SNF: Synthesizing high performance NFV service chains

Georgios P. Katsikas[1], Marcel Enguehard[2,3], Maciej Kuźniar[1], Gerald Q. Maguire Jr.[1], and Dejan Kostić[1]

[1]**KTH Royal Institute of Technology, Kista, Sweden**
[2]**Network and Computer Science Department (INFRES), Telecom ParisTech, Paris, France**
[3]**Paris Innovation and Research Laboratory (PIRL), Cisco Systems, Paris, France**

Corresponding author:
Georgios P. Katsikas[1]

Email address: katsikas@kth.se

ABSTRACT

In this paper we introduce SNF, a framework that synthesizes (S) network function (NF) service chains by eliminating redundant I/O and repeated elements, while consolidating stateful cross layer packet operations across the chain. SNF uses graph composition and set theory to determine traffic classes handled by a service chain composed of multiple elements. It then synthesizes each traffic class using a minimal set of new elements that apply single-read-single-write and early-discard operations.
Our SNF prototype takes a baseline state-of-the-art network functions virtualization (NFV) framework to the level of performance required for practical NFV service deployments. Software-based SNF realizes long (up to 10 NFs) and stateful service chains that achieve line-rate 40 Gbps throughput (up to 8.5x greater than the baseline NFV framework). Hardware-assisted SNF, using a commodity OpenFlow switch, shows that our approach scales at 40 Gbps for Internet Service Provider-level NFV deployments.

INTRODUCTION

Middleboxes hold a prominent position in today's networks as they substantially enrich the dataplane's functionality (Sherry et al., 2012; Gember-Jacobson et al., 2014). However, to manage traditional middleboxes requires costly capital and operational expenditures; hence, network operators are adopting network functions virtualization (NFV) (European Telecommunications Standards Institute, 2012).

Among the first challenges in NFV was to scale software-based packet processing by exploiting the characteristics of modern hardware architectures. To do so, several works leveraged parallelism first across multiple servers and then across multiple cores, sockets, memory controllers, and graphical processing units (GPUs) (Han et al., 2010; Kim et al., 2015b) within a single server (Dobrescu et al., 2009, 2010).

Attaining hardware-based forwarding performance was difficult to achieve, even with highly-scalable software-based packet processing frameworks. The main reason was the poor I/O performance of these frameworks. Thus, the focus of both industry and academia shifted to customizing the operating systems (OSs) to achieve high-speed network I/O. For example, by using batch packet processing (Kim et al., 2012), static memory pre-allocation, and zero copy data transfers (Rizzo, 2012; DPDK, 2016).

Modern applications require combinations of network functions (NFs), also known as service chains, to satisfy their services' quality requirements (Quinn and Nadeau, 2015). With all the above advancements in place, NFV instances achieved line-rate forwarding at tens of millions of packets per second (Mpps); however, performance issues remain when several NFs are chained together. State-of-the-art frameworks such as ClickOS (Martins et al., 2014) and NetVM (Hwang et al., 2014) have reported substantial throughput degradation when realizing chains of interconnected, monolithic NFs.

The first consolidation attempts targeted application layer (e.g., deep packet inspection (DPI)) (Bremler-Barr et al., 2014) and session layer (e.g., HTTP) (Sekar et al., 2012) consolidation. However, a lot of redundancy still resides lower in the network stack. Anderson et al. (2012) describe how xOMB allows them to build programmable and extensible open middleboxes specialized for request/response based communication. In addition, Slick (Anwer et al., 2015) introduced a programming language to deploy network-wide service chains, driven by a controller. Slick avoids redundant operations and shares common elements; however, its decentralized consolidation still realizes a chain of NFs as distributed processes. Most recently, E2 (Palkar et al., 2015) showed how to schedule NFs across a cluster of machines for high throughput. Also, OpenBox (Bremler-Barr et al., 2016) introduced an algorithm that merges processing graphs from different NFs into a single processing graph. Contemporaneously with E2 and OpenBox, our work implements the mechanisms fully specified in (Enguehard, 2016) and represents the next logical step of high-performance NFV research *.

In the case of network-wide deployments, chains suffer from the latency imposed by interconnecting different machines, processes, and switches, along with potential virtualization overheads. In the case of single-server deployments, where the NFs are pinned to a specific (set of) core(s), throughput is bounded by the increasing number of context switches as the length of the chain increases. Based on our measurements, context switches cause a domino effect on cache utilization because of continuous data invalidations and the number of CPU cycles spent forwarding packets along the chain. This leads to increased end-to-end packet latency and considerable variation in latency (jitter).

In this paper, we describe the design and implementation of the Synthesized Network Functions (SNF), our approach for dramatically increasing the performance of NFV service chains. The idea in SNF is simple: create spatial correlation to execute service chains as close as possible to the speed of CPU cores operating on the fastest, L1 cache of modern multi-core machines. SNF leverages the ever-continuing increases in core counts of modern machines and the recent advances in user-space networking.

SNF automatically derives traffic classes of packets that are traversing a provider-specified service chain of NFs. Packets in a traffic class are all processed the same way. Additionally, SNF handles stateful NFs. Using its understanding of each of the per-traffic class chains, SNF then *synthesizes equivalent, high-performance NFs* for each of the traffic classes. In a straightforward SNF deployment, one CPU core processes one traffic class. In realistic scenarios, SNF allocates multiple CPU cores to execute different sets of traffic classes in isolation (see § 2).

SNF's optimization process performs the following tasks: (*i*) consolidates all the **read** operations of a traffic class into one element, (*ii*) early-discards those traffic classes that lead to packet drops, and (*iii*) associates each traffic class with a **write-once** element. Moreover, SNF shares elements among NFs to avoid unnecessary overhead, and compresses the number and length of the chain's traffic classes. Finally, SNF scales with an increasing number of NFs and traffic classes.

This architecture shifts the challenge to packet classification, as one component of SNF has to classify an incoming packet into one of the pre-determined traffic classes, and pass it to the synthesized function. We extended popular, open-source software to improve the performance of software-only packet classification. In addition, we employed an OpenFlow (McKeown et al., 2008) switch as a packet classifier demonstrating the performance possible by a sufficiently powerful programmable network interface (commonly abbreviated as NIC). The benefits for network operators are multifold: (*i*) SNF dramatically increases the throughput of long NF chains, and achieves low latency, and (*ii*) it does so while preserving the functionality of the original service chains.

We implemented the SNF design principles into an appropriately modified version of the Click (Kohler et al., 2000) framework. To demonstrate SNF's superior performance, we compare it against the fastest Click variant todate, called FastClick (Barbette et al., 2015). To show SNF's generality we tested its performance in three uses cases: (*i*) a chain of software routers, (*ii*) nested network address and port translators (NAPTs) (Liu et al., 2014), and (*iii*) access control lists (ACLs) using actual NF configurations taken from Internet Service Providers (ISPs) (Taylor and Turner, 2007).

Our evaluation shows that software-based SNF achieves 40 Gbps, even with small Ethernet frames, across long (up to 10 NFs), stateful chains. In particular, it achieves up to 8.5x more throughput and 10x lower latency with 2-3.5x lower latency variance than the original NF chains implemented with FastClick-when running on the same hardware. Offloading traffic classification to a commodity OpenFlow switch allows SNF to realize realistic ISP-level chains at 40 Gbps (for most of the frame sizes), while bounding the median chain latency below 100 μs (measured from separate sending and receiving machines).

In the rest of this paper, we provide an overview of SNF in § 2. We introduce our synthesis approach in § 3 and a motivating example in § 4. Implementation details and performance evaluation are presented in § 5 and § 6 respectively. We discuss verification aspects in § 7. § 8 discusses the limitations of this work and § 9 positions our work with respect to the state of the art. Finally, § 10 concludes this paper.

*We provide a detailed comparison of our work with both E2 and OpenBox in § 9.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2477v2 | CC BY 4.0 Open Access | rec: 28 Sep 2016, publ: 28 Sep 2016

SNF OVERVIEW

The idea of synthesizing network service components consorts with a powerful property: *the data correlation of network traffic*. In a network system, this property is mapped to a *spatial locality with respect to the caches*. SNF aggregates parts of the flow space into traffic class units (TCUs) (the detailed definition is in § 3.1), which are then mapped to sets of (re)write operations. By carefully setting the CPU affinity of each TCU, this aggregation enforces a large degree of correlation in the traffic requests (seen as logical units of data) resulting in high cache hit rates.

Our overarching goal is to design a system that efficiently utilizes per core and across cores cache hierarchies. With this in mind, we design SNF based on Figure 1. Let us assume that a network operator wants to deploy a service chain between network domains 1 and 2. For simplicity let us also assume that there is one NIC per domain. A set of dedicated cores (i.e., Core 1 and 2 for the NICs facing domains 1 and 2, respectively) undertakes to read and write frames at line-rate. Once a set of frames is received, say by core 1, it is transferred to the available processing cores (i.e., Cores 3 to k). Frame transfers can occur at high speed via a shared cache, which has substantial capacity in modern hardware architectures.

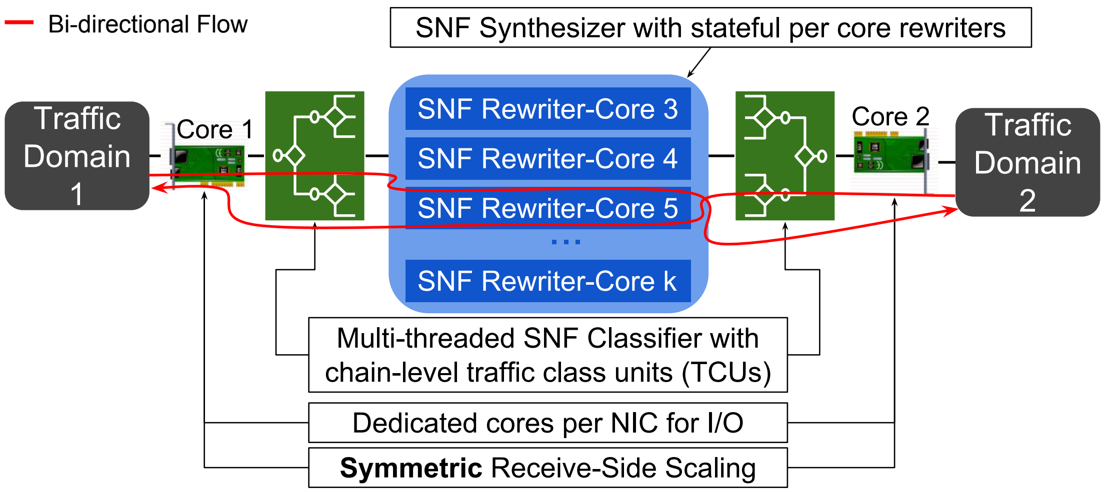


Figure 1. An overview of SNF running on a machine with k CPU cores and 2 NICs. Dedicated CPU cores per NIC deliver bi-directional flows to packet processing CPU cores via Symmetric RSS. Processing cores concurrently classify traffic and access individual, stateful SNF rewriters to modify the traffic.

