Realizing High Performance NFV Service Chains

GEORGIOS P. KATSIKAS

Licentiate Thesis in Information and Communication Technology
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden 2016

KTH School of Information and
Communication Technology
SE-164 40 Kista
SWEDEN

TRITA-ICT 2016:35
ISBN 978-91-7729-163-3

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av licentiatexamen i Informations- och kommunikationsteknik tisdagen den 6 december 2016 klockan 13:00 i Sal C, Electrum, Kungl Tekniska högskolan, Kistagången 16, Kista.

© Georgios P. Katsikas, December 2016

Tryck: Universitetsservice US AB

Abstract

Network functions (NFs) hold a key role in networks, offering in-network services, such as enhanced performance, policy enforcement, and security. Traditionally, NFs have been implemented in specialized, thus expensive hardware. To lower the costs of deploying NFs, network operators have adopted network functions virtualization (NFV), by migrating NFs from hardware to software running in commodity servers.

Several approaches to NFV have shown that commodity network stacks and drivers (e.g., Linux-based) struggle to keep up with increasing hardware speed. Despite this, popular networking services still rely on these commodity components. Moreover, chaining NFs (also known as service chaining) is challenging due to redundancy in the elements of the chain. This licentiate thesis addresses the performance problems of NFV service chains.

The first contribution is a framework that (*i*) profiles NFV service chains to uncover performance degradation reasons and (*ii*) leverages the profiler's data to accelerate these chains, by combining multiplexing of system calls with scheduling strategies. These accelerations improve the cache utilization and thereby the end-to-end latency of chained NFs is reduced by a factor of three. Moreover, the same chains experience a multi-fold latency variance reduction; this result improves the quality of highly-interactive services.

The second contribution of this thesis substantially revises the way NFV service chains are realized. NFV service chains are synthesized while eliminating redundant input/output and repeated elements, providing consolidated stateful cross layer packet operations across the chain. This software-based synthesis achieves line-rate 40 Gbps throughput for stateful and long service chains. This performance is 8.5x higher than the performance achieved by the software-based state of the art FastClick framework. Experiments with three example Internet Service Provider-level service chains show that this synthesis approach operates at 40 Gbps, when the classification of these chains is offloaded to an OpenFlow switch.

Keywords: NFV, service chains, profiler, scheduling, multiplexing, synthesis, line-rate, 40 Gbps.

Sammanfattning

Nätverksfunktioner (NF) har en nyckelroll i nätverk. De erbjuder tjänster i nätverken som förbättrad prestanda, policy övervakning och säkerhetsfunktioner. Vanligtvis så har NF implementerats med hjälp av specialiserad, och därmed kostsam, hårdvara. Detta har lett till att nätverksoperatörer har börjat använda nätverksfunktionsvirtualisering (NFV) för att minska kostnaden. NFV implementeras genom att NF flyttas från specialiserad hårdvara till mjukvara som kör på vanliga servrar.

Flera försök med NFV har visat att vanliga nätverksstackar och drivrutiner (exempelvis Linux baserade) har svårt att erbjuda samma prestanda som hårdvaran gör. Trots detta bygger flera populära nätverkstjänster på NFV. Dessutom är det en utmaning att koppla samman NFV i kedjor, då redundanta operationer utförs. I den här avhandlingen försöker vi lösa prestanda problem kopplade till kedjor av NFV.

Det första bidraget i den här avhandlingen är ett ramverk som (*i*) profilerar NFV kedjor för att hitta orsaker till prestanda problem samt (*ii*) använder profileringsdata för att förbättra prestandan i kedjorna. Detta görs genom att kombinera multiplexing av systemanrop med planläggningsstrategier. Tillsammans förbättrar dessa lösningar cache användningen och minskar därmed end-to-end latensen i kedjade NFV med en faktor tre. Dessutom minskar vår metod variansen i latens, något som är viktigt för tjänstekvalitén i interaktiva tjänster.

Det andra bidraget i den här avhandlingen är en omarbetning av hur kedjade NFV konstrueras. Vi syntetiserar NFV service kedjor genom att ta bort redundanta element och konsoliderar paketoperationer som sträcker sig över flera lager i nätverksstacken. Vår mjukvarubaserade lösning klarar av 40 Gbps genomströmning i en lång kedja. Detta är 8.5 ggr mer än vad som uppnåtts med den tidigare standard lösningen för mjukvara, ramverket FastClick. Vi presenterar experiment med tre servicekedjor för nätverksleverantörer där vår syntetiserade lösning klarar 40 Gbps, när klassificeringen av kedjan görs med hjälp av en OpenFlow switch.

Nyckelord: NFV, service kedjor, profilering, multiplexing, planläggningsstrategier, syntetiserade, 40 Gbps.

Acknowledgements

I am first and foremost thankful to my advisors and mentors, Professors Dejan Kostić and Gerald Q. Maguire Jr. They have provided to me all the means, great ideas, and timely feedback to conduct research on a very interesting and increasingly popular area. Meeting and brainstorming with people of this spiritual level is priceless.

I would also like to thank Marcel Enguehard for sharing with me the pain to realize one of the contributions of this licentiate. We worked hard together in absolute harmony to produce the first working prototype of SNF. Maciej Kuzniar played also an important role in realizing the hardware-assisted version of SNF. He is definitely an OpenFlow master. We had to "run a marathon" to meet the OSDI 2016 deadline, while pushing SNF to the limit! Pehr Söderman gave me useful feedback regarding the abstract of this licentiate thesis and he also sacrificed some of his hunting time to translate the abstract to Swedish.

Associate Professor Markus Hidell was the internal reviewer of my licentiate proposal and the advance reviewer of this licentiate thesis. I would like to thank him very much for his patience and excellent feedback during the last nine months.

Kirill Bogdanov is my colleague and the person that shares the same office, whiteboard, ambitions, and chocolates with me. He contributed to this licentiate by tolerating me for two years and by refactoring some ugly parts of my code.

Last but not least, my family and friends deserve a special mention. Their unconditional support over these years has been crucial. Without it, this licentiate thesis would never have been written.

Georgios P. Katsikas,
Stockholm, November 3, 2016

The research leading to these results has been co-funded by the European Union (EU) in the context of (*i*) the European Research Council (ERC) under EU's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement 259110 and (*ii*) the BEhavioural BAsed forwarding (BEBA) project with grant agreement number 644122.

Contents

List of Figures

List of Tables

List of Acronyms

ACL	Access Control List. 92, 107, 110, 111, 114, 115, 117, 119, 121, 123, 128
API	Application Programming Interface. 4, 12, 15, 17, 27, 128
B2B	Back-to-Back. 32, 33, 52, 53, 85, 87, 88, 132
CFS	Completely Fair Scheduler. 55, 77, 78, 83–87, 90
CPU	Central Processing Unit. 11, 12, 14, 15, 18–21, 26, 32–34, 36, 39, 43, 44, 51, 54, 56, 58, 60, 62, 65, 66, 69, 70, 73–75, 77–80, 84, 85, 87–89, 91, 92, 103, 105, 106, 108, 110, 111, 123, 129, 137, 140
CRC	Cyclic Redundant Check. 33, 62, 108
DAG	Directed Acyclic Graph. 47, 94, 96–99, 102, 105, 129, 135
DDIO	Direct Data I/O. 19, 56, 61, 65
DMA	Direct Memory Access. 11, 13, 14, 18, 19, 65
DPDK	Data Plane Development Kit. 9, 16, 21, 24–26, 33, 34, 36, 38, 44, 53, 62–68, 82, 106, 114, 119, 126, 131, 132, 136
DPI	Deep Packet Inspection. 6, 27, 127
DRAM	Dynamic Random Access Memory. 60, 61
DTLB	Data Translation Lookaside Buffer. 56, 60–62, 64, 65

FIFO First In, First Out. 55, 78, 79
FW Firewall. 3, 6, 47, 102, 110, 111, 115, 117

GbE Gigabit Ethernet. 31, 35, 36, 44, 62, 108, 136, 137
GPU Graphics Processing Unit. 26, 127, 129

HSA Header Space Analysis. 27, 107, 125

I/O Input/Output. 8, 9, 15–17, 19–21, 24–26, 33, 34, 36, 38, 39, 45, 47, 50–55, 62, 64, 65, 67, 70–74, 78–81, 83–88, 90, 94, 98, 102, 103, 105, 106, 111, 114, 126, 129, 131, 132, 136, 140, 141, 143
IP Internet Protocol. 2, 3, 6, 14, 24, 26, 47, 94, 96, 101, 103–107, 109, 111, 121, 128
IPv4 Internet Protocol version 4. 3, 95, 100, 109
IPv6 Internet Protocol version 6. 109
ISP Internet Service Provider. 2, 6, 9, 92, 107, 109–111, 115, 118, 121, 123, 125, 128, 132, 133, 140, 143

