i to $\mathcal{S}$;
5: $\mu_i \leftarrow$ Mean($\mathcal{S}$); $\sigma_i \leftarrow$ Variance($\mathcal{S}$);
6: **if** $|\mu_i - \mu_{i-1}| < \varepsilon$ and $|\sigma_i - \sigma_{i-1}| < \varepsilon$ **then**
7: Return μ_i, σ_i;
8: **else**
9: $\mu_{i-1} \leftarrow \mu_i$; $\sigma_{i-1} \leftarrow \sigma_i$;
10: **end if**
11: **end while**
12: **end function**

Firewall, LB, and NAT, and measure their latency on both a programmable switch and a commodity server for 1000 times. We plot the latency distributions in Figure 3. The results indicate that the latency of these NFs exactly fits Gaussian distribution. To justify our observation, we use Shapiro-Wilk test [23] 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. Moreover, the p-values produced by Shapiro-Wilk test are 0.69 and 0.21, which are higher than 0.05. Thus, the null hypothesis that the NF latency follows Gaussian distribution is accepted. Note that (1) the results of other NFs follow the same trend, which are elided here due to space limitation; (2) 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.

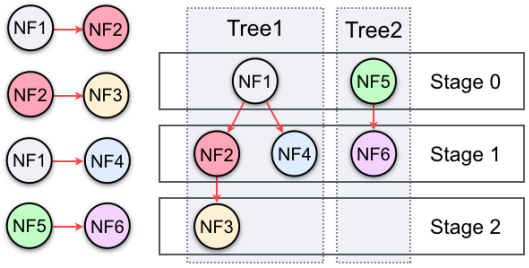
Parameter estimation. According to our observation, we estimate the mean and variance of Gaussian distribution of NF latency. By the law of large numbers, LightNF repeats measurements until the mean and variance of processing latency converge. As shown in Algorithm 1, for each NF, LightNF repeatedly 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).

Throughput profiling. LightNF also profiles NF throughput. It allocates the required amount of resources (e.g., CPU cores),

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** $(u, v) \in R_E(n)$ **do** ▷ Create dependency trees
3: Add v to $c(u)$
4: **end for**
5: $L(n) \leftarrow 0$
6: **for** $(u, v) \in R_E(n)$ **do** ▷ Obtain maximum height among all trees
7: $h \leftarrow$ Max_Height$(u, c(u))$
8: $L(n) \leftarrow$ Max$(L(n), h)$
9: **end for**
10: Return $L(n)$
11: **end function**



(a) $R_E(n)$ (b) Maximum Stage Number L(n) = 3

Fig. 4: Calculating maximum stage number $L(n)$.

which is obtained via resource analysis, 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 1 K runs. Note that we use the average throughput as the result since NF throughput is much more stable than latency statistics under fixed experimental settings.

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, 12]: (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 [12] because it exhibits the same pattern as the *Read-after-Write* dependency. For any pair of two NFs (u, v), the SFC analyzer examines every pair of operations in them. 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 §III-C.

Moreover, when placing interdependent NFs on a programmable switch, these NFs must be separated in different switch stages due to the switch restriction [24]. 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.

Algorithm. 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 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 $R_E(n)$.

Example. As shown in Figure 4(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 4(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.

C. Optimization Framework

Problem statement. 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 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 III.

TABLE III: Notation.

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 in a programmable switch.
$L(n)$	Number of stages occupied by intra-device NF dependencies.
x_n^u	Decision variable indicating if an NF is deployed on a device.

Solution. We design an optimization framework in LightNF to solve the problem of offloading NFs.

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.

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 [25], 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 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 §III-B3. 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:

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

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

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

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

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

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:

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

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

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

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 $P^{ST}(n)$:

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

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

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

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 $L(n)$.

D. Deployment

Steering network traffic. After problem solving, LightNF deploys NFs on network devices. However, interdependent NFs in an SFC may be placed on different devices, which violates inter-device NF dependencies. In response, LightNF installs routing rules to direct traffic among devices. Specifically, if two NFs are deployed on different devices but have dependencies, LightNF will install routing rules to the devices resided in the paths connecting the two devices.