Once a processing core acquires a frame, it executes SNF as shown in Figure 1. First the core classifies the frame (green rectangles in Figure 1) in one of the chain's TCUs and then applies the required synthesized modifications (blue rounded-rectangle in Figure 1) that correspond to this TCU. Both classification and modification processes are highly parallelized as different cores can simultaneously drive frames that belong to different TCUs out of the chain. We detail both processes in § 3.2.

However, the key point of Figure 1 lies in the fact that a core's pipeline shares nothing with any other pipeline. We employed the symmetric Receive Side Scaling (RSS) (Intel, 2016) scheme by Woo and Park (2012) to hash input traffic in a way that a flows' bi-directional packets are always served by the same SNF rewriter, hence the same processor. This scheme allows a processing core to drive a TCU at the maximum processing speed of the machine.

Main Objectives

The primary goal of SNF is to eliminate redundancy along the chain. The sources of redundancy in current NF chains and the solutions that our approach offers are:

A. Multiple network I/O interactions between the chain and the backend dataplane occur because each NF is an individual process. We solve this by placing NF chains in a single logical entity. Once a packet enters this entity, it does not exit until all the chain operations are applied.

B. Late packet drops appear in NF chain implementations when packets unnecessarily pass through several elements before getting dropped. SNF discards these packets as early as possible.

C. Multiple read operations on the same field occur because each NF contains its own decision elements. A typical example is an Internet protocol (IP) lookup in a chain of routers. While SNF is parsing the

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2477v2 | CC BY 4.0 Open Access | rec: 28 Sep 2016, publ: 28 Sep 2016

initial chain, it marks the read operations and constructs traffic classes encoded as paths of elements in a directed acyclic graph (DAG). Then, SNF synthesizes these elements into a *single* classifier to realize both routing and filtering.

D. Multiple write operations on the same field overwrite previous values. For example, the IP checksum is modified twice when a decrement time to live (TTL) operation follows a destination IP address modification. SNF associates a set of (stateful) write operations with a traffic class, hence it can modify each field of a traffic class all at once.

Next, we describe in detail how SNF *automatically* synthesizes the equivalent of a service chain.

SNF ARCHITECTURE

Taking into account the main objectives listed above, this section presents the design of SNF. § 3.1 defines the synthesis abstraction, § 3.2 presents the formal synthesis steps, and § 3.3 describes how stateful functions are realized.

Abstract Service Chain Representation

The crux of SNF's design is an abstract service chain representation. We begin by describing a mathematical model to represent packet units in § 3.1.1. Next, we model an NF's behavior in an abstract way in § 3.1.2. Finally, we define our target service-level network function in § 3.1.3.

Packet Unit Representation

Inspired by the approach of Kazemian et al. (2012), we represent each packet as a vector in a multi-dimensional space. However, we follow a protocol-aware approach by dividing a packet according to the unsigned integer value of the different header fields. Thus, if p is an IPv4/TCP packet, we represent it as:

$$p = (p_{\text{ip\_version}}, p_{\text{ip\_ihl}}, \ldots, p_{\text{tcp\_sport}}, p_{\text{tcp\_dport}}, \ldots)$$

From now on, we call P the space of all possible packets. For a given header field f of length l bits, we define a field filter F_f as a union of disjoint intervals $(0, 2^l - 1)$:

$$F_f = \bigcup_{s_i \subset (0, 2^l - 1)} s_i \text{ where } \begin{cases} \forall i, & s_i \text{ is an interval} \\ \forall i \neq j, & s_i \cap s_j = \emptyset \end{cases}$$

This allows grouping packets into a data structure that we call a *packet filter*, defined as a logical expression of the form:

$$\phi = \{(p_1, \ldots, p_n) \in P | (p_1 \in F_1) \wedge \ldots \wedge (p_n \in F_n)\}$$

where $(F_1, \ldots, F_n)$ are field filters. The space of all possible packet filters is Φ. Then:

$$u : \begin{cases} \phi & \mapsto & (F_1, .., F_n) \\ \Phi & \mapsto & \{(F_1, .., F_n) | \forall i, F_i\}_{(F_1, .., F_n)} \end{cases}$$

is a bijection and we can assimilate ϕ to $(F_1, \ldots, F_n)$.

If ϕ_1 and ϕ_2 are two packet filters defined by their field filters $(F_{1,1}, \ldots, F_{1,n})$ and $(F_{2,1}, \ldots, F_{2,n})$, then $\phi_1 \cap \phi_2$ is also a packet filter and is defined as $(F_{1,1} \cap F_{2,1}, \ldots, F_{1,n} \cap F_{2,n})$.

Network Function Representation

Network functions typically apply read and write operations to traffic. While our packet unit representation allows us to compose complex read operations across the entire header space, we still need the means to modify traffic. For this, we define an operation as a function $\omega : P \mapsto \Phi$ that associates a set of possible outputs to a packet. We add the additional constraint that for any given operation ω, there is $\omega_1, \ldots, \omega_n \in \mathbb{N}^{\mathbb{N}}$ such as:

$$\forall p = (p_1, \ldots, p_n) \in P, \omega(p) = (\omega_1(p_1), \ldots, \omega_n(p_n))$$

Note that we use sets of possible values (instead of fixed values) to model cases where the actual value is chosen at run-time (e.g., source port in an S-NAT). *Therefore, SNF does support both deterministic and conditional operations.*

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2477v2 | CC BY 4.0 Open Access | rec: 28 Sep 2016, publ: 28 Sep 2016

If we define Ω as the space of all possible operations, we can express a **processing unit** PU as a conditional function that maps packet filters to operations:

$$PU : p \mapsto \begin{cases} \omega_1(p) & \text{if } p \in \phi_1 \\ \ldots \\ \omega_m(p) & \text{if } p \in \phi_m \end{cases}$$

where $(\omega_1, \ldots, \omega_m) \in \Omega^m$ are operations and $(\phi_1, \ldots, \phi_m) \in \Phi^m$ are mutually distinct packet filters.

An NF is simply a DAG of PUs. For instance, SNF can express a simplified router's NF as follows:

$$NF_{ROUTER} : PU\{Lookup\} \rightarrow PU\{DecIPTTL\} \rightarrow PU\{IPChecksum\} \rightarrow PU\{MAC\}$$

where, 4 PUs take place. An IP lookup PU is followed by decrement IP TTL, IP checksum update, and source and destination MAC address modification PUs.

The Synthesized Network Function

In the previous section we laid the foundation to construct NFs as graphs of PUs. Now, at the service level where multiple NFs can be chained, we define a TCU as a set of packets/flows, represented by disjoint unions of packet filters, that are processed in the same fashion (i.e., undergo the same set of synthesized operations). This definition allows us to construct the service chain's $SynthesizedNF$ function (in short SNF) as a DAG of PUs, or equivalently, as a map of TCUs that associates operations to their packet filters:

$$SynthesizedNF : \Phi \mapsto \Omega$$

Formally, the complexity of the $SynthesizedNF$ is upper-bounded by the function $O(n \cdot m)$, where n is the number of TCUs and m is the number of packet filters (or conditions) per TCU. Each TCU turns a textual packet filter specification (such as "proto tcp && dst net 10.0/16 && src port 80") into a binary decision tree traversed by each packet. Therefore, in the absolute worst case, an input packet might traverse a skewed binary tree of the last TCU, yielding the above complexity bound. The average case occurs in a relatively balanced tree ($O(\log m)$), in which case the average complexity of the $SynthesizedNF$ is bounded by the function $O(n \cdot \log m)$.

Synthesis Steps

Leveraging the abstractions introduced in § 3.1, we detail the steps that translate a set of NFs into an equivalent SNF. The SNF architecture is comprised of three modules (shown in Figure 2). We describe each module in the following sections.

Service Chain Configurator

The top left box in Figure 2 is the Service Chain Configurator; the interface that a network operator uses to specify a service chain to be synthesized by SNF. Two inputs are required: a set of service components (i.e., NFs), along with their topology. SNF abstracts packet processing by using graph theory. That said, a chain is described as a DAG of interconnected NFs (i.e., chain-level DAG), where each NF is a DAG of abstract packet processing elements (i.e., NF DAG). The NF DAG is implementation-agnostic, similar to the approaches of Bremler-Barr et al. (2016); Anwer et al. (2015); Kohler et al. (2000). The network operator enters these inputs in a configuration file using the following notation:

Vertices (NFs): Each service component (i.e., an NF) of a chain is a vertex in the chain-level DAG for which, the Service Chain Configurator expects a name and an NF DAG specification (see Figure 2). Each NF can have any number of input and output ports as specified by its DAG. An NF with one input and one output interface is denoted as: $[interface_0]NF_1[interface_1]$.

Edges (NF inter-connections): The connections between NFs are the edges of the chain-level DAG. We interconnect two NFs as follows: $NF_1[interface_1] \rightarrow [interface_0]NF_2$.

No loops: Since the chain-level DAG is acyclic by construction, SNF must prevent loops (e.g., two interfaces of the same NF cannot be connected to each other).

Entry points: In addition to the internal connections within a chain (i.e., connections between NFs), the Service Chain Configurator also requires the entry points of the chain. These points are the interfaces of the chain with the outside world and indicate the existence of traffic sources. An interface that is neither internal nor an entry point can only be an end-point; these interfaces are discovered by the Service Chain Parser as described below.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2477v2 | CC BY 4.0 Open Access | rec: 28 Sep 2016, publ: 28 Sep 2016

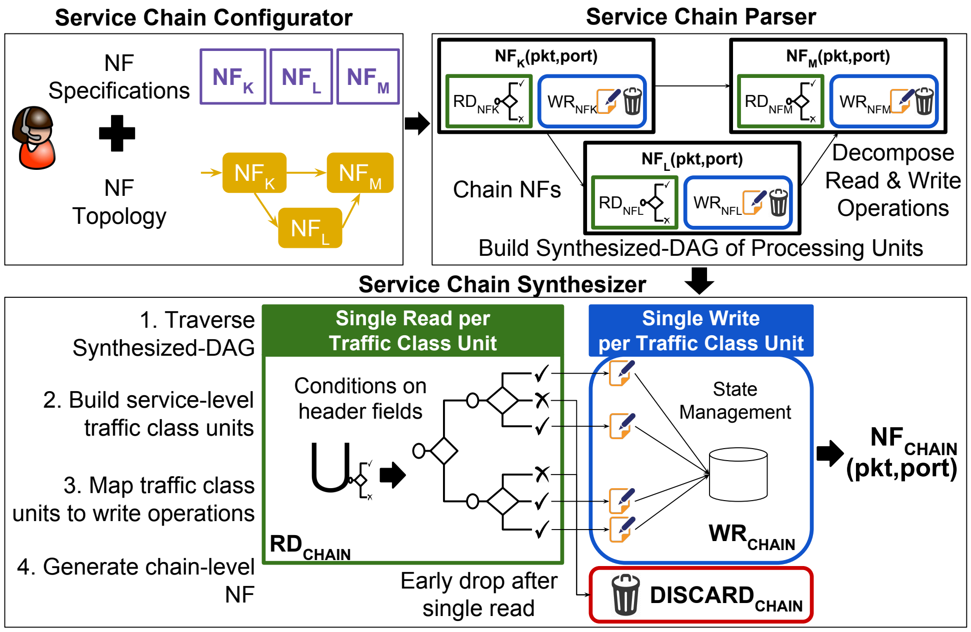


Figure 2. The SNF framework. The network operator inputs a service chain and its topology (top left part). SNF parses the chained NFs, decomposes their read and write parts, and composes a Synthesized-DAG (top right part). While traversing the Synthesized-DAG, SNF builds the TCUs of the chain, associates them with write/discard operations, leading to a synthesized chain-level NF.