JSON JavaScript Object Notation. 72, 73

LB Load Balancer. 3, 6, 47, 102, 136
LLC Last Level Cache. 18–20, 56, 59, 61, 65, 66, 68, 89

MAC Medium Access Control. 11, 47, 96
Mpps Millions of Packets Per Second. 33, 62, 80, 114

NAPT Network Address and Port Translator. 3, 6, 47, 92, 96, 100–102, 107, 109–111, 115, 117, 121, 132, 136
NAT Network Address Translator. 109, 125

NF	Network Function. 3, 5, 6, 8, 9, 20, 23–27, 32–34, 36, 38–40, 43, 45, 47, 49, 50, 52, 54, 60–63, 66, 69, 70, 72–75, 80, 83–87, 90–92, 94–103, 105, 108, 110, 111, 114, 117–119, 121, 125–129, 132, 135, 136, 141, 143
NFV	Network Functions Virtualization. 5–11, 20, 23–28, 31–40, 43–45, 49–52, 54–58, 60–62, 67, 68, 71–76, 78–80, 82, 83, 85, 89–91, 100, 105–108, 119, 121, 125–127, 129, 131–133, 135, 136, 139–141, 143
NIC	Network Interface Card. 11–14, 16, 18–21, 25, 31, 33, 34, 36, 43–45, 49–51, 54, 61, 62, 65, 68, 90, 92, 93, 106, 108–111, 118, 119, 136, 137
NUMA	Non-Uniform Memory Access. 18, 20, 106
OS	Operating System. 8, 11–16, 20, 24, 26, 28, 29, 31, 38, 43, 50–52, 54, 55, 61, 65, 70, 71, 73, 75, 83, 89, 90, 126, 137, 143
OVS	Open vSwitch. 25, 68
OVSK	Kernel-based Open vSwitch. 32, 33, 52, 53, 62, 63, 67–69, 72, 84, 85, 87
PCM	Performance Counter Monitoring. 55, 60, 135
PID	Process Identifier. 54, 73, 75
PMU	Performance Monitoring Unit. 29, 89
pps	Packets Per Second. 20, 24, 74, 114
PU	Processing Unit. 96, 99, 100
RR	Round-Robin. 55, 78, 79, 83
RSS	Receive-Side Scaling. 93, 106, 108, 111
Rx	Reception. 14, 16, 20, 25, 26, 45, 65
SCC	Service Chain Coordinator. 50, 52, 54, 55, 58, 60–62, 64–76, 79, 80, 82–89, 131–133, 135, 136, 140, 141, 143
SDN	Software Defined Networking. 5–8, 24–28, 125, 127

skbuff Socket Buffer. 14, 17, 54, 58, 66, 67

SNF Synthesized Network Functions. 91–94, 96–103, 105–112, 114, 115, 117–119, 121, 123, 125–129, 131–133, 135, 136, 140, 141, 143

TCP Transmission Control Protocol. 14, 28, 44, 95, 102, 103, 106, 118

TCU Traffic Class Unit. 92, 93, 96, 99, 100, 106, 107, 111, 115, 125, 129

TLB Translation Lookaside Buffer. 56, 89

TTL Time To Live. 24, 94, 96, 103–106, 128

Tx Transmission. 14, 16, 20, 25, 26, 45, 66, 67

UDP User Datagram Protocol. 28, 44, 102–106, 118, 128

VM Virtual Machine. 8, 26, 34, 50, 60, 129, 141

Chapter 1

Introduction

Human beings are, by nature, social animals as defined by Aristotle in Politics [1]. As such, over the centuries human beings have developed ways to communicate. For example, the wheel was a radical invention that, among its various contributions, has facilitated people's movement between physically remote locations (i.e., by using the wheel as a component to produce carriages). More recently, over the 20^{th} century, the need for communication was further facilitated by numerous technological advancements, such as telecommunications.

> Telecommunications: The electronic transmission of information over distances.
>
> — Gupta and Sharma [2]

At the very beginning, telecommunications allowed the transportation of voice, allowing people in different places to talk to each other. This comfort inspired people to generalize telecommunications, allowing the exchange of any kind of information, such as data, text, images, etc. Therefore, in the second half of the 20^{th} century, telecommunications together with the digital revolution and the concomitant emergence of computers, allowed people to organize remote computer systems into telecommunications networks.

> Telecommunications network: An arrangement of computing and telecommunications resources for communication of information between distant locations.
>
> — Gupta and Sharma [2]

1.1 Today's Telecommunications Landscape

The 20th century established the background of a global telecommunications network widely known as Internet. Over the first two decades of the 21st century, a tremendous amount of information has been produced and exchanged among users across the globe. This voluminous information exchange is, in turn, pushing the scale and dynamics of telecommunications networks to extremes. Social, mobile and wireless communications, sensor-driven networks in machine-to-machine applications, the emergence of novel applications (e.g., high-quality video streams), the increased numbers and varieties of devices (e.g., smartphones, smartwatches), and the advent of future generation communication technologies (i.e., fifth generation networks) have substantially grown the Internet, contributing to an ever-increasing load upon telecommunications networks.

A forecast from Cisco's Visual Networking Index [3] offers interesting findings regarding the evolution of the telecommunications as a result of near future traffic expectations:

1. Annual global Internet Protocol (IP) traffic will pass the *Zettabyte (1000 Exabytes)* threshold by the end of 2016, and will reach 2.3 Zettabytes per year by 2020;

2. The number of devices connected to IP networks will be *more than 3 times the global population* by 2020; and

3. Every second, nearly a *million minutes of video content* will cross the network by 2020.

At the same time, IBM's Institute for Business Value [4] investigated the effects of this phenomenon on the industry. The key conclusion is that the historically common connection between traffic and revenue has been blurred because of the increases in data volumes due to multimedia content and other sources of traffic require investments in infrastructure, but at the same time revenues from voice calls are rapidly declining.

1.2 Network Functions: A Blessing and a Curse

Despite the trends described above, telecommunications' stakeholders such as network operators, Internet Service Providers (ISPs), and content providers are striving to preserve their share of the market. To achieve this, they have to attract customers by offering services that satisfy the customer's quality requirements. Some key quality factors for services in telecommunications networks are latency, throughput, and security. The importance of these factors has been reported

by service providers; for example Amazon [5] reported that a latency increase of 100 ms causes 1% loss in their sales. Verizon promotes services by cleverly advertising how their networks combine high bandwidth with high throughput [6]. Moreover, market studies [7] showed that security is of utmost importance for industrial information technology stakeholders, hence information technology security services faced significant growth between 2009 and 2015.

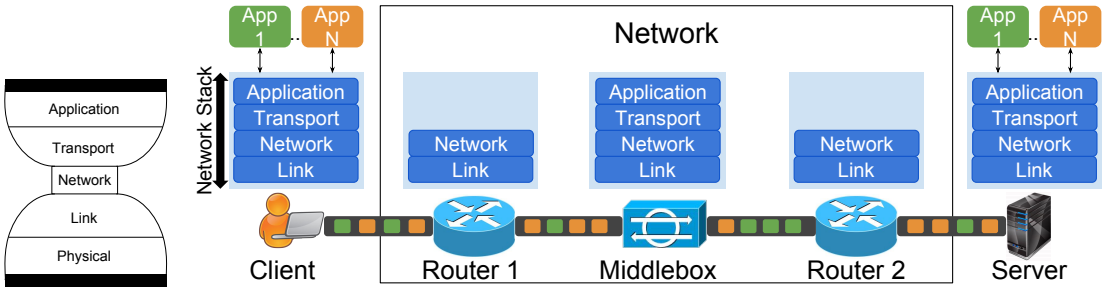
A means for telecommunications' stakeholders to increase the quality of the provided services is through carefully placing Network Functions (NFs) in the network. An NF takes packets at a network element's input port(s), and outputs (potentially modified) packets from the same network element's output port(s). Based on Internet Engineering Task Force's request for comments 3234 [8], NFs are separated into two main categories: (*i*) routing/forwarding and (*ii*) all the remaining functions within the network layer and above.