Scale to multiple SFCs. Administrators often need to deploy multiple SFCs to improve resource usage [7]. To this end, LightNF applies two strategies. First, it adopts incremental deployment. When a new SFC arrives, it re-runs the optimization framework to place the SFC on residual resources. Second, for the SFCs needed to be simultaneously deployed, it adopts P4SC [7] to merge SFCs by eliminating redundancy among SFCs, and places the merged SFC on the network.

IV. 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:

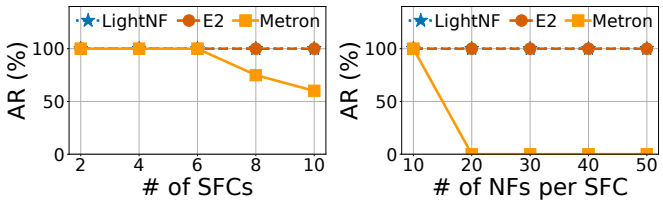
- LightNF achieves higher scalability than Metron [13] by fulfilling all SFC placement requests (Exp#1).
- For real SFCs, LightNF reduces SFC latency by one order of magnitude compared to E2 [18] (Exp#2).
- For complex SFCs, LightNF reduces SFC latency by two orders of magnitude, and achieves 9.5× higher 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).

A. Setup

Prototype. We implement the LightNF primitives that build NFs in both P4 [11] (if offloadable) and DPDK 17.11 [26]. 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 [25] 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 [12]. 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,

TABLE IV: SFCs used by our experiments.

Name	LightNF Primitives for Building SFC
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 [18]	`Connect`(NAT, FW); `Connect`(FW, IDS); `Connect`(FW, VPN); `Connect`(IDS, VPN)



(a) Normal SFCs (b) Complex SFCs

Fig. 5: (Exp#1) Scalability of LightNF.

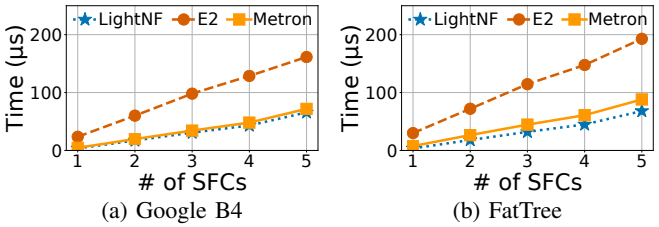
and VPN. We configure these NFs as suggested by [1], and use these NFs to compose the real SFCs [7, 1, 18] in Table IV.

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 traffic workload based on a CAIDA trace [27]. We also implement a simulator in C++ to evaluate LightNF at scale.

Comparison solutions. We compare LightNF with two representative state-of-the-art solutions, including a software-based method, E2 [18], and a hardware-assisted method, Metron [13]. 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.

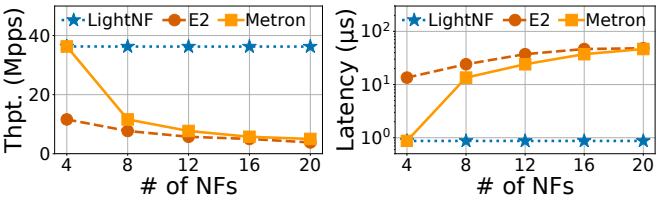
B. Experimental Results

(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 remaining devices as servers [28]. 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 [12, 24]:



(a) Google B4 (b) FatTree

Fig. 6: (Exp#2) Performance benefits for real SFCs.