Service Chain Parser

The Service Chain Configurator outputs a chain-level DAG that describes the chain to the Service Chain Parser. As shown in the top right box of Figure 2, the parser iterates through all of the input NF DAGs (i.e., one per NF); while parsing each NF DAG, the parser marks each element according to its type. We categorize NF elements in four types: I/O, parsing, read, and write elements. As an example NF, consider a router that consists of interconnected elements, such as *ReadFrame, StripEthernetHeader, IPLoookUp, and DecrementIPTTL*. ReadFrame is an I/O element, StripEthernetHeader is a parsing element (moves a frame's pointer), IPLoookUp is a read element, while DecrementIPTTL is a write element.

The parser stitches together all the NF DAGs based on the topology graph and builds a Synthesized-DAG (see Figure 2) that represents the entire chain. This process begins from an entry point and searches recursively until an output element is found. If the output element leads to another NF, the parser keeps a jump pointer and cross checks that the encountered interfaces match the interfaces declared in the Service Chain Configurator. After collecting this information, the parser omits the I/O elements because one of SNF's objectives is to eliminate inter-NF I/O interactions. The process continues until an output element that is not in the topology is found; such an element can only be an **end-point**. Along the path to an output element the parser separates the read from the write elements and transforms NF elements into PUs, according to § 3.1.2. Next, the parser considers the next entry point until all are exhausted.

The final output of the Service Chain Parser is a large Synthesized-DAG of PUs that models the behavior of the entire input service chain.

Service Chain Synthesizer

After building the Synthesized-DAG, our next target is to create the *SynthesizedNF* introduced in § 3.1.3. To do so, we need to derive the SNF's TCUs. To build a TCU we execute the following steps: from each entry port of the Synthesized-DAG, we start from the identity TCU $tcu_0 \in \Phi \times \Omega$ defined as: $tcu_0 = (P, id_P)$, where id_P is the identity function of P, i.e., $\forall x \in P, id_P(x) = x$. Conceptually, tcu_0 represents an empty packet filter and no operations, which is equivalent to a transparent NF. Then, we search the Synthesized-DAG, while updating our TCU as we encounter conditional (read) or modification (write) elements. Algorithms 1 and 2 build the TCUs using an adapted depth-first search (DFS) of the Synthesized-DAG.

Now let us consider a TCU t, defined by its packet filter ϕ and its operation ω, that traverses a PU U using the adapted DFS. The TRAVERSE function in Algorithm 1 creates a new TCU for each possible pair of (ω_i, ϕ_i). In particular, it creates a new packet filter ϕ' returned by the INTERSECT function (line 3). This function is described in Algorithm 2 and considers previous write operations while updating a packet filter. For each field filter ϕ_i of a packet filter, the function checks whether the value has been modified by the corresponding ω_i operation (condition in line 8) and whether the written value is in the intersecting field filter ϕ_i^0 (line 10). It then updates the TCU by intersecting it with the new filter, if the value has not been modified (action in line 8). After the INTERSECT function returns in Algorithm 1, TRAVERSE creates a new operation by composing ω and ω_i (line 4).

The recursive algorithm terminates in two cases: (i) when the packet filter of the current TCU is the empty set, in which case the function does not return anything, (ii) when the PU U does not have any successors, in which case it returns the current TCUs. In the latter case, the returned TCUs comprise the final *SynthesizedNF* function.

Algorithm 1 Building the SNF TCUs

1: **function** TRAVERSE($t = (\phi, \omega)$,$U = \{(\phi_i, \omega_i)_{i \leq m}\}$)
2:     **for** $i \in (1, m)$ **do**0
3:         $\phi' \leftarrow$ INTERSECT(t, ϕ_i)
4:         $\omega' \leftarrow \omega_i \circ \omega$
5:         $t' = (\phi', \omega')$
6:         TRAVERSE(t',$U.successors[i]$)

Algorithm 2 Intersecting a TCU with a filter

1: **function** INTERSECT($t = (\phi, \omega), \phi^0$)
2:     $\phi' \leftarrow P$
3:     $(\omega_1, ..., \omega_n) \leftarrow \omega$.COORDINATES
4:     $(\phi_1, ..., \phi_n) \leftarrow \phi$.COORDINATES
5:     $(\phi_1^0, ..., \phi_n^0) \leftarrow \phi^0$.COORDINATES
6:     $(\phi_1', ..., \phi_n') \leftarrow \phi'$.COORDINATES
7:     **for** $i \in (1, n)$ **do**
8:         **if** $\omega_i = id_{\mathbb{N}}$ **then** $\phi_i' \leftarrow \phi_i \cap \phi_i^0$
9:         **else**
10:             **if** $\omega_i(\phi_i) \subset \phi_i^0$ **then** $\phi_i' \leftarrow \phi_i$
11:             **else** $\phi_i' \leftarrow \emptyset$
12:     **return** ϕ'

Managing Stateful Functions

A difficulty when synthesizing NF chains is managing successive stateful functions. It is crucial to ensure that the states are properly located in a synthesized NF and that every packet is matched against the correct state table. At the same time, SNF should hold the promise that NFV service chains must be realized without redundancy, hence single-read and single-write operations must be applied per packet.

To highlight the challenges of maintaining the state in a chain of NFs, consider the example topology shown in Figure 3. In this example, a large network operator has run out of private IPv4 addresses in the 10.0/8 prefix and has been forced to share the same network prefix between two distinct zones (i.e., zones 1 and 2), using a chain of NAPTs. This is not unlikely to happen, as an 8-byte network prefix contains less than 17 million addresses and recent surveys have predicted that 50 billion addresses will be connected to the Internet by 2020 (Evans, D., 2011).

Consolidating this chain of NFs into a single SNF instance poses a problem. That is, traffic originating from zones 1 and 2 shares the same source IP address and port range, but to ensure that all the traffic is translated properly, the corresponding synthesized chain must share their NAPT table. However, since traffic also shares the same destination prefix (i.e., towards the same Internet gateway), a host from the outside world cannot possibly distinguish the zone where the traffic is originating from.

Obviously, the question that SNF has to address in general, and particularly in this example is: “How can we synthesize a chain of NFs, ensuring that (i) traffic mappings are unique and (ii) no redundant operations will be applied?” To solve this conundrum, the SNF design respects the following properties:

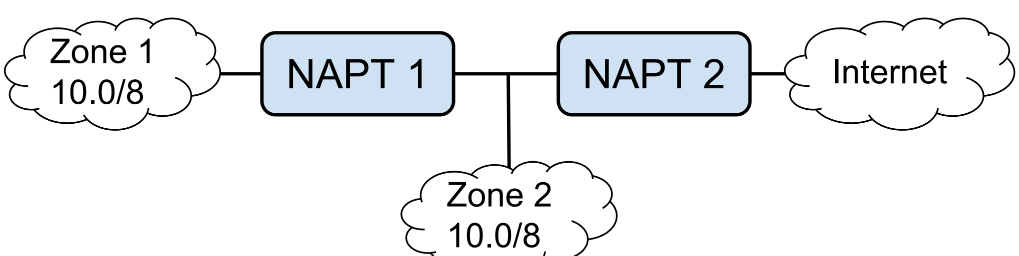


Figure 3. Example of stateful NAPT chains, where two zones share the same IPv4 prefix.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2477v2 | CC BY 4.0 Open Access | rec: 28 Sep 2016, publ: 28 Sep 2016

Property 1 We enforce the uniqueness of flow mappings by ensuring that all egress traffic that shares the same last stateful (re)write operation also shares the same state table.

Property 2 The state table of SNF must be origin-aware. To redirect ingress traffic towards the correct interface, while respecting the single-read principle of SNF, the SNF state table must collocate flow information and the origin interface for each flow.

To generalize the state management problem, Figure 4 shows how SNF handles stateful configurations with e.g., three egress interfaces. We apply "Property 1" by having exactly one stateful (re)write element (denoted as Stateful RW) per egress interface. We apply "Property 2" by having one input port in each of these (re)write elements, associated with an ingress interface. Therefore, a state table in SNF not only contains flow-related information, but also keeps a linking of a flow entry with its origin interface.

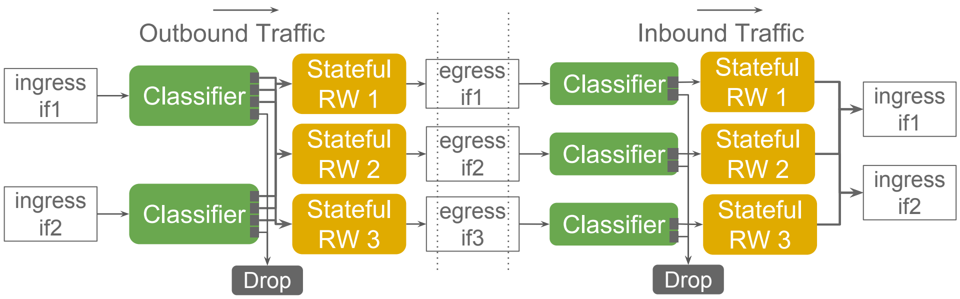


Figure 4. State management in SNF.

A MOTIVATING USE CASE

To understand how SNF works and what benefits it can offer, we quantify the processing and I/O redundancies in an example use case of an NF chain and then compare it to its synthesized counterpart. We use Click to specify the NF DAGs of this example, but SNF is applicable to other frameworks. The example chain consists of a NAPT, a L4 firewall (FW), and a L3 load balancer (LB) that process transmission control protocol (TCP) and user datagram protocol (UDP) traffic as shown in Figure 5.