1.2.1 Benefits of Network Functions

The first category of NFs is realized by IP routers, whose only functions are to determine routes and forward packets, while also updating fields that are necessary for this forwarding process. Route selection functions offer great benefits, such as low latency by selecting e.g., the shortest path between a given source and destination networks. Another advantage is that these functions are simple and keep the network layer thin, because this functionality has to be implemented by all the devices along a given source-destination path. As a result, a variety of upper layer protocols run atop the IP layer. IP packets are forwarded as frames over different hardware layers. This is the well-known hourglass [8] model, with IP over everything and everything over IP, as depicted in Figure 1.1a.

Simple NFs were not sufficient to accommodate the rapid evolution and high penetration of the Internet. Therefore, network operators sought additional network functionality to increase security, isolation, and network performance. As a result, a second category of NFs, also known as middleboxes, arose. According to request for comments 3234 [8], these advanced functions (deeply and often statefully) inspect and modify the packets' structure, modifying fields across the entire header, and in some cases even examining and modifying the packets' payload. Middlebox functionality offers valuable benefits by:

1. allowing reuse of portions of the Internet Protocol version 4 (IPv4) address space via Network Address and Port Translators (NAPTs);
2. network resource optimization using e.g., Load Balancers (LBs) or wide area network optimizers; and
3. increasing security using e.g., Intrusion Detection System and Firewall (FW) middleboxes.



(a) The hourglass model. **(b)** Traffic traversing the network stacks of end hosts and network elements.

Figure 1.1: End-to-end view of today's networks.

1.2.2 Problems of Network Functions

Traditionally, middleboxes have been seen as parts of the network fabric and have mostly been implemented in specialized hardware. According to a 2012 study of 57 enterprise networks [9], the typical number of middleboxes in an enterprise network is comparable to the number of switches and routers, while the proliferation of smartphones and wireless video streaming are expected to further expand the range of such middleboxes. Consequently, traditional middleboxes pose the following challenges:

1. Specialization comes at great cost, as the evolution of hardware follows Moore's Law [10]. Hence, continuous infrastructure replacements of specialized hardware are required by the network operators to scale the network up;

2. Instead of concentrating middleboxes at the edges and keeping the network layer thin, network operators have deployed middleboxes "on path" at key network choke points, as shown in Figure 1.1b. As a result:

   2 a) network complexity has increased, hence the hourglass model shown in Figure 1.1a does not hold anymore;

   2 b) traffic steering interferes with routing, hence it requires manual effort and expertise. Despite recent research contributions [11], production networks still rely on middleboxes from different vendors that expose proprietary Application Programming Interfaces (APIs), thus posing complex management requirements [12];

3. Over the past 5 years, large enterprises had to budget an additional $1 million dollars in order to maintain their middleboxes [9]; and

4. Network administrators need to over-provision these (hardware-based) middleboxes to accommodate the peak load. Moreover, there is no ability to dynamically scale-in/out the entire network [9].

1.3 Network "Softwarization"

In an attempt to keep up with the changing landscape in the global telecommunications market, telecommunications' stakeholders need to address the challenges listed in § 1.2.2, by lowering the deployment, maintenance, and management costs of NFs. Therefore, software has become a first class component in modern networks, shifting telecommunications stakeholders' focus towards Software Defined Networking (SDN) and Network Functions Virtualization (NFV) [13].

SDN

The idea of SDN is to decouple the control logic from the fast traffic path by remotely configuring a device's flow tables using a logically centralized controller. This allows network administrators to take and enforce network-wide decisions, by gaining control of the logic of simple NFs such as a switches and routers. This is achieved by moving the computation of routes (for routers) and next hops (for switches) to a software-based SDN controller. SDN adoption has gained momentum since the introduction of the OpenFlow [14] protocol in 2008, with campus [15], wide area network [16, 17, 18], and datacenter [19] deployments gradually replacing traditional network designs.

NFV

Building the network state in the control plane using SDN is not always effective. Advanced NFs might require dynamically handling of the flow state directly in the data plane. Often, this requirement is combined with complex packet operations that take place on the packets' payload. As discussed in § 1.2, these functions have been traditionally offered by hardware-based middleboxes. NFV migrates middlebox functionality from hardware to software, running in commodity off-the-shelf servers, potentially eliminating most of the challenges posed by traditional middleboxes (see § 1.2.2). Although early NFV deployments have recently begun [20, 21], as we will see later on, NFV comes with its own problems and challenges.

1.4 Network Functions' Composition for Service Chains

Modern services require combinations of NFs, also known as service chains, to satisfy their quality requirements [22]. Service chains commonly appear in networks. For example, as shown in Figure 1.2, when a user at his/her home network requests a web page via the Hypertext Transfer Protocol, his/her traffic might traverse a service chain before reaching the destination. Along the path from the user to the server that hosts the web page, a NAPT might be present at the local ISP's network, a set of core routers to forward user's traffic across the Internet, while a FW, a Deep Packet Inspection (DPI), and an LB might all be residing before a server at the target datacenter [23].

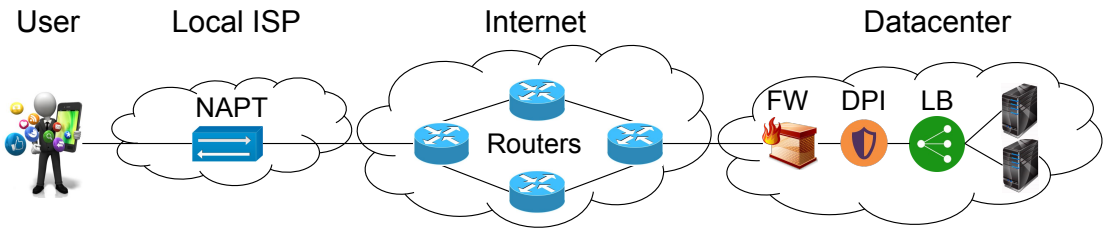


Figure 1.2: An example service chain between a user connected to a local ISP and a service hosted by a server farm in a datacenter. The chain consists of a NAPT, several core routers, a FW, a DPI function, and an LB.

The service chain illustrated in Figure 1.2 offers the following benefits: First, the NAPT helps the ISP of the user's access network to conserve public IP addresses by multiplexing multiple users' traffic into traffic that uses a single public IP address. Second, the core routers route traffic along the shortest path between the local ISP's network and the closest datacenter that hosts the web page. Third, the FW drops illegitimate traffic attempting to enter the target datacenter. The DPI adds another layer of security by flagging malicious content in that traffic. Finally, the LB ensures that legitimate user requests will be served as soon as possible, by selecting the least overloaded server.

By offloading middlebox functionality to the cloud, telecommunications' stakeholders can realize the example service chain shown in Figure 1.2 in software, using SDN and NFV. As depicted in Figure 1.3, a system administrator can specify such service chain as a high-level policy that targets the green portion of the traffic (legitimate requests). At first, this high-level policy will trigger the dynamic instantiation of the appropriate (number and types of) NFV instances that comprise this service chain. Then, the SDN controller will translate the stated policy into low-level SDN rules (e.g., using the OpenFlow protocol), that

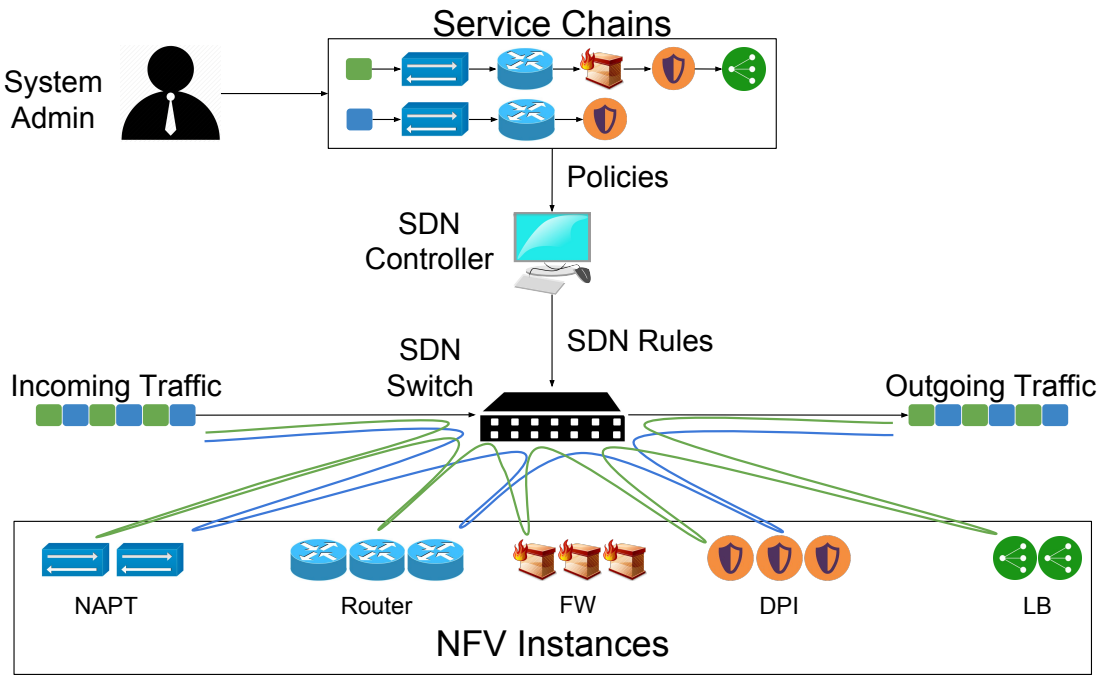


Figure 1.3: An example set of service chains translated by an SDN controller into SDN rules that steer the traffic through multiple NF instances.