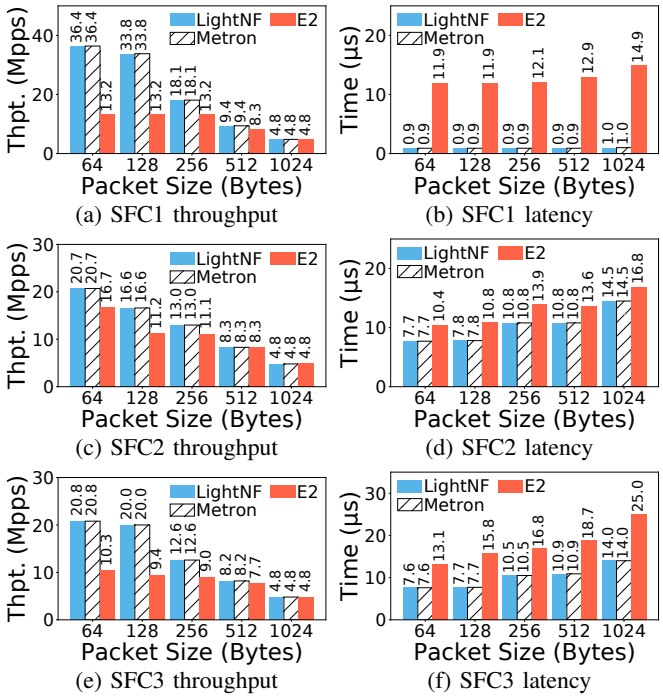
(a) Throughput (b) Latency

Fig. 7: (Exp#3) Performance benefits for complex SFCs.

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 consumes 6$\sim$12 MB TCAM and SRAM, 12$\sim$24 MB RAM, 1000$\sim$1500 action units, 10$\sim$50 B PHV and 1$\sim$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 5, 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 two topologies, Google B4 [29] and 4-ary FatTree, to deploy SFCs. We consider the five SFCs in Table IV 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 6, 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, alike LightNF, Metron achieves higher performance than E2 via NF offloading. However, due to its random selection mechanism, it suffers from a latency overhead of up to 20 μs compared to LightNF, which reduces quality of services of latency-sensitive applications.

(Exp#3) Performance benefits for complex SFCs. We evaluate the performance benefits incurred by LightNF for



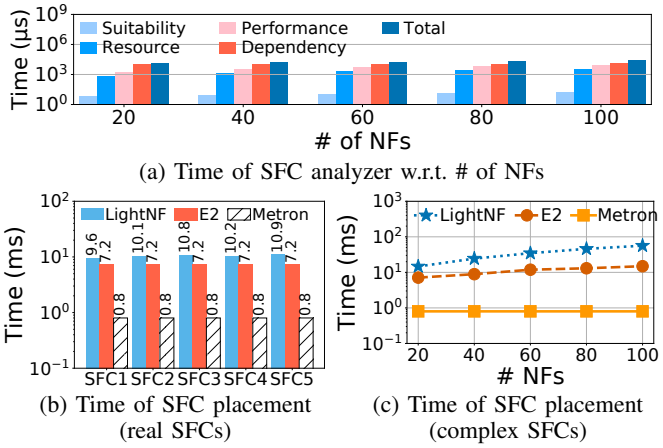
(a) SFC1 throughput (b) SFC1 latency
(c) SFC2 throughput (d) SFC2 latency
(e) SFC3 throughput (f) SFC3 latency

Fig. 8: (Exp#4) Testbed performance.

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 and an enhanced version of Metron that can use multiple servers for SFC deployment. We apply the same settings as Exp#1. Figure 7 shows that compared to E2 and Metron, 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.

(Exp#4) Testbed performance. We deploy real SFCs on our testbed via LightNF and measure SFC performance. Figure 8 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 8(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 space limitation, we elide the results of remaining SFCs, which follow the same trend.

(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



(a) Time of SFC analyzer w.r.t. # of NFs
(b) Time of SFC placement (real SFCs)
(c) Time of SFC placement (complex SFCs)

Fig. 9: (Exp#5) Execution time of LightNF.

the LightNF primitives that construct SFCs based on artificial NFs. The SFCs contain various number of NFs ranging from 20 to 100. Figure 9(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 9(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 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.

Summary. LightNF conducts timely analysis of NF properties and leverages its results in its SFC placement. Thus, it not only simplifies NF offloading, but also outperforms state-of-the-art solutions with higher SFC performance and scalability.