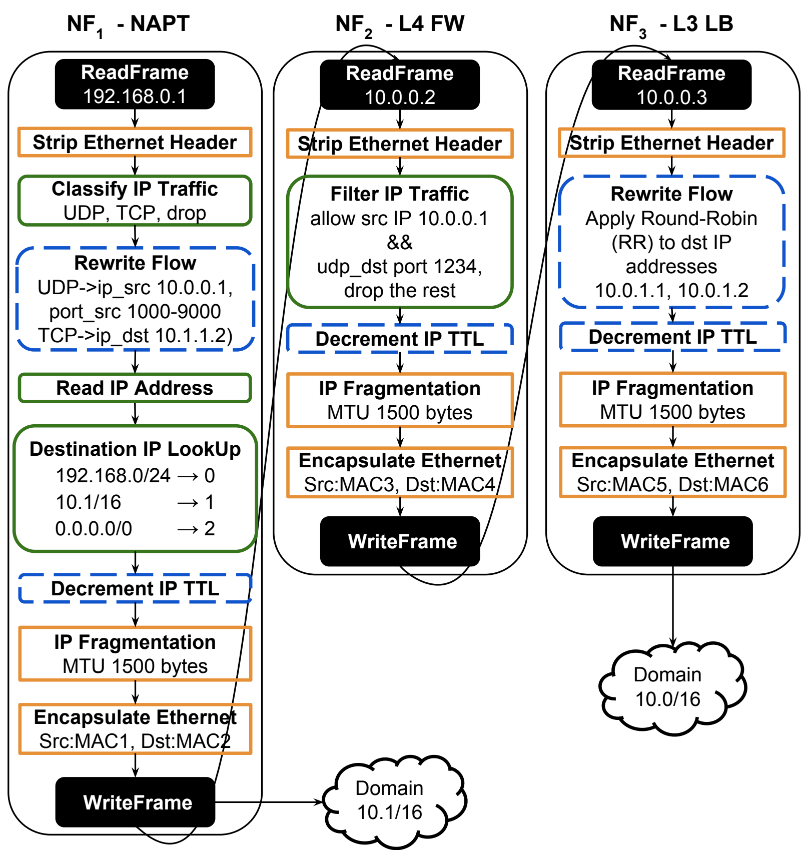


Figure 5. The internal components of an example NAPT - L4 FW - L3 LB chain.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2477v2 | CC BY 4.0 Open Access | rec: 28 Sep 2016, publ: 28 Sep 2016

The TCP traffic is NAPT'ed in the first NF and then leaves the chain, while UDP is filtered at the FW (the second NF) and the UDP datagrams with destination port 1234 are load balanced across two servers by the last NF. For simplicity, we discuss only the traffic going in the direction from the NAPT to the LB.

The rectangular operations in Figure 5 are interface-dependent, e.g., an "Encapsulate Ethernet" operation encapsulates the IP packets in Ethernet frames before passing them to the next NF where a "Strip Ethernet Header" operation turns them back into IP packets. Such operations occur 3 times because there are 3 NFs, instead of only once (because the processing operates at the IP layer). Ideally, strip should be applied before, and Ethernet encapsulation after all of the IP processing operations. Similarly, the "IP Fragmentation" should only be applied before the final Ethernet encapsulation.

The remaining operations (illustrated as rounded rectangles) of the three processing stages are those that (*i*) make decisions based upon the contents of specific packet fields (read operations with a solid round outline, e.g., "Classify IP Traffic" and "Filter IP Traffic") or (*ii*) modify the packet header (rewrite operations with a blue dashed outline e.g., "Rewrite Flow" and "Decrement IP TTL"). We found redundancy in both types of operations. In the read operations, one IP classifier is sufficient to accommodate the three traffic classes of this example and perform the routing. Thus, all the round-outlined operations with solid lines (green) can be replaced by a single "Classify IP Traffic" operation.

Large savings are also possible with the rewrite operations. For example, the initial chain calculates the TTL field 3 times and IP checksum 5 times, whereas only one computation for these fields suffices in the synthesized chain. Based on our measurements on an Intel Xeon E5 processor the checksum calculations cost 10-40 CPU cycles/packet. By integrating the "Decrement IP TTL" into the "Rewrite Flow" operation and enforcing the checksum calculation only once, saves 237 CPU cycles/packet.

Figure 6 depicts a synthesized version of the NF chain shown in Figure 5. Following the SNF paradigm presented in § 3, the synthesized chain forms a graph with two main parts. The left-most part (rounded rectangles with solid outline in Figure 6) encodes all the read operations by composing paths that begin from a specific interface and traverse the three traffic classes of this chain, until a packet is output or dropped. Each path keeps a union of filters that represents the header space that matches the respective traffic class. In this example, the filter for e.g., the allowed UDP packets is the union of the protocol and destination port numbers. Such a filter is part of a classifier whose output port is linked with a set of write operations (dashed vertices in Figure 6) associated with this traffic class (right-most part of the graph). As shown in Figure 6, with SNF a packet passes through all the read operations once (guaranteeing a single-read) and either the packet is discarded early or each header field is written once (ensuring a single-write) before exiting the chain.

Synthesizing the counterpart of this example implies several code modifications to avoid the redundancy caused by the design of each NF. To apply a per flow, per-field single-write operation we ensure that the "Rewrite Flow" will smartly calculate the checksums once IP addresses, ports, and the IP TTL fields are written. Therefore, in this example we saved four unnecessary operations (3 "Decrement IP TTL" and 1 "Rewrite Flow") and four checksum calculations (3 IP and 1 IP/UDP). Moreover, integrating all decisions (i.e., routing, filtering) in one classifier caused this operation to be slightly heavier, but saved another two redundant function calls to "Destination IP LookUp" and "Filter IP Traffic" respectively.

The final form of the synthesized chain requires only 5 processing operations to transfer the UDP datagrams along the entire chain. The initial chain implements the same functionality using 18 processing

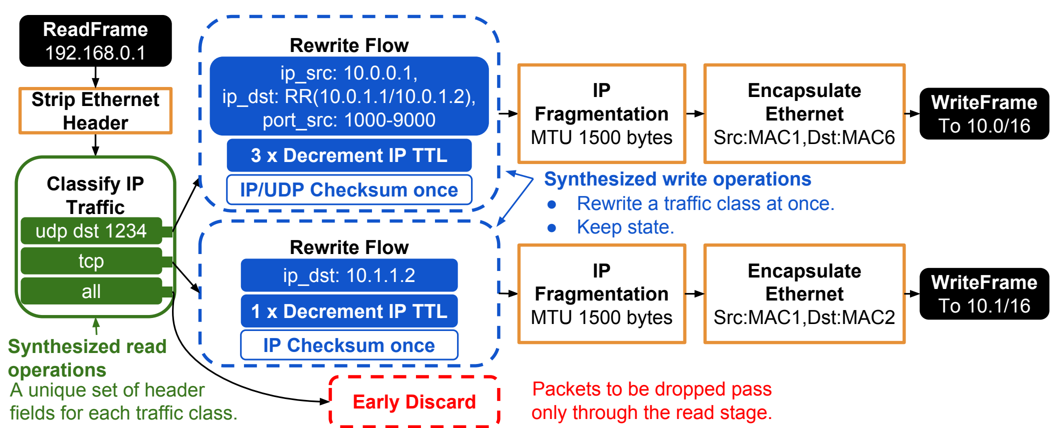


Figure 6. The synthesized chain equivalent to Figure 5. The SNF contributions are shown in floating text.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2477v2 | CC BY 4.0 Open Access | rec: 28 Sep 2016, publ: 28 Sep 2016

operations and two additional pairs of I/O operations. Based on our measurements the total *processing* cost of the initial chain is 2206 cycles/packet, while the synthesized chain requires 3x less (roughly 720) cycles/packet. If we account for the extra I/O cost per hop for the initial chain the difference becomes even greater. In production service chains, where packets arrive at high rates, this overhead can play a major role in limiting the throughput of the chain and the imposed latency; therefore, the advantages of synthesizing more complex service chains than this simple use case are expected to be even greater.

IMPLEMENTATION

As we stated earlier, SNF's basic assumption is that each input service component (i.e., NF) is expressed as a graph (i.e., the NF DAG), composed of individual packet processing elements. This allows SNF to parse the NF DAG and infer the internal operations of each NF, producing a synthesized equivalent. Among the several candidate platforms that allow such a representation, we developed our prototype atop Click because it is the most widely used NFV platform in the academia. Many earlier efforts built upon it to improve its performance and scalability, hence we believe that this choice will maximize SNF's impact as it allows direct comparison with state-of-the-art Click variants such as RouteBricks (Dobrescu et al., 2009), PacketShader (Han et al., 2010), Double-Click (Kim et al., 2012), SNAP (Sun and Ricci, 2013), ClickOS (Martins et al., 2014), and FastClick (Barbette et al., 2015).

We adopt FastClick as the basis of SNF as it uses DPDK, a state-of-the-art user-space I/O framework that exploits modern hardware amenities (including multiple CPU cores) and NIC features (including multiple queues and offloading mechanisms). Along with batch processing, non-uniform memory access support, and fine grained CPU core affinity techniques, FastClick scales a single router achieving line-rate throughput at 40 Gbps. **SNF aims for similar performance for an entire service chain**.

FastClick Extensions

We implemented SNF in C++11. The modules depicted in Figure 2 are 14376 lines of code. The integration with FastClick required another 1500 lines of code (modifications and extensions). Although FastClick improves a router's throughput and latency, it lacks features required for broader NFV applications; therefore, we made the following extensions to target a service-oriented platform:

Extension 1: Stateful elements that deal with flow processing such as IP/UDP/TCPRewriter were not equipped with FastClick's optimizations such as computational batching or cache prefetching. Moreover, these elements were not designed to be thread-safe hence they could cause race conditions when accessed by multiple CPU cores at the same time. We designed thread-safe data structures for these elements while also applying the necessary modifications to equip them with the FastClick optimizations.

Extension 2: We tailored several packet modification FastClick elements to comply with the synthesis principles, as we found that their implementation was not aligned with our single-write approach. For instance, we improved the IP/UDP/TCP checksum calculations by calling the respective functions only once all the header field modifications are applied. Moreover, we extended IP/UDP/TCPRewriter elements with additional input arguments. These arguments extend the elements' packet modification capabilities (e.g., decrement IP TTL field to avoid unnecessary element calls) and guarantee that a packet entering these elements undergo a single-write operation per header field.

Extension 3: We developed a new element, called IPSynthesizer, in the heart of our execution model shown in Figure 1. This element implements per-core stateful flow tables that can be safely accessed in parallel allowing multiple TCUs to be processed at the same time. To avoid inter-core communication, thus keep the per-core cache(s) hot, we extended the RSS mechanism of DPDK (see Figure 1) using a symmetric approach proposed by Woo and Park (2012).

Extension 4: To make software-based classification more scalable, we implemented the lazy subtraction algorithm introduced in Header Space Analysis (HSA) (Kazemian et al., 2012). With this extension, SNF aggregates common IP prefixes in a filter and applies the longest one while building a TCU, thus producing shorter traffic class expressions. [†]

Our prototype supports a large variety of packet processing libraries, fully covering both native FastClick and hypervisor-based ClickOS deployments. Our prototype also takes advantage of FastClick's computation batching with a processing core moving a group of packets between the classifier and the synthesizer with a single function call. New packet processing elements can be incorporated with minor effort. We made the FastClick extensions available at Katsikas, Georgios (2016).