need to be installed in the underlying SDN switch, to forward the green portion of the traffic through the required NFV instances. If another service chain needs to be collocated with this example service chain (i.e., the blue portion of the traffic), then the NFV framework will ensure that new instances will be deployed in isolation and new SDN rules will guide the blue packets through that chain. The software-driven paradigm depicted in Figure 1.3 also ensures on demand resource provisioning, by tearing down the NFV instances of these chains, once the traffic has been served.

1.5 High-level Research Challenges

Despite the advances in SDN and NFV, there are still several challenges that need to be addressed. The foremost challenge is to make software-based packet processing perform as well as its hardware counterpart (i.e., the hardware implementation of the middlebox device). Despite the dramatic evolution of high-speed computing during the last decades, this is still a hard target to achieve. To further clarify this, let us look again at the service chains shown in Figure 1.3.

First of all, this model implies that by offloading the NFs to the cloud, these functions will no longer be in the direct path between two communicating endpoints, as per the traditional approach shown in Figure 1.2. In fact, traffic has to be sent on a path through the cloud, leading to a potential increase in the end-to-end packet latency.

Secondly, these software-based service chains imply that traffic has to move from the SDN switch to different processes (i.e., NFs), running on bare metal or in Virtual Machines (VMs), multiple times along the end-to-end path between the user and a server in a datacenter. In such a path, multiple Input/Output (I/O) and processing operations will occur for the switch and each process, hence the services' performance might be seriously affected by the presence of the example service chains. Consequently, a true challenge in realizing high-performance NFV service chains is to ensure low latency and high throughput, while maintaining the cost and management benefits offered by NFV and SDN.

Next, we present the outline of this thesis that paves the way to: (*i*) experimentally verify and refine the challenges above (see § 4) and (*ii*) provide our original contributions to the literature to address these research challenges (see § 6 and § 7).

1.6 Summary of Contributions

The challenges discussed in § 1.5 motivated the work for this thesis. The first target of this thesis was to measure the performance of state of the art NFV frameworks when executing service chains, in order to verify that the challenges exist and then derive real scientific problems from these challenges. To do so, I designed and implemented a fully-controllable NFV experimental testbed that comprises of state of the art hardware & software components and techniques. The details of this experimental testbed are given in Chapter 5. Using this experimental testbed, I conducted the research that led to the contributions described below.

1.6.1 Contribution 1

First, I studied the performance of service chains using a state of the art NFV framework on top of the Linux Operating System (OS) and the standard Linux network drivers. In this study, I designed and implemented an NFV profiler (see § 6.2) that can thoroughly monitor low-level performance counters, during the execution of NFV service chains. The goal of this profiler was to uncover the reasons that cause these service chains to exhibit high latency with an increasing length of a chain. The results of the profiler uncovered I/O and scheduling overheads that linearly increase the per packet latency with an increasing length

of the chain (see § 6.3). To mitigate these overheads, I built a tool that accelerates user-space NFV service chains by combining custom scheduling policies with I/O multiplexing techniques (see § 6.4). Then, I evaluated the effects of my tool on the performance of NFV service chains using both single and multi-core deployment scenarios (see § 6.5). In summary, service chains that use my tool achieved a multi-fold latency and latency variance reduction compared to a baseline approach. A summary of my results is given in Table 6.1.

To the best of my knowledge, this is the first NFV profiler in the literature. Moreover, it is the first time that I/O multiplexing and scheduling techniques are combined to improve the performance of NFV service chains. The originality of this work with respect to earlier relevant efforts is described in § 6.6.

1.6.2 Contribution 2

The standard Linux-based network driver used in the first contribution exhibited excessive latency that could not be completely eliminated by my solutions. Therefore, in order to realize low-latency NFV service chains, we* shifted our attention to service chains that run on top of state of the art network drivers (such as Data Plane Development Kit (DPDK)). Despite using the latest advancements in the NFV research, performance bottlenecks remained, mainly due to redundancy in the internal operations applied by those chains. The next logical step towards faster and low-latency NFV service chains was to revise the way service chains were deployed. For this reason, we designed and implemented a framework that drastically consolidates chained NFs by synthesizing their internal operations. This contribution is described in detail in Chapter 7, where we propose a practical NFV system (see § 7.1) and the synthesis techniques (see § 7.2). Building upon a state of the art NFV framework (see § 7.4), we realized long and stateful NFV service chains in software at the speed of our experimental testbed (see § 7.5.2, § 7.5.3, and § 7.5.4.1). By enhancing our testbed with a hardware-based OpenFlow switch, we realized ISP-level service chains with increasing complexity at 40 Gbps (see § 7.5.4.2). The originality of this work with respect to earlier relevant efforts is described in § 7.7.

A summary of both contributions is provided in § 8.1, where I also challenge the research problem tackled by this thesis.

*The first single-core prototype of this work was jointly implemented by Georgios P. Katsikas and Marcel Enguehard. Then, Georgios P. Katsikas extended the work to meet multi-core requirements leading to line-rate performance. Finally, the hardware-assisted experiments of this contribution were jointly conducted by Georgios P. Katsikas and Maciej Kuźniar.

1.7 Thesis Outline

The rest of this thesis is structured as follows. Chapter 2 provides the background information required to understand this thesis. Chapter 3 provides an extensive literature study in the area of networked systems and NFV in particular. Given this background, in Chapter 4 we present evidence that shows performance issues of state of the art NFV service chains. We also identify the challenges that need to be addressed to solve these issues, linking these challenges with the contributions of this thesis. Having identified the problems and challenges, we then formally compose a clear research problem. Chapter 5 describes the experimental design and tools we use to conduct our experiments. Chapters 6 and 7 introduce our contributions to approaching and solving two key aspects of the research problem. In Chapter 8 we revisit the hypotheses made in Chapter 4 and relate our results from Chapters 6 and 7 with these hypotheses. We discuss the limitations of our work and sketch our future work plans in Chapter 9. Finally, Chapter 10 positions our work in today's societal, ecological, and economical planes, while Chapter 11 concludes this thesis.

Chapter 2

Background

This chapter provides the necessary background to understand the concepts and techniques used in the rest of this thesis. The focus of this discussion is mainly on networking, OSs, and hardware-related concepts, since these are the key concept behind NFV systems. To this end, a quick description of Network Interface Cards (NICs) is given in § 2.1. A study of the traditional Linux networking approach is conducted in § 2.2 to understand how applications interact with the Linux OS and the underlying hardware. Then, § 2.3 discusses how recent efforts have revised the Linux networking approach to improve its efficiency, thus achieving better performance. Finally, § 2.4, § 2.5, § 2.6 and § 2.7 discuss various auxiliary technologies and tools that affect the NFV performance.

2.1 Network Interface Cards

A NIC is an appliance that transmits and receives data to/from a physical medium, such as a coaxial cable or an optical fiber. Assuming that this communication is done via Ethernet, a block diagram of a NIC's architecture is depicted in Figure 2.1.

According to this diagram, the Medium Access Control (MAC) unit interacts with the control logic unit to send frames from the NIC's local buffers (located in the memory box of Figure 2.1) out to the network and to receive frames into the NIC's local buffers. The memory unit is used to provide temporary storage for frames (i.e., the local buffers), buffer descriptors, and other control data. The Direct Memory Access (DMA) unit is driven by the control logic unit to read and write data between the NIC's local memory and the main memory or the Central Processing Units (CPUs)' memory. This process is entirely coordinated and executed by the control logic unit of the NIC. This unit contains one or more processors that run firmware, hence these processors allow DMA to run in parallel with the system's CPU.