V. Related Work

Offloading NFs to programmable switches. Recent research efforts offload various 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 performance of NFs. However, these efforts focus on offloading a specific NF rather than SFCs. Moreover, 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. Metron [13] 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. These solutions lack of a deep analysis of NF properties, leading to failed or sub-optimal SFC placement. Instead, LightNF analyzes comprehensive NF properties and leverages NF properties to make the performance-optimal offloading decisions. Thus, its SFC placement outperforms the above solutions with higher performance and scalability, as indicated by our experiments.

SFC acceleration. Some solutions leverage packet delivery acceleration [19], NF modularization [15], and NF parallelism [1] to accelerate SFCs while high-performance network I/O engines such as DPDK [26] boost up SFC performance via kernel bypassing. However, these studies are still based on software, thus suffering from limited CPU processing capability and highly variable latency. In contrast, LightNF offers the performance-optimal NF offloading on programmable switches, which brings significant performance benefits.

SFC placement. Existing solutions present trade-offs between SFC performance and available resources in SFC placement [18, 30, 31]. However, they deploy SFCs on servers, while offloading NFs to programmable switches remains non-trivial. In contrast, LightNF provides an in-depth analysis of comprehensive NF properties while eliminating manual efforts to reduce user burdens. It formulates its analysis results as constraints to address challenges and achieve the maximum processing performance in SFC placement.

VI. Conclusion

We propose LightNF, an NFV system that performs comprehensive analysis of NF properties, and leverages the analysis results to achieve performance-optimal SFC placement. We have implemented LightNF that supports Tofino switches. Extensive experiments show that LightNF outperforms existing solutions with orders-of-magnitude latency reduction and higher scalability. In the future, we plan to integrate LightNF with other NFV frameworks to support more types of NF operations and programmable hardware.

Acknowledgement

We thank our reviewers for their valuable and constructive 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), "FANet: PCL Future Greater-Bay Area Network Facilities for Large-scale Experiments and Applications (No. LZC0019)", 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 *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 *SIGCOMM*, 2017, pp. 15–28.

[3] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *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, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, "Netchain: Scale-free sub-rtt coordination," in *NSDI*, 2018, pp. 35–49.

[6] D. Sanvito, G. Siracusano, and R. Bifulco, "Can the network be the ai accelerator?" in *NetCompute*, 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 *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*, 2019, pp. 141–149.

[9] K. Zhang, D. Zhuo, and A. Krishnamurthy, "Gallium: Automated software middlebox offloading to programmable switches," in *SIGCOMM*, 2020, pp. 283–295.

[10] Tofino. "https://www.barefootnetworks.com/technology/#tofino".

[11] 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.

[12] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, 2013.

[13] 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 *NSDI*, 2018, pp. 171–186.

[14] P. Zheng, A. Narayanan, and Z.-L. Zhang, "A closer look at nfv execution models," in *APNet*, 2019, pp. 85–91.

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

[16] B. Turkovic, F. Kuipers, N. van Adrichem, and K. Langendoen, "Fast network congestion detection and avoidance using p4," in *NEAT*, 2018, pp. 45–51.

[17] 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.

[18] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: a framework for nfv applications," in *SOSP*, 2015, pp. 121–136.

[19] 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.

[20] Linux command: pmap. "https://linux.die.net/man/1/pmap".

[21] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat, "Chronos: Predictable low latency for data center applications," in *SOCC*, 2012, p. 9.

[22] A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, and R. Ricci, "Taming performance variability," in *OSDI*, 2018, pp. 409–425.

[23] 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.

[24] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling packet programs to reconfigurable switches." in *NSDI*, 2015, pp. 103–115.

[25] Gurobi optimizer. "http://www.gurobi.com".

[26] Intel corporation. Data Plane Development Kit. "http://dpdk.org".

[27] The caida internet traces. "http://www.caida.org/data/overview/".

[28] X. Chen, H. Liu, Q. Huang, P. Wang, D. Zhang, H. Zhou, and C. Wu, "Speed: Resource-efficient and high-performance deployment for data plane programs," in *ICNP*. IEEE, 2020, pp. 1–12.

[29] S. Jain, A. Kumar, S. Mandal *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.

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

[31] 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.