[†] This extension is not a direct part of FastClick, since the optimized classification rules are computed by SNF beforehand; then, SNF uses these rules as arguments when calling FastClick's Classifier or IPClassifer elements.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2477v2 | CC BY 4.0 Open Access | rec: 28 Sep 2016, publ: 28 Sep 2016

PERFORMANCE EVALUATION

Recent efforts, such as ClickOS (Martins et al., 2014) and NetVM (Hwang et al., 2014), are unable to maintain constant high throughput and low latency for chains of more than 3 NFs when processing packets at high speed. This problem hinders large-scale hypervisor-based NFV deployments that could reduce network operators' expenses and provide more flexible network management and services (Cisco, 2014; SDX Central, 2015).

We envision SNF to be the key component of future NFV deployments, thus we evaluate the synthesis process using real service chains to exercise its true potential. In this section, we demonstrate SNF's ability to address three types of service chains:

Chain 1: Scale a long series of routers at the cost of a single router.

Chain 2: Nest multiple NAPT middleboxes.

Chain 3: Implement high performance ACLs of increasing cardinality at the borders of ISP networks.

We use the experimental setup described in § 6.1 to measure the performance of the above three types of chains and answer the following questions: Can we synthesize (stateful) chains *without* sacrificing throughput as we increase the chain length (see § 6.2, § 6.3)? What is the effect of different packet sizes on a system's throughput (see § 6.3)? What are the current limits of purely software-based packet processing (see § 6.4) and how can we overcome them (see § 6.5)?

Testbed

We conducted our experiments on six identical machines each with a dual socket 16-core Intel® Xeon® CPU E5-2667 v3 clocked at 3.20 GHz. The cache sizes are: 2x32 KB L1, 256 KB L2, and 20 MB L3. Hyper-threading is disabled and the OS is the Ubuntu 14.04.1 distribution with Linux kernel v.3.13. Each machine has two dual-port 10 GbE Intel 82599 ES NICs.

Unless stated otherwise, we use two machines to generate and sink bi-directional traffic using MoonGen (Emmerich et al., 2015), a DPDK-based traffic generator. MoonGen allows us to saturate 10 Gbps NICs on a single machine using a set of cores, while receiving the same amount of traffic on another set of cores. To gain insight into the performance of the service chains, we measure the throughput and end-to-end latency to traverse the chains, at the endpoints. We use FastClick as a baseline and compare FastClick against SNF (which extends FastClick). We create service chains that run natively in a single process using RSS and multiple CPU cores, as this is the fastest FastClick configuration. We follow two different setups for our software-based and hardware-assisted SNF deployments as follows.

Software-based SNF: In § 6.2, § 6.3, and § 6.4 we stress different purely software-based NFV service chains that run in one machine following the execution model of Figure 1. This machine has 4 10 GbE NICs connected to the two traffic source/sink machines (two NICs on each machine), hence the total capacity of the NFV machine is 40 Gbps. The goal of this testbed is to show how much NFV processing FastClick and SNF can fit into a single machine and what processing limits this machine has.

Hardware-assisted SNF: For the complex NFV service chains, presented in § 6.4, we also deploy a testbed (see § 6.5) where we offload the traffic classification to a Noviflow 1132 OpenFlow switch with firmware 300.1.0. The switch is connected with two 10 GbE NICs to each of the two senders/receivers, and with one link to each of the four processing servers in our SNF cluster. This testbed has a total of 40 Gbps capacity (same as the software-based setup above), but the processing is distributed to more machines in order to show how our SNF system scales.

A Chain of Routers at the Cost of One

This first use case targets a direct comparison with the state-of-the-art. Specifically, we chain a popular implementation of a software-based router that, after several years of successful research contributions (Dobrescu et al., 2009; Han et al., 2010; Kim et al., 2012; Sun and Ricci, 2013; Martins et al., 2014; Barbette et al., 2015), achieves scalable performance at tens of Gbps.

As we show in this section, a naive chaining of individual, fast NFs does not achieve high performance. To examine this we linearly connect 1-10 FastClick routers, where each router has four 10 Gbps ports (hence such a chain has a 40 Gbps link capacity). The down-pointing (green) triangular points of Figure 7 show the throughput achieved by these chains versus the increasing length of the chains, when we inject 60-bytes long frames, excluding the cyclic redundant check (CRC). The maximum throughput for this frame size size is 31.5 Gbps and this is the limit of our NICs, as reported earlier (Barbette et al., 2015).

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2477v2 | CC BY 4.0 Open Access | rec: 28 Sep 2016, publ: 28 Sep 2016

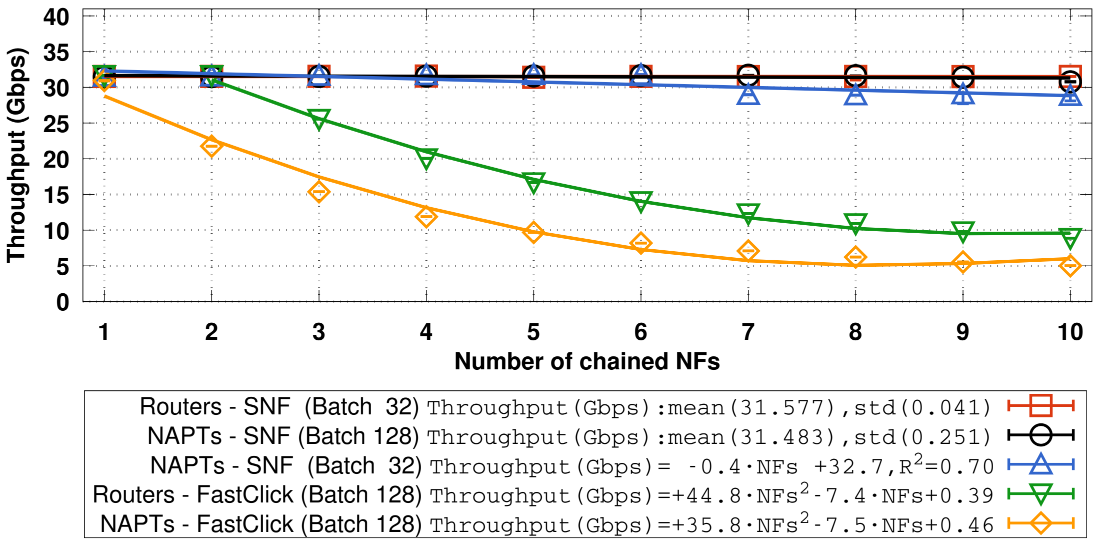


Figure 7. Throughput (Gbps) of chained routers and NAPTs using (*i*) FastClick and (*ii*) SNF versus the numbers of chained NFs (60-byte frames are injected at 40 Gbps). Bigger batch sizes achieve higher throughput.

In our experiment, FastClick can operate at the maximum throughput only for a chain of 1 or 2 routers. As denoted by the equation in this fit to the graph, after this point there is a quadratic throughput degradation that results in a chain of 10 routers achieving less that 10 Gbps of throughput.

SNF automatically synthesizes this simple chain (shown with red squares) to achieve the maximum possible throughput using this hardware, despite the increasing length of the chain. The fitted equation confirms that SNF operates at the speed of the NICs.

Stateful Service Chaining

The problem of Service Function Chaining has been recently investigated by Quinn and Nadeau (Quinn and Nadeau, 2015) and several relevant use cases (Liu et al., 2014) have been proposed. In some of these use cases, traffic needs to support distinct address families while traversing different networks. For instance, within an ISP, IPv4/IPv6 traffic might either be directed to NAT64 (Bagnulo et al., 2011), or a Carrier Grade NAT (Perreault et al., 2013). In more extreme cases, this traffic might originate from different access networks such as fixed broadband, mobile, datacenters, or cloud customer premises (CPE), thus causing the nested NAT problem (Penno et al., 2013).

The goal of this use case is to test SNF in such a stateful context using a chain of 1-10 NAPTs. Each NAPT maintains a state table that stores the original and translated source and destination IP addresses and ports of each flow, associated with the input interface where a flow was originated. The rhomboid points of Figure 7 show that the chains of FastClick NAPTs suffer a steeper (according to the fitted equation) quadratic degradation than the FastClick routers. Although we extended FastClick to support thread-safe, parallelized NAPT operations across multiple cores, it is still unable to drive the NAPT chain at line-rate, despite using 8 CPU cores and 128-packet batches.

SNF requires a certain batch size to realize the synthesized NAPT chains at the speed of hardware as shown by the black circles of Figure 7. The curve with the up-pointing (blue) triangles indicates that a batch size of 32 packets leads to a slight throughput degradation after the 6^{th} NAPT in the chain. State lookup and management operations executed for every packet cause this degradation. Depending on the performance targets, a network operator might tolerate an increased latency to achieve the higher throughput offered by an increased batch size.

Next, we explore the effect of different frame sizes on the chains of routers and NAPTs. We run the longest chain (i.e., 10 NFs) for frame sizes in [60, 1500] (bytes). Figure 8 shows that SNF follows the NICs' performance achieving line-rate forwarding at 40 Gbps for frames greater than 128 bytes. FastClick catches up the line-rate performance for frame sizes greater than 800-1000 bytes.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2477v2 | CC BY 4.0 Open Access | rec: 28 Sep 2016, publ: 28 Sep 2016

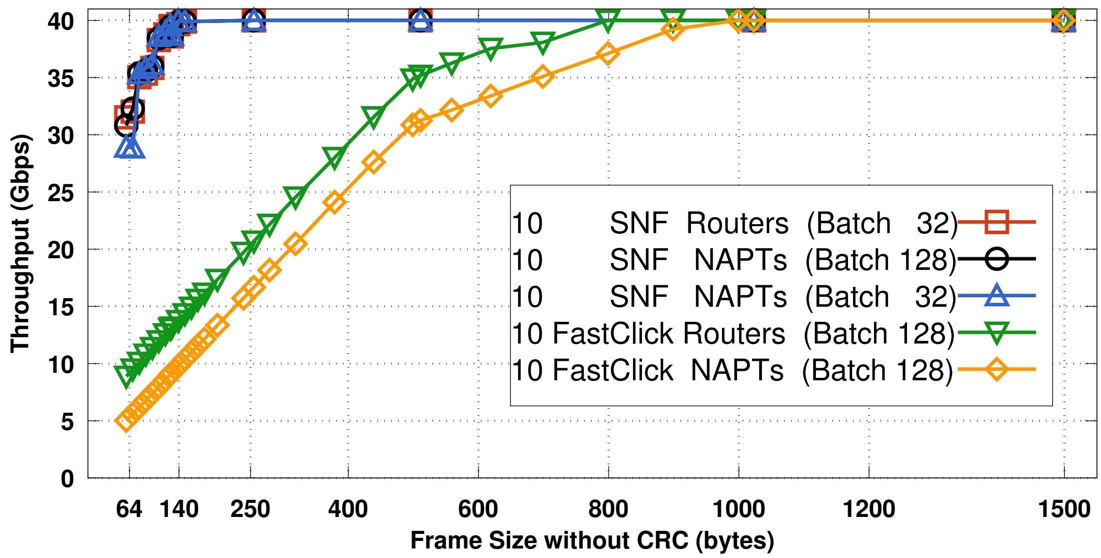


Figure 8. Throughput of 10 routers and NAPTs chained using (i) FastClick and (ii) SNF versus the frame size in bytes (without CRC). The different frames are injected at 40 Gbps.