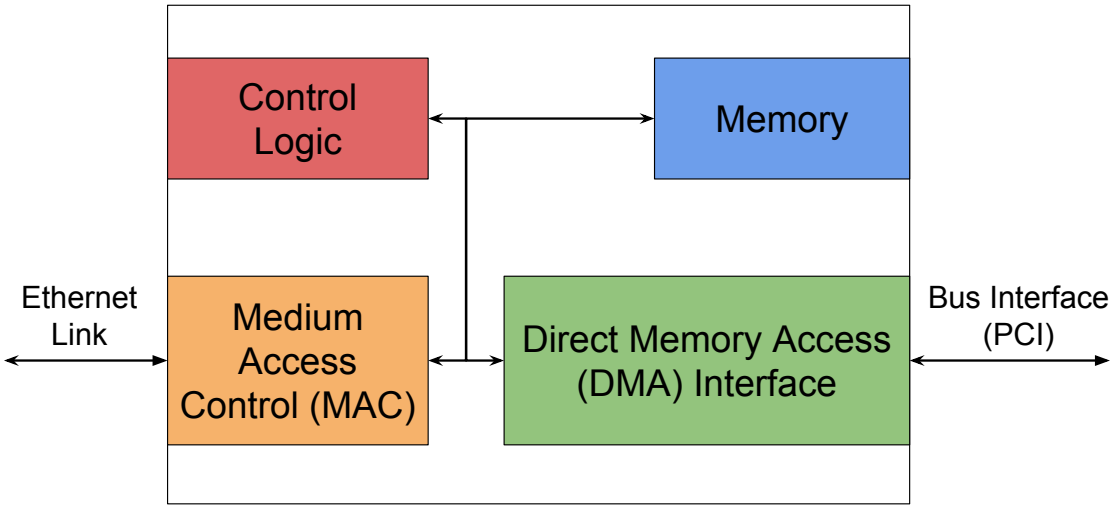


Figure 2.1: An example architecture of a programmable NIC.

2.2 Traditional Networking Paradigm

Remotely located applications communicate with each other via a network. As explained in Chapter 1, such a network comprises of interconnected devices responsible to find a path between remote applications. In a computer system, a typical way for applications to interact with this underlying network is via an OS. An OS provides APIs that allow applications to access and share the underlying hardware. In particular, network applications require to interact with three key parts of the hardware: (*i*) the CPU, (*ii*) the memory system, and (*iii*) one or more network devices, also known as NICs. This section describes these interactions focusing on the Linux OS.

2.2.1 NIC - OS Interaction

The Linux OS maintains a circular queue of buffer descriptors, also called ring buffers, and a series of buffers which are used for hosting frame headers and contents. Each buffer descriptor indicates the location of a buffer in memory and the buffer size. A buffer descriptor is the transaction unit between the OS and the NIC.

When the OS needs to indicate that a frame is ready to be sent, the steps shown in Figure 2.2 are followed. First, the OS is informed (i.e., by an application) that a frame in memory is ready to be sent. To do so, the OS creates a buffer descriptor of this frame in memory (Step 1). Then, the OS notifies the NIC that a new buffer descriptor is available in memory, ready to be fetched (Step 2). The NIC initiates a

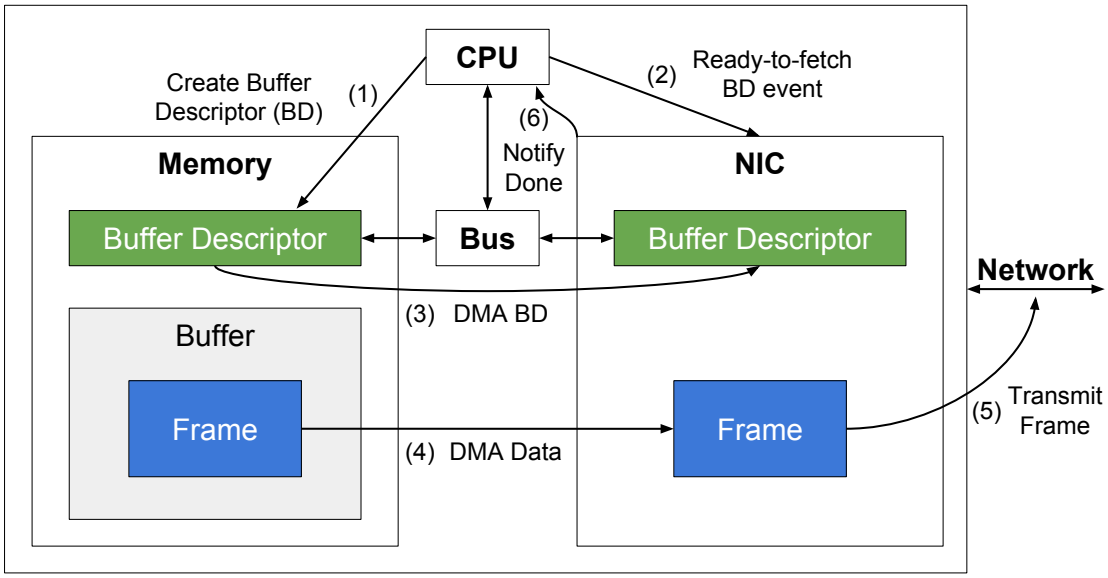


Figure 2.2: Steps for sending an Ethernet frame.

DMA read operation of the buffer descriptor (Step 3). Having processed the buffer descriptor, the NIC knows the address of the pending frame in memory, hence it initiates another DMA read operation to retrieve the frame's content (Step 4). When all the segments of the frame arrive at the NIC's local memory, the NIC transmits the frame onto the wire (Step 5). Finally, depending on how the OS has configured the NIC, an interrupt might be generated by the NIC to indicate that a frame has been transmitted (Step 6). This approach requires copying between the main memory and the NIC's local memory.

Likewise, to receive an Ethernet frame, the steps of Figure 2.3 are followed. First, we assume that the OS has already allocated a buffer descriptor that points to a free memory location and the NIC has fetched this descriptor into its' local memory via DMA. Then, upon a frame reception, the NIC stores it to its local receive buffer (Step 1). Assuming that the system has enough memory available to host this frame, the NIC initiates a write operation of the frame's data via DMA (Step 2). By examining the next free buffer descriptor (which was previously fetched as we assumed), the NIC determines the memory address where the frame will be stored. Next, once the frame is written into memory, the NIC updates the buffer descriptor with the size occupied by the new frame and possibly some checksum information. The buffer descriptor is now ready to be written into the memory via DMA (Step 3). Finally, depending on how the OS has configured the NIC, an interrupt might be generated by the NIC to indicate that the frame has been received (Step 4).

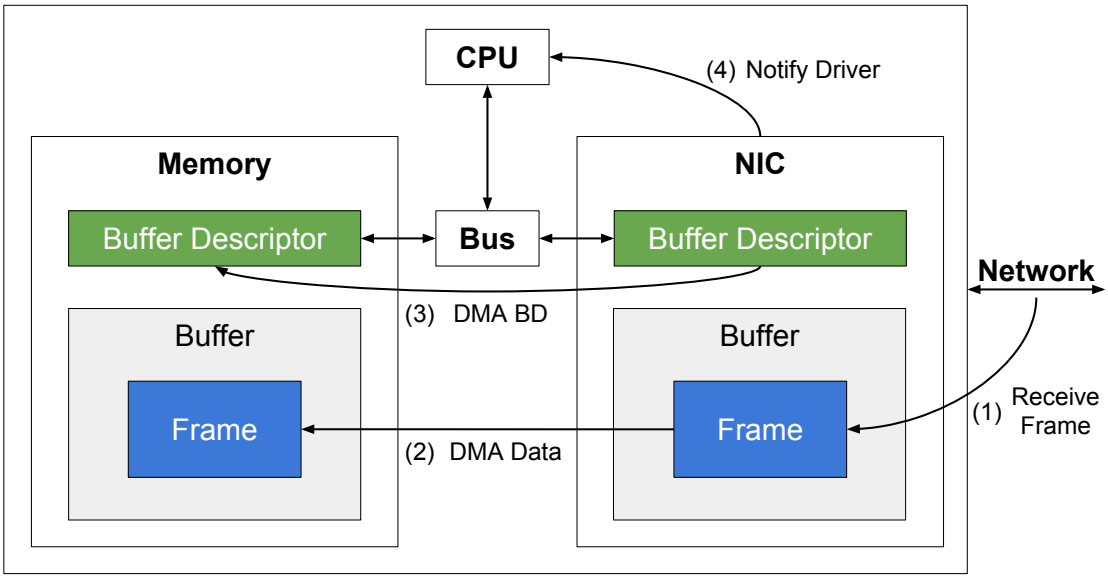


Figure 2.3: Steps for receiving an Ethernet frame. We assume that the OS has already created a buffer descriptor that points to a free memory region and the NIC has read this descriptor into its' local memory via DMA.