Real Service Chain Deployments

Another common use case for an ISP is to deploy a service chain of a FW, a router, and a NAPT as depicted in Figure 9. The FW of such a chain may contain thousands of rules in its ACL causing serious performance issues for software-based NF implementations.

In this section we measure the performance of SNF using actual FW configurations of increasing cardinality and complexity, while exploring the limits of software-based packet processing on our hardware. We utilize a set of three actual ACLs (Taylor and Turner, 2007), taken from several ISPs, to deploy the service chain of Figure 9. The FW implements one ACL with 251, 713, or 8550 entries. The second NF is a standards-compliant IP router that redirects packets either towards the ISP's domain (intra-ISP traffic with prefix 204.152.0.0/16) or to the Internet. For the latter traffic, the third NF interconnects the ISP with the Internet by performing source and destination NAPT.

We use the above ACLs to generate traces of 64-byte frames that systematically exercise all of their entries. The generated packets emulate intra-ISP, inbound and outbound Internet traffic (see Figure 9). Figure 10 presents the performance of the 3 chains versus the different frames sizes (64, 128, 256, and 1500 bytes). We implemented the chains in FastClick and a purely software-based SNF using the full capacity of our processor's socket (i.e., 8 cores in one machine), symmetric RSS, and a batch size of 128 packets.

Figure 10a shows that the small ACL (251 rules), executed as a single FastClick instance, achieves satisfactory throughput, equal to its synthesized counterpart. This indicates that a small ISP or a chain

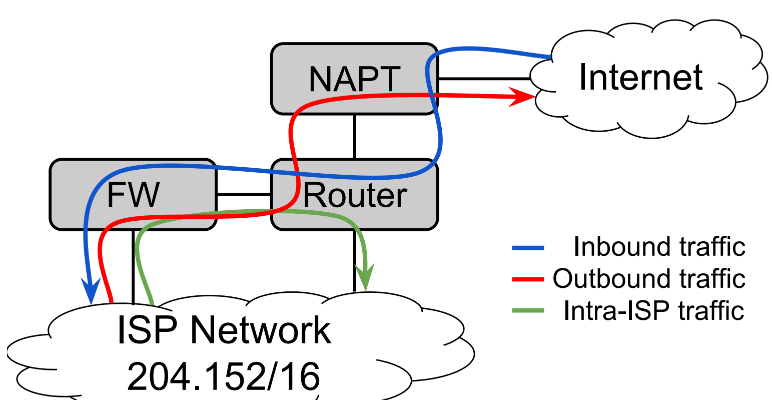
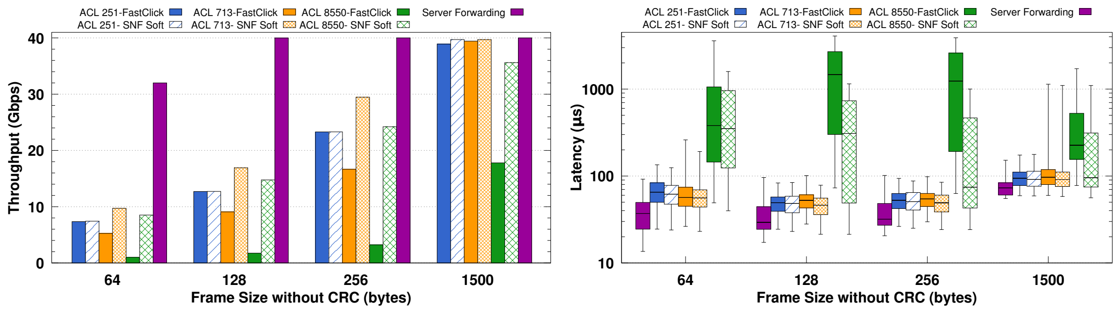


Figure 9. An ISP's service chain that serves inbound and outbound Internet traffic as well as intra-ISP traffic using three NFs.



(a) Throughput (Gbps)

(b) Latency (μs) on a logarithmic scale. The lower and upper percentiles are 1% and 99% respectively.

Figure 10. System's performance versus 4 frame sizes (64, 128, 256, and 1500 bytes) of three different ISP-level chains with 251, 713, and 8550 rules in their ACLs. FastClick and SNF implement these chains in software using 8 CPU cores (in a single machine with four NICs), symmetric RSS, and batch size of 128 packets. Input rates are 40 Gbps for the throughput test and 5 Gbps for the latency test.

deployment in small subnets (e.g., using links with capacity equal or less than 10 Gbps) may not fully benefit from SNF. As depicted in Figure 10b, the latency is also bounded below 100 μs. This time is dominated by the fact that our traffic flows as follows: traffic originating from one machine enters an SNF server and, after being processed, sent back to the origin server. We believe that the observed latency values are realistic for such a topology.

However, for the ACLs with 713 and 8550 rules the combination of all possible traffic classes among the FW, router, and NAPT boxes causes the classification tree of the chain to explode in size, hence **synthesis is a powerful yet necessary solution**. This causes three problems to FastClick: (*i*) the throughput when executing the last two ACLs (713, and 8550 rules) is reduced by almost 1.5x-10x respectively (on average), (*ii*) the median latency of the largest ACL is at least an order of magnitude greater than the median latencies of the smaller ACLs (see Figure 10b), and consequently (*iii*) the 99th percentile of the latency increases (up to almost 4 ms).

In contrast, SNF effectively synthesizes the large ACLs (i.e., 713 and 8550 rules) maintaining high throughput despite their increasing complexity. In the case of 713 rules, the synthesis is so effective that leads to better throughput than the 251-rule case. Regarding latency, SNF demonstrates 1.1-10x lower median latency (bounded below 500 μs) and 2-3.5x lower latency variance (slightly above 1 ms in some cases). The throughput gain of SNF is up to 8.5x greater than the FastClick chains.

Hardware-accelerated SNF

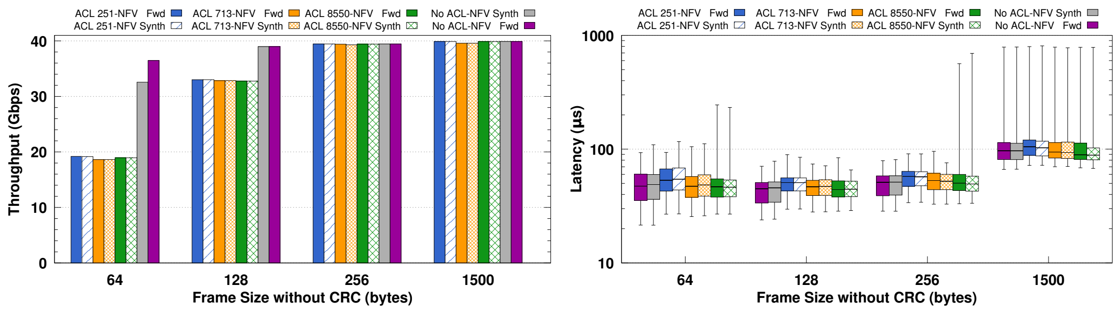
The results presented in the previous section show that software-based SNF cannot handle packet processing at a high enough rate when the NFs are complex. We analyzed the root cause and concluded that the packet classifier (that dispatches incoming packets to synthesized NFs) is the bottleneck. To overcome this problem, we run additional experiments, in which we offload packet classification to a hardware OpenFlow switch (since commodity NICs do not offer sufficient programmability). By doing so, we showcase SNF's ability to scale to high data rates with realistic NFs. In addition, we hint at the performance that is potentially achievable by offloading packet classification to a programmable interface.

Throughput Measurements

This extended version of SNF includes a script that converts the classification rules computed by the original SNF to OpenFlow 1.3 rules. The translation is not straightforward because the switch rules are less expressive than the ones accepted by the NFs. Specifically, rules that match on TCP and UDP port ranges are problematic. While OpenFlow does allow only matches on concrete values of ports, naive unrolling of ranges into multiple OpenFlow matches leads to an unacceptable number of rules. Instead, we solve the problem by utilizing a pipeline of flow tables available in the switch. The first two tables match only on the source and destination ports respectively, assign them to ranges, and write metadata that defines the range. Further tables include the real ACL rules and also match on the metadata previously added to a packet. Moreover, since the rules in the NFs are explored in the top-to-bottom order, we emulate the same behavior by assigning decreasing priorities to the OpenFlow rules.

We use the same sets of ACLs as before, and evaluate throughput and latency in the hardware-accelerated SNF. We first measure the throughput that SNF can achieve leveraging OpenFlow classification.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2477v2 | CC BY 4.0 Open Access | rec: 28 Sep 2016, publ: 28 Sep 2016



(a) Throughput (Gbps) **(b)** Latency (μs) on a logarithmic scale. The lower and upper percentiles are 1% and 99% respectively.

Figure 11. Hardware-assisted SNF's performance versus 4 frame sizes (64, 128, 256, and 1500 bytes) of three different ISP-level chains with 251, 713, and 8550 rules in their ACLs. SNF' s classification is offloaded to an OpenFlow switch, while stateful processing occurs in 4 servers connected to the switch. Input rates are 40 Gbps for the throughput test and 5 Gbps for the latency test.

We design an experiment where two machines use a total of four 10 Gbps links to send traffic. The packets are crafted so that they uniformly exercise all visible classification rules (some rules from the original data set are fully covered by other rules). We use the same frame sizes as in § 6.4. The switch classifies the packets and forwards them across four SNF servers that are using 10 Gbps links to connect to the switch. The servers work in two modes: (i) forward only, where they do not implement any NFs and simply forward packets (the first bar in each pair in Figure 11a), and (ii) synthesized mode, where they implement the real NF chain (the second bar in each pair in Figure 11a). Additionally, for comparison, we created an experiment where the switch installs only four basic classification rules (to do simple forwarding) to measure the performance of the NFs themselves (the last pair of bars in Figure 11a).

We observe that throughput depends mostly on the frame size. The system can operate at almost 20 Gbps for small frames (i.e., 64 bytes), and it reaches the full line-rate for 256-byte frames. Interestingly, the rule set size does not affect the throughput.