2.2.2 Applications' Perspective

This section details how applications that run on top of the OS utilize the OS-NIC interactions described in § 2.2.1. Figure 2.4 illustrates the way that data flows from a NIC to an application through the Linux OS and the reverse. Let us discuss the case of a frame reception in detail. When a new frame arrives at the Reception (Rx) queue of the NIC (shown in Figure 2.4), the NIC uses the DMA mechanism described in § 2.2.1 (and illustrated in Figure 2.3) to pass the frame to the ring buffers of the Linux kernel. At this point a hardware interrupt* is generated by the NIC to indicate the arrival of this frame, the frame is placed into a *Socket Buffer (skbuff)*, and then enqueued in a queue of skbuffs maintained by the kernel. An skbuff stores useful metadata for each frame, used to indicate its size, location in memory, input device and socket related to this frame, Transmission (Tx) or Rx timestamps, etc. To notify the CPU about the availability of the frame in the kernel's queue, a software interrupt is triggered by the kernel as shown in Figure 2.4. Next, the payload that is encapsulated in the frame will indicate the parts of the network stack that this frame has to traverse. For example a frame that contains an IP packet, which in turn contains a Transmission Control Protocol (TCP) segment, has to traverse the IP and TCP parts of the network stack in order for its' contents to be properly extracted.

*Interrupts are discussed in § 2.7.

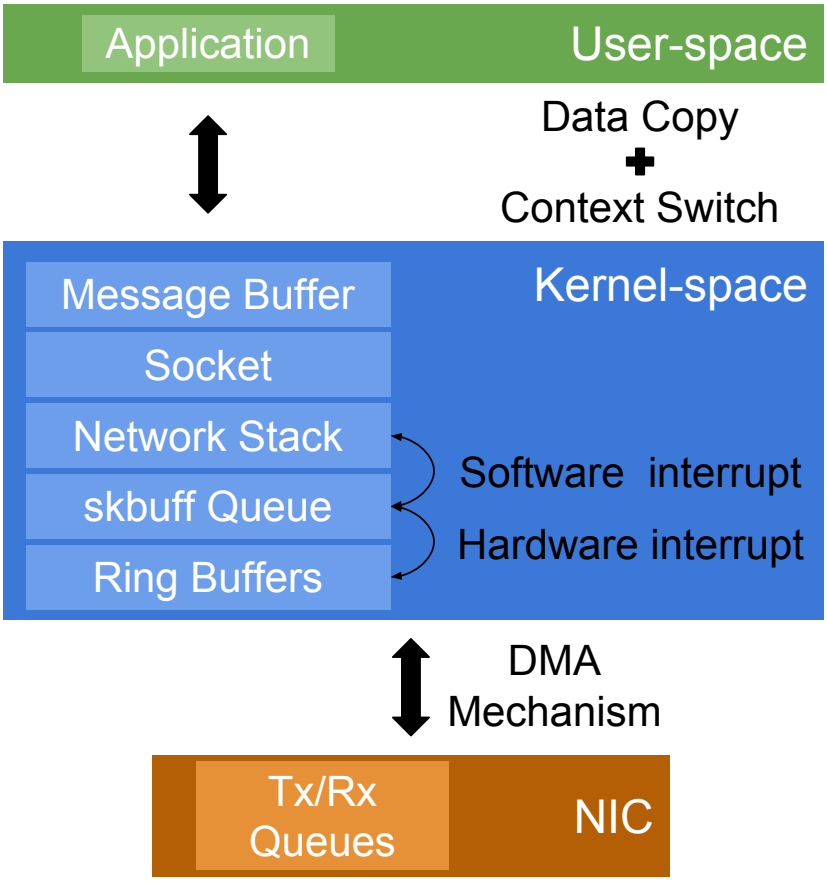


Figure 2.4: The traditional Linux network I/O paradigm.

After the decapsulation of all the headers, the data of this frame is available to the application and can be retrieved using a read system call.* This system call is issued by the application running in user-space via the receive socket function call. This function call interfaces with the kernel's socket API to copy data from the kernel's message buffer to the memory region of the application in user-space. For this to happen, the application has to yield the CPU to the OS in order for the latter to perform the I/O; this operation is known as a context switch. The next time the application gets the CPU, it can access the data according to the application's logic.

*In this example we assume that the application does not use a raw socket, hence only the payload is passed to the application by the kernel.

2.3 Revised Networking Paradigm

The interactions between applications and NICs through the Linux OS described in the previous section indicate the presence of overhead. This overhead mainly stems from the expensive system calls required to perform memory communication between user and kernel spaces, and the concomitant data copy (shown in Figure 2.4). Software interrupts are another source of overhead as explained later in § 2.7.

This section discusses recent approaches [24, 25, 26] that revise the traditional Linux network I/O paradigm in an attempt to reduce these overheads, thus increasing the performance of network applications. The details of these approaches are provided in Chapter 3 and specifically in § 3.3. The goal of this section is to pinpoint the main differences between the traditional Linux networking paradigm and these new approaches. Figure 2.5 illustrates a revised network I/O paradigm using the DPDK framework as an example.

The main principle of the DPDK network I/O paradigm is that the Linux kernel does not possess the memory allocated for packets. Instead, this I/O scheme allows applications to map regions of the NIC's local memory to their context, hence accessing the ring buffers directly. This paradigm implies that the majority of the network driver's functionality, traditionally residing in the kernel (as shown in Figure 2.4), is now implemented in user-space as shown in Figure 2.5. Chapter 4 provides an early experimental assessment of some of the benefits of this new paradigm; these benefits are summarized below:

1. costly Tx and Rx system calls are now replaced by "cheaper" memory accesses on the shared memory between the application and the device.
2. data copies from kernel to user-space and the reverse are eliminated because user-space applications have direct access to the device's packet pool.
3. context switches are almost eliminated because most of the network driver's functionality is now moved to user-space (some functionality still resides in the kernel's user-space I/O driver as shown in Figure 2.5).
4. costly software interrupts are no longer required because the user-space network driver polls the device.*

The performance benefits of the revised network I/O scheme shown in Figure 2.5 come with some implications. First, applications are now in charge of implementing parts of the network stack, as the revised paradigm bypasses the equivalent functionality offered by the Linux kernel. This increases the performance at the cost of adding complexity to the application development

*Polling is discussed in § 2.7.

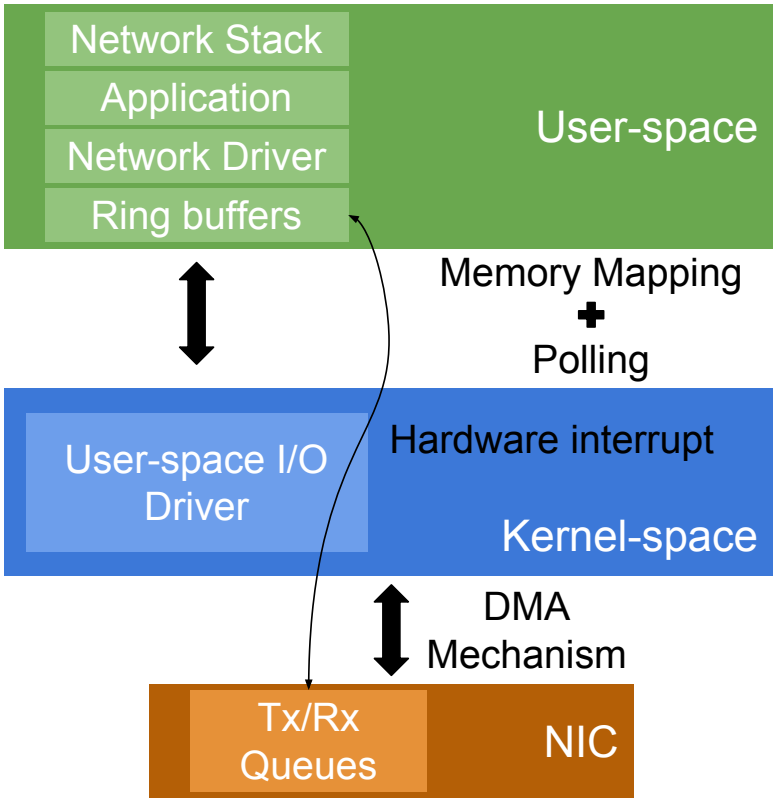


Figure 2.5: A revised Linux network I/O paradigm.

process. Moreover, the resource isolation scheme provided by the Linux kernel using the traditional approach is "violated", since parts of the in-memory packet representation model (i.e., the data structures that comprise the ring buffers) are now exposed to the user-space applications. It is up to the user-space network driver to guarantee that applications use the resources "wisely". As of the traditional Linux network I/O, this packet representation model is kept in the kernel using skbuffs; applications can access the skbuffs' contents only (indirectly) using the socket system calls' API.