In the real data sets, the second bar in each pair is almost as high as the first one, which shows that the *software part of SNF does not limit the performance*. Finally, with simple forwarding rules in the switch (the first pair of bars in Figure 11a) the overall throughput is high even for small frames, which confirms that packet processing at the switch is the bottleneck of the whole system. To further prove this point, we run an experiment with only 2 ports sending traffic at an aggregate speed of 20 Gbps. In this case, SNF processes packets at the line-rate except for the smallest frames, where it achieves 15 Gbps.

Latency Measurements

A middlebox chain should induce low, bounded packet processing delays. In this set of experiments, we send traffic at a lower rate and measure latency. The setup is the same as in the previous scenario. Thus, the latency we show includes the time for frames to be: (i) transmitted out of the network interface of the traffic generating machines, (ii) received, processed, and forwarded by the OpenFlow switch, (iii) received, processed, and forwarded by the SNF machines, and (iv) received by the destination server (the same machine as the sender).

Figure 11b shows the latency depending on the frame size and the synthesized function (results for the input rate of 20 Gbps are very similar). Our results show that the median latencies are low and stable across all frame sizes and chains. There are several main observations here. First, the 75th percentiles (marked by the top horizontal line of the boxplots) are close to the median latencies and we find this result to be encouraging. Second, large frames (i.e., 1500 bytes) face two times greater median latency than the smaller ones regardless of the rule configuration. Third, there are outliers that are an order of magnitude less/greater than the medians (e.g., 10 μs at the 1st and 100 μs at 99th percentiles for 64-byte frames and 80 μs at the 1st and 800 μs at 99th percentiles for MTU-sized frames). Part of this latency variance is due to the batch I/O and processing techniques of the FastClick framework; as shown in Figure 11, these techniques offer high throughput, but have a well-studied effect on the latency variance.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2477v2 | CC BY 4.0 Open Access | rec: 28 Sep 2016, publ: 28 Sep 2016

VERIFICATION

In this section we discuss possible tools that could be utilized to *systematically* verify the correctness of the synthesis proposed by SNF.

Recent efforts have employed model checking (Canini et al., 2012; Kim et al., 2015a) techniques to explore the (voluminous) state space of modern networked systems in an attempt to find state inconsistencies due to etc. bugs or misconfigurations. Symbolic execution has also been utilized either alone (Kuzniar et al., 2012; Dobrescu and Argyraki, 2014) or combined with model checking (Canini et al., 2012), to systematically identify representative input events (i.e., packets) that can adequately exercise code paths without requiring to exhaust the input space (hence bound the verification time).

Specifically, Software Dataplane Verification (Dobrescu and Argyraki, 2014) is a close fit for verifying NFV service chains. The authors proposed a scalable approach to verifying complex NFV pipelines, by verifying each internal element of the pipeline in isolation; then by composing the results the authors proved certain properties about the entire pipeline. One could use this tool to systematically verify a complex part of SNF, which is the traffic classification. However, this tool might not be able to provide sound proofs regarding all the stateful modifications of SNF, since the authors verified only two simple stateful cases (i.e., a NAT and a traffic monitor) and did not generalize their ideas for a broader list of NFV flow modification elements.

SOFT (Kuzniar et al., 2012) could also be employed to test the interoperability between a chain realized with and without SNF. In other words, SOFT could inject a broad set of inputs to test whether the SynthesizedNF defined in § 3.1.3 outputs packets that are identical with the packets delivered by the original set of NFs. Similarly, HSA (Kazemian et al., 2012) could be used to verify loop-freedom, slice isolation, and reachability properties of SNF service chains. Unfortunately, HSA statically operates on a snapshot of the network configuration, hence is unable to track dynamic state modifications caused by continuous events. Similarly, SOFT is a special-purpose verification engine for software-defined networking (SDN) agent implementations. Therefore, both works require significant additional effort to verify stateful NFV pipelines.

Finally, translating an SNF processing graph into a finite state machine understandable by Kinetic (Kim et al., 2015a) would potentially allow Kinetic to use its model checker to verify certain properties for the entire pipeline. However, Kinetic does not systematically verify the actual code that runs in the network, but rather builds and verifies a model of this code. Therefore, we are concerned (*i*) whether a Kinetic model can sufficiently cover complex service chains such as the ISP-level chains presented in § 6.4 and (*ii*) whether Kinetic's located packet equivalence classes (LPECs) can handle the complex TCUs of SNF without causing state space explosion.

To summarize, although the works above have provided remarkable advancements in software verification, a substantial amount of additional research is required to provide strong guarantees about the correctness of SNF. For this reason, in this paper we focus our attention on delivering ultra high speed pipelines for complex and stateful NFV service chains and leave the verification of SNF as a future work.

LIMITATIONS

We do not attempt to provide a solution that can synthesize arbitrary software components, but rather target a broad but finite set of middlebox-specific NFs that operate on the entire space of a packet's header. SNF makes two assumptions:

1. An NFV provider must specify an NF as an ensemble of abstract packet processing elements (i.e., the NF DAG defined in § 3.2.1). We believe that this is a reasonable assumption, followed also by other state-of-the-art approaches such as Click, Slick, and OpenBox. However, if a middlebox provider does not want to share this information under non-disclosure or via a licensing agreement, SNF can synthesize the middleboxes before and after this provider's middlebox. This is possible by omitting the processing graph of this middlebox from the inputs given to the Service Chain Configurator (see § 3.2.1).
2. No further decision (i.e., read) utilizes an already rewritten field, therefore, an LB that splits traffic based on source port after a source NAPT, might not work. Similarly, in this case, SNF can exclude the LB from the synthesis.

Moreover, our tool does not support network-wide placement of the chain's components, but we envision SNF being integrated in controllers, such as E2 or Slick.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2477v2 | CC BY 4.0 Open Access | rec: 28 Sep 2016, publ: 28 Sep 2016

RELATED WORK

Over the last decade, there has been considerable evolution of software-based packet processing architectures that realize wireline throughputs, while providing flexible and cost effective in-cloud network processing.

Monolithic middlebox implementations. Until recently, most NFV approaches have treated NFs as monolithic entities placed at arbitrary locations in the network. In this context, even with the assistance of state-of-the-art OSs, such as the Click-based (Kohler et al., 2000) ClickOS (Martins et al., 2014) as well as fast network I/O (Rizzo, 2012; DPDK, 2016) and processing (Kim et al., 2012, 2015b; Barbette et al., 2015) mechanisms, chaining more than 2 NFs leads to serious performance degradation as stated by the authors of both ClickOS and NetVM (Hwang et al., 2014). The main reason, also shown in our experiments, for this poor performance is the I/O overhead due to forwarding packets along physically remote and virtualized NFs. More recently, OpenNetVM (Zhang et al., 2016) showed that VM-based NFV deployments do not scale with increasing number of chained instances, hence opted for NFs running in lightweight Docker (Docker, 2016) containers interconnected with shared memory segments.

Consolidation at the machine level. Concentrating network processing into a single machine is a logical way to overcome the limitations stated above. CoMb (Sekar et al., 2012) consolidates middlebox-oriented flow processing into one machine, mainly at the session layer. Similarly, OpenNF (Gember-Jacobson et al., 2014) provides a programming interface to migrate NFs, which can in turn be collocated in a physical server. DPIaaS (Bremler-Barr et al., 2014) reuses the costly deep packet inspection (DPI) logic across multiple instances. RouteBricks (Dobrescu et al., 2009) exploits parallelism to scale software routers across multiple servers and cores within a single server, while PacketShader (Han et al., 2010) and NBA (Kim et al., 2015b) take advantage of cheap and powerful auxiliary hardware components such as GPUs to provide fast packet processing. All of these works only partially exploit the benefits of sharing common middlebox functionality, thus they are far from supporting optimized service chains.

Consolidation at the individual function level is the next level of composition of scalable and efficient NF deployments. In this context, Open Middleboxes (xOMB) (Anderson et al., 2012) proposes an incrementally scalable network processing pipeline based on triggers that pass the flow control from one element to another in a pipeline. The xOMB architecture allows great flexibility in sharing parts of the pipeline; however, it only targets request-oriented protocols and services, unlike our generic framework.

Slick (Anwer et al., 2015) operates on the same level of packet processing as SNF to compose distributed, network-wide service chains driven by a controller. Slick provides its own programming language to achieve this composition and unlike our work, it addresses placement requirements. Slick is very efficient when deploying service chains that are not necessarily collocated. However, we argue that in many cases all the NFs of a service chain need to be deployed in one machine and effectively being dispatched across cores in the same socket. Slick does not allow all the NF elements to be physically placed into a single process. Our work goes beyond Slick by trading the flexibility of placing NF elements on demand for extensive consolidation of the chain processing. Our synthesized SNF realizes such chains with zero context switching and zero redundancy of individual packet operations.

Very recently, Bremler-Barr et al. (2016) applied the SDN control and dataplane separation paradigm to OpenBox; a framework for network-wide deployment and management of NFs. OpenBox applications input different NF specifications to the OpenBox controller via a north-bound application programming interface. The controller communicates the NF specifications to the OpenBox Instances (OBIs) that constitute the actual dataplane, ensuring smart NF placement and scaling. An interesting feature of the OpenBox controller is its ability to merge different processing graphs, from different NFs, into a single and shorter processing graph, similar to our SNF. The authors of OpenBox made a similar observation with us regarding the need to classify the traffic of a service chain only once, and then apply a set of operations that originate from the different NFs of the chain.

However, OpenBox does not highly optimize the result chain-level processing graph for two reasons: (i) The OpenBox merge algorithm can only merge homogeneous packet modification elements (i.e., elements with the same type). For example, two “Decrement IP TTL” elements, that each decrements the TTL field by one, can be merged into a single element that directly decrements the TTL field by two. Imagine, however, the case where OpenBox has to merge the NFs of Figure 5. In this example, OpenBox cannot merge the “Rewrite Flow” element (that modifies the source and destination IP addresses as well as the source port of UDP packets) with the 3 “Decrement IP TTL” elements, since these elements do not belong to the same type. This means that the final OpenBox graph will have 2 distinct packet modification

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2477v2 | CC BY 4.0 Open Access | rec: 28 Sep 2016, publ: 28 Sep 2016

elements (i.e., 1 “Rewrite Flow” and 1 “Decrement IP TTL”) and each element has to compute the IP and UDP checksums separately. *Therefore, OpenBox does not completely eliminate redundant operations.* In contrast, SNF effectively synthesized the operations of all these elements into a *single* element (see Figure 6) that computes the IP and UDP checksums only once. Consequently, SNF produces both a *shorter* processing graph and a synthesized chain with *no redundancy*, hence achieving lower latency.

(*ii*) Although OpenBox can merge the classification elements of a chain into a single classifier, the authors have not addressed how they handle the increased complexity of the final classifier. Our preliminary experiments showed that in complex use cases, such as the ISP-level traffic classification presented in § 6.4, the complexity of the chain-level classifier dramatically increases with increasing number of ACL rules. Therefore, SNF implements the lazy subtraction optimization proposed by Kazemian et al. (2012). The benefits of this algorithm are stated in § 5.1.