2.4 Memory Access Models

Traditionally, processors have had uniform access time to memory regions over a common bus as shown in the left-most part of Figure 2.6. This access time has been independent of which processor makes the request or which memory chip contains the transferred data. The increasing number of integrated cores into a processor's chipset rendered this memory model inefficient because more and more cores had to share the common bus interconnect causing contention.

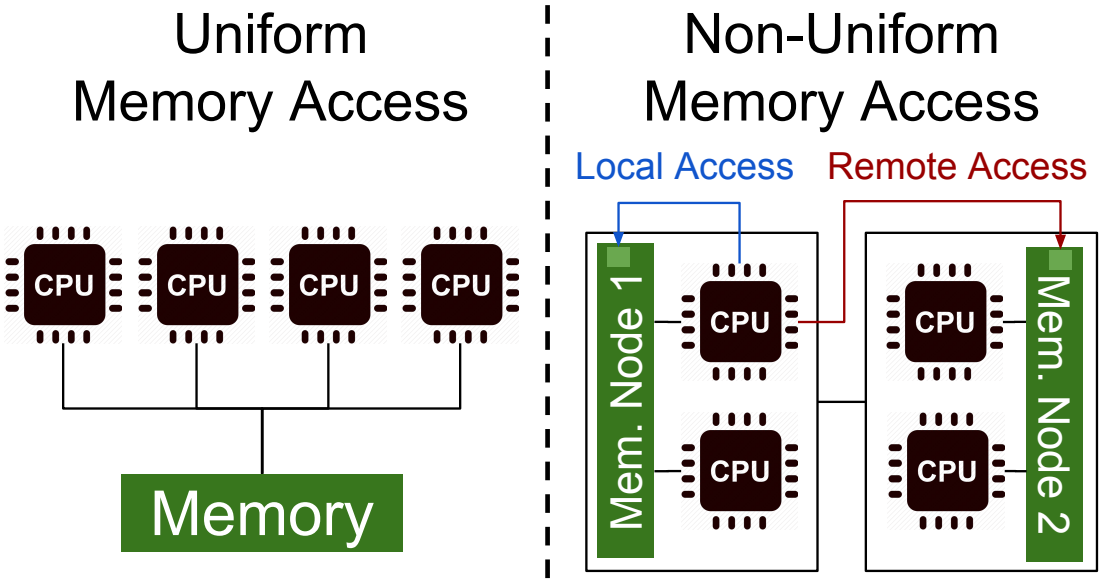


Figure 2.6: Uniform (left) vs. non-uniform (right) memory access models.

For this reason, the Non-Uniform Memory Access (NUMA) computer memory design has been fostered by multi-core architectures. In a NUMA-aware system, cores are organized into clusters that share the same Last Level Cache (LLC) and a portion of the main memory. This scheme adds an intermediate level of memory shared among the cores of each cluster, so that fewer data travel on the main bus. Each CPU core has a designated share of the memory hierarchy; thus, contention can be limited when threads are co-scheduled on cores that share parts of the memory hierarchy [27]. As shown in Figure 2.6, the memory access time depends upon the memory location relative to the processor, hence a processor can access data faster when the data is in its local memory.

2.5 Direct Data I/O

Section 2.2.1 described how NICs interact with a computer system's memory system via DMA. The memory component that undertakes this exchange is usually main memory. The left-most part of Figure 2.7 shows how main memory is involved when a frame arrives at a NIC (step 1) and has to be read by a CPU core. Specifically, step 2 requires a DMA write operation from the NIC to main memory. Then, the CPU core issues a read operation to fetch the frame to its local memory (step 3), but since the frame does not exist in the cache yet, a cache miss occurs and the frame is brought from main memory to the processor's LLC (step 4). Only then the core is able to fetch the data by issuing a transaction between LLC and its local cache.

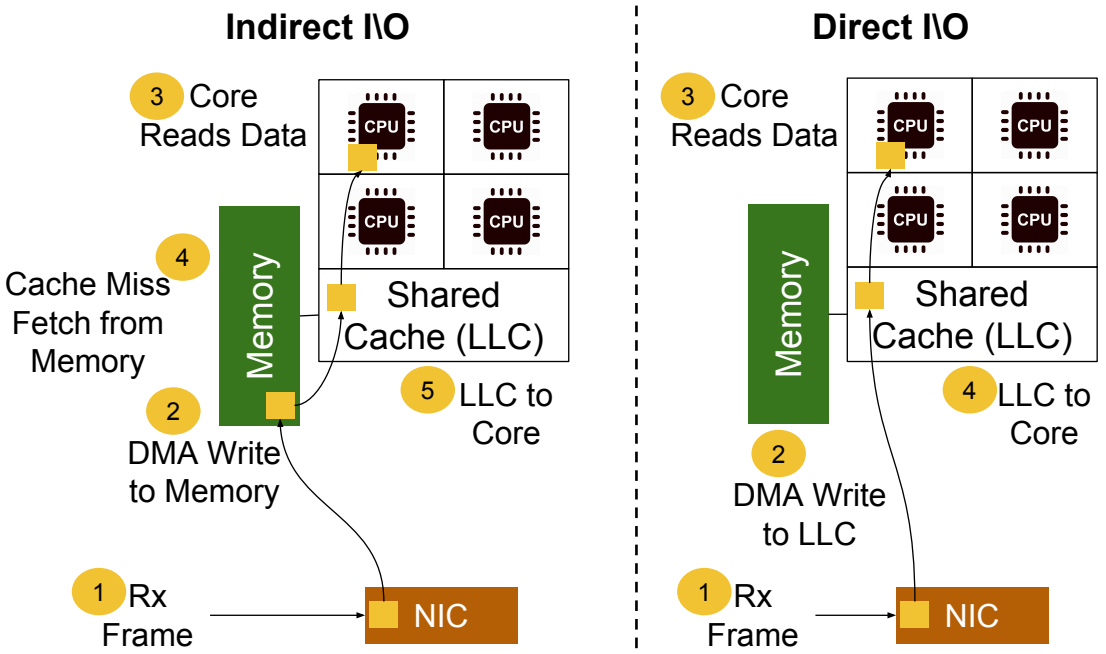


Figure 2.7: Indirect (left) vs. direct (right) network I/O models during a frame reception operation.

Apparently, this network I/O model is limited by the available memory bandwidth that interconnects main memory and the LLC as well as from the additional latency required to access the main memory. For this reason, hardware vendors developed alternative techniques to mitigate this problem. As an example, Intel released the Direct Data I/O (DDIO) technology [28]. As shown in the rightmost part of Figure 2.7, DDIO allows the Ethernet controller to use a portion of the processor's LLC as a primary source and destination of data rather than the main memory (step 2), thus achieving lower latency. The LLC portion used for DDIO might depend on the hardware architecture. CPU cores use this portion to fetch the data directly from the LLC (steps 3 and 4). If this memory limit is exceeded, new inbound data from the NIC(s) will continue to go to the LLC, but the least-recently used data will be evicted to main memory in order to make room for new data. Outbound data that resides in the LLC will be directly transferred to the NIC. This novel DMA technology enables high performance implementations of network services that have limited interactions with main memory, provided that the I/O and processing mechanisms of the network services keep most of their data in the caches.

2.6 CPU Pinning and Isolation

Running software-based packet processors on commodity hardware demands dedicated CPU resources. Hence, an NFV ecosystem should guarantee that a large set of CPU cores be dedicated to NFV tasks. To provide such a guarantee, the Linux kernel configuration needs to be modified. Specifically, setting a list of cores in the kernel boot parameter *isolcpus* ensures that the Linux scheduler will exclude these cores when selecting the next task to be executed, unless explicitly assigning a task to that core (e.g., by using the *taskset* or *numactl* commands). Consequently, pinning an NFV process (i.e., NF or software switch) to such a CPU core guarantees that this core does not serve any other processes in the system, hence the NFV process will not be preempted and the core's full power and availability can be exploited.