Finally, the authors of OpenBox did not stress the limits of the OpenBox framework in their performance evaluation. An input packet rate of 1-2 Gbps cannot adequately stress the memory utilization of the OBIs. Moreover, there is limited discussion related to how OpenBox exploits the multi-core capacities of modern NFV infrastructures. In contrast, in § 6.2, § 6.3, and § 6.4 we demonstrated how SNF realizes complex, purely software-based service chains at 40 Gbps line-rate. This is possible by exploiting multiple CPU cores and by fitting most of the data of an entire service chain into those cores’ L1 caches.

Scheduling NFs for high throughput. Recently, the E2 framework (Palkar et al., 2015) demonstrated a scalable way of deploying NFV services. E2 mainly tackles placement, elastic scaling, and service composition by introducing pipelets. A pipelet defines a traffic class and a corresponding DAG of NFs that should process this traffic class. SNF’s TCUs are somewhat similar to E2’s pipelets but SNF aims to make them more efficient. Concretely, an SNF TCU is not processed by a DAG of NFs, but rather by a highly optimized piece of code (produced by the synthesizer) that directly applies a set of operations to this specific traffic class.

Impact. E2 can use SNF to fit more service chains into one machine, hence postpone its elastic scaling. Existing approaches can transparently use our extensions to provide services such as (*i*) lightweight Xen VMs that run synthesized ClickOS instances using the netmap network I/O, (*ii*) parallelized service chains using the multi-server, multi-core RouteBricks architecture, and (*iii*) synthesized chains that are load balanced across heterogeneous hardware components (i.e., CPU and GPU) using NBA.

CONCLUSION

We have addressed the problem of synthesizing chains of NFs with SNF. SNF requires minimal I/O interactions with the NFV platform and applies single-read-single-write operations on the packets, while early-discarding irrelevant traffic classes. SNF maintains state across NFs.To realize the above properties, we parse the chained NFs and build a classification graph whose leaves represent unique traffic class units. In each leaf we perform a set of packet header modifications to generate an equivalent configuration that implements the same functionality as the initial chain using a minimal set of elements.

SNF synthesizes stateful chains that appear in production ISP-level networks realizing high throughput and low latency, while outperforming state-of-the-art works.

REFERENCES

Anderson, J. W., Braud, R., Kapoor, R., Porter, G., and Vahdat, A. (2012). xOMB: Extensible Open Middleboxes with Commodity Servers. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS ’12, pages 49–60, New York, NY, USA. ACM.

Anwer, B., Benson, T., Feamster, N., and Levin, D. (2015). Programming Slick Network Functions. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR ’15, pages 14:1–14:13, New York, NY, USA. ACM.

Bagnulo, M., Matthews, P., and van Beijnum, I. (2011). Stateful NAT64: Network Address and Protocol Translation from IPv6 Clients to IPv4 Servers. RFC 6146 (Proposed Standard).

Barbette, T., Soldani, C., and Mathy, L. (2015). Fast Userspace Packet Processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS ’15, pages 5–16, Washington, DC, USA. IEEE Computer Society.

Bremler-Barr, A., Harchol, Y., and Hay, D. (2016). OpenBox: A Software-Defined Framework for

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2477v2 | CC BY 4.0 Open Access | rec: 28 Sep 2016, publ: 28 Sep 2016

Developing, Deploying, and Managing Network Functions. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, pages 511–524, New York, NY, USA. ACM.

Bremler-Barr, A., Harchol, Y., Hay, D., and Koral, Y. (2014). Deep Packet Inspection as a Service. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 271–282, New York, NY, USA. ACM.

Canini, M., Venzano, D., Perešíni, P., Kostić, D., and Rexford, J. (2012). A NICE Way to Test Openflow Applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 10–10, Berkeley, CA, USA. USENIX Association.

Cisco (2014). Scaling NFV - The Performance Challenge. http://blogs.cisco.com/enterprise/scaling-nfv-the-performance-challenge.

Dobrescu, M. and Argyraki, K. (2014). Software Dataplane Verification. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 101–114, Berkeley, CA, USA. USENIX Association.

Dobrescu, M., Argyraki, K., Iannaccone, G., Manesh, M., and Ratnasamy, S. (2010). Controlling Parallelism in a Multicore Software Router. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '10, pages 2:1–2:6, New York, NY, USA. ACM.

Dobrescu, M., Egi, N., Argyraki, K., Chun, B.-G., Fall, K., Iannaccone, G., Knies, A., Manesh, M., and Ratnasamy, S. (2009). RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 15–28, New York, NY, USA. ACM.

Docker (2016). Docker Containers. https://www.docker.com/.

DPDK (2016). Data Plane Development Kit (DPDK). http://dpdk.org.

Emmerich, P., Gallenmüller, S., Raumer, D., Wohlfart, F., and Carle, G. (2015). MoonGen: A Scriptable High-Speed Packet Generator. In *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*, IMC '15, pages 275–287, New York, NY, USA. ACM.

Enguehard, M. (2016). Hyper-NF: synthesizing chains of virtualized network functions. *Master Thesis, KTH School of Information and Communication Technology (ICT)*. http://urn.kb.se/resolve?urn=urn%3Anbn%3Ase%3Akth%3Adiva-180397.

European Telecommunications Standards Institute (2012). NFV Whitepaper. https://portal.etsi.org/NFV/NFV_White_Paper.pdf.

Evans, D. (2011). The internet of things: How the next evolution of the internet is changing everything. *Cisco Internet Business Solutions Group (IBSG)*, pages 1–11. https://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf.

Gember-Jacobson, A., Viswanathan, R., Prakash, C., Grandl, R., Khalid, J., Das, S., and Akella, A. (2014). OpenNF: Enabling Innovation in Network Function Control. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 163–174, New York, NY, USA. ACM.

Han, S., Jang, K., Park, K., and Moon, S. (2010). PacketShader: A GPU-accelerated Software Router. *SIGCOMM Comput. Commun. Rev.*, 40(4):195–206.

Hwang, J., Ramakrishnan, K. K., and Wood, T. (2014). NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 445–458, Berkeley, CA, USA. USENIX Association.

Intel (2016). Receiver-Side Scaling (RSS). http://www.intel.com/content/dam/support/us/en/documents/network/sb/318483001us2.pdf.

Katsikas, Georgios (2016). SNF extensions of FastClick's stateful flow processing elements. https://github.com/gkatsikas/fastclick/tree/snf.

Kazemian, P., Varghese, G., and McKeown, N. (2012). Header Space Analysis: Static Checking for Networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 9–9, Berkeley, CA, USA. USENIX Association.

Kim, H., Reich, J., Gupta, A., Shahbaz, M., Feamster, N., and Clark, R. (2015a). Kinetic: Verifiable Dynamic Network Control. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 59–72, Berkeley, CA, USA. USENIX Association.

Kim, J., Huh, S., Jang, K., Park, K., and Moon, S. (2012). The Power of Batching in the Click Modular Router. In *Proceedings of the Asia-Pacific Workshop on Systems*, APSYS '12, pages 14:1–14:6, New York, NY, USA. ACM.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2477v2 | CC BY 4.0 Open Access | rec: 28 Sep 2016, publ: 28 Sep 2016

Kim, J., Jang, K., Lee, K., Ma, S., Shim, J., and Moon, S. (2015b). NBA (Network Balancing Act): A High-performance Packet Processing Framework for Heterogeneous Processors. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 22:1–22:14, New York, NY, USA. ACM.

Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. (2000). The Click Modular Router. *ACM Trans. Comput. Syst.*, 18(3):263–297.

Kuzniar, M., Peresini, P., Canini, M., Venzano, D., and Kostic, D. (2012). A SOFT Way for Openflow Switch Interoperability Testing. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 265–276, New York, NY, USA. ACM.

Liu, W., Li, H., Huang, O., Boucadair, M., Leymann, N., Fu, Q., Sun, Q., Pham, C., Huang, C., Zhu, J., and He, P. (2014). Service Function Chaining (SFC) General Use Cases. Internet-Draft draft-liu-sfc-use-cases-08, IETF Secretariat. https://tools.ietf.org/html/draft-liu-sfc-use-cases-08.

Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R., and Huici, F. (2014). ClickOS and the Art of Network Function Virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 459–473, Berkeley, CA, USA. USENIX Association.

McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. (2008). OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74.

Palkar, S., Lan, C., Han, S., Jang, K., Panda, A., Ratnasamy, S., Rizzo, L., and Shenker, S. (2015). E2: A Framework for NFV Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 121–136, New York, NY, USA. ACM.

Penno, R., Wing, D., and Boucadair, M. (2013). PCP Support for Nested NAT Environments. Internet-Draft draft-penno-pcp-nested-nat-03, IETF Secretariat. https://tools.ietf.org/html/draft-penno-pcp-nested-nat-03.

Perreault, S., Yamagata, I., Miyakawa, S., Nakagawa, A., and Ashida, H. (2013). Common Requirements for Carrier-Grade NATs (CGNs). RFC 6888 (Best Current Practice).

Quinn, P. and Nadeau, T. (2015). Problem Statement for Service Function Chaining. RFC 7498 (Informational).

Rizzo, L. (2012). Netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 9–9, Berkeley, CA, USA. USENIX Association.

SDX Central (2015). Performance - Still Fueling the NFV Discussion. https://www.sdxcentral.com/articles/contributed/vnf-performance-fueling-nfv-discussion-kelly-leblanc/2015/05/.

Sekar, V., Egi, N., Ratnasamy, S., Reiter, M. K., and Shi, G. (2012). Design and Implementation of a Consolidated Middlebox Architecture. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 24–24, Berkeley, CA, USA. USENIX Association.

Sherry, J., Hasan, S., Scott, C., Krishnamurthy, A., Ratnasamy, S., and Sekar, V. (2012). Making Middleboxes Someone else's Problem: Network Processing As a Cloud Service. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 13–24, New York, NY, USA. ACM.

Sun, W. and Ricci, R. (2013). Fast and Flexible: Parallel Packet Processing with GPUs and Click. In *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '13, pages 25–36, Piscataway, NJ, USA. IEEE Press.

Taylor, D. E. and Turner, J. S. (2007). ClassBench: A Packet Classification Benchmark. *IEEE/ACM Trans. Netw.*, 15(3):499–511.

Woo, S. and Park, K. (2012). Scalable TCP Session Monitoring with Symmetric Receive-side Scaling. KAIST Technical Report. pages 1–7.

Zhang, W., Liu, G., Zhang, W., Shah, N., Lopreiato, P., Todeschi, G., Ramakrishnan, K., and Wood, T. (2016). OpenNetVM: A Platform for High Performance Network Service Chains. In *Proceedings of the 2016 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. ACM.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2477v2 | CC BY 4.0 Open Access | rec: 28 Sep 2016, publ: 28 Sep 2016