2.7 Interrupts versus Polling

In § 2.2.1 we explained the internal steps required to send/receive a frame to/from the network using the traditional Linux networking paradigm. The last step of each process (i.e., Tx or Rx) involves a potential generation of an interrupt. Interrupts are signals emitted by hardware or software components indicating an event that needs attention. For example, when a frame is received by a NIC, an interrupt is sent from the NIC to the OS indicating that the frame is ready to be processed. The act of initiating a hardware interrupt is commonly referred to as an *interrupt request*. Building a high-performance NFV system requires special attention to interrupt handling. This is because when processing millions of Packets Per Second (pps), handling an interrupt for each packet is very costly and unnecessary.

For this reason, several techniques have been developed to mitigate the cost of interrupts. First, interrupt request balance (commonly abbreviated as IRQ balance) is a daemon in the Linux OS that distributes the interrupts across all the available cores. This means that no single core will be loaded by serving those interrupts. If a single core were used to perform all interrupt processing, then the other cores would wait for their I/O requests to be served by that core. In modern NUMA-based hardware architectures, this daemon is smart enough to perform the interrupt request processing as close to the process as possible (i.e., same core, a core on the same die sharing the same LLC cache, or a core in the same NUMA zone). However, when the incoming packet rate is high, the throughput of the packet processing application decreases substantially if a single core undertakes both application and interrupt request processing. For this reason, we prefer to take full control of the interrupt request, thus we dedicate a core to serve interrupt

requests. For performance reasons, in a NUMA-based hardware architecture, both the interrupt request processing and application cores should be done on the same CPU socket.

To further mitigate the interrupt request cost, interrupt coalescing techniques have been incorporated in network drivers. These techniques attempt to batch a set of interrupts to alleviate the cost of generating one interrupt per packet. This implies less work for the core that processes the interrupts, but it might impose some additional latency to the application, as some of the packets might wait at the NIC's queue until the batch size is met. One way to coalesce interrupts is by specifying a time window (during which the NIC buffers interrupts), while another way is to specify the number of frames that a NIC buffers before an interrupt is generated.

Finally, a well-known technique to eliminate the interrupt cost is to constantly poll the NIC. In this case, no interrupts are generated by the NIC as there is always a (set of) core(s) listening for incoming packets. This technique achieves the best performance and is widely used by modern I/O frameworks such as DPDK and netmap. One should pay attention to the use of polling as it implies that a CPU core must be dedicated to listening for incoming packets from the NIC. Therefore, some implications of polling are: (*i*) the core is no longer available for performing other tasks and (*ii*) if no or very few packets arrive at the NIC where the core performs polling, this core is heavily underutilized. To solve these problems, one could employ load balancing techniques to redistribute the load across several cores by dynamically assigning flows to different hardware queues at the NIC.

Chapter 3

Related Work

This chapter reviews the literature in several key research areas, in particular in NFV. The goal is to draw inspiration from these efforts and identify those gaps that motivated this thesis.

Sections 3.1, 3.2, and 3.3 summarize the historical evolution of the technology used by telecommunications stakeholders to provide NFs during the last two decades. Sections 3.4 and 3.5 discuss a summary of works that leverage these NFs to provide network management solutions and consolidated services respectively. This is followed by a discussion of recent system performance accelerations related to networked systems (see § 3.6), and tools that can be used to analyze the performance of these systems, while executing NFV service chains (see § 3.7).

This chapter provides a high-level overview of the literature. A direct comparison of the contributions reported in this thesis with the literature are given in § 6.6 and § 7.7.

3.1 From Specialized Hardware to Software

In the early 1990's, specialized hardware was the only means to provide line-rate packet processing and forwarding for high speed links. However, since that time research in NF "softwarization" began.

The UNIX System V STREAMS [29] was a modular packet processing system that implemented implicit queuing and packet scheduling mechanisms spread throughout a STREAMS configuration. Decasper et al. [30] implemented modular per flow packet processing elements executed after a classifier. In router plugins, these packet classifiers are installed in fixed points in the forwarding path. Moreover, in 1999 David Cinege published the Linux Router Project [31] which provides a tailored version of the Linux kernel specialized for routing. Originally, this project was designed as a "router on a floppy disk" and evolved into a

streamlined network OS. Research attempts also focused on routing operations, such as IP lookups. In 1997, Degermark et al. designed and implemented a data structure specialized for quick IP lookups [32]. The authors achieved full routing lookup at gigabit speeds on commodity hardware, by fitting the forwarding table into a processor's cache.

Driven by the need to fully control queuing and packet scheduling operations, Kohler et al. developed Click [33], a platform to build NFs using simple, modular packet computation elements. Combining 16 Click elements, such as IP classification and decrement Time To Live (TTL) field, one can build a standards-compliant software router. To extend this router into a more sophisticated NF (e.g., a middlebox that implements drop policies or differentiated services), we simply add a few elements at the correct place in a Click configuration. In 2000, when Click was published, achieving a loss-free forwarding rate of 333,000 64-byte pps was an important performance milestone. Click's performance was comparable to the performance of a modified software-based Linux router using Click's network device extensions [33]. Also, as compared to router plugins, Click allows more dynamic traffic classification, embedded into Click's pipeline.

Software-based packet processing architectures achieved better performances with the advent of multi-processor computing architectures around 2007. In conjunction with OpenFlow's [14] introduction in 2008, software has become a first class component creating the trend that is today called SDN. Because of these changes, the logical successor of NF "softwarization" is NFV.

3.2 Software-based Packet Processing Frameworks

The Linux router project, router plugins, and Click were among the most mature software-based packet processing engines of the previous century. Recently, new contributions were introduced in this area. Slick [34] introduced a programming language based on Python that allows a programmer to compose distributed, network-wide service chains driven by a controller. Additionally, Slick is able to address placement requirements of these chains.

The DPDK Packet Framework [35] is a production-level software development kit that provides reusable and extensible tools to build complex packet processing pipelines exploiting the fast I/O provided by DPDK. A major difference between DPDK's Packet Framework and earlier works is that DPDK inherently supports hardware acceleration.

P4 [36] is a combination of a high-level language for programming protocol-independent packet processors, coupled with a compiler that maps programs to a

variety of target hardware switches. P4 is considered as the logical continuation of the current OpenFlow [14] match-action protocol, with a simpler and more flexible specification of the same functionality.

Contemporaneously with P4, OpenState [37] allowed SDN switches to perform stateful packet processing without involving the SDN controller. To do so, Bianchi et al. modeled control and processing tasks that can be executed by SDN switches using extended finite state machines. This model aims to increase the SDN switches' capabilities, while retaining (*i*) centralized control of their execution, (*ii*) platform independency, and (*iii*) high performance and scalability. This idea was extended by the BEhavioural BAsed forwarding project [38], leading to even more powerful SDN switches that can e.g., generate packets [39].

3.3 Network I/O and Processing

As a consequence of the above evolution, networked systems' research community focused on improving the I/O performance of software-based packet processors.

First, the Linux kernel has evolved over the past fifteen years and today provides sufficient tools to speed up NFV applications running on top of unmodified network drivers. In 2005, Olsson introduced pktgen [40]; a modular component of the Linux kernel that permits quick and fast Tx and Rx tests by closely interacting with the NICs via the kernel's network driver.

Linux developers realized that system calls add substantial overhead to network I/O intensive applications. To amortize this overhead, vectorized I/O [41], introduced in version 2.5 of the Linux kernel, permits reading/writing frames from/to multiple buffers using a single transaction with the kernel. User-space network I/O was further accelerated by integrating the asynchronous, zero-copy network I/O solution proposed by Drepper [42].

Traversing the entire Linux network stack was still costly, despite the above advancements. For this reason, researchers tailored the host's and/or guests' network stacks and drivers to achieve line-rate forwarding and to maximize throughput. In this direction, netmap [25], DPDK [24], and PFQ [26] are fast network I/O frameworks that boost the performance of middleboxes by (*i*) granting a user-space application access to the NIC buffers, enabling zero copy data transfers from kernel to user-space, (*ii*) pre-allocating memory resources, and (*iii*) batching packet processing to amortize system call overheads over multiple packets. Software switch implementations such as Open vSwitch (OVS) [43, 44], VALE [45], CuckooSwitch [46], and mSwitch [47] take advantage of these frameworks to realize fast switching across (virtualized) NF instances.

With all these accelerations in place, Kim et al. in [48] introduced an extension of the Click modular router that reached 28 Gbps of throughput using aggressive

computation and I/O batching techniques. ClickOS [49] and NetVM [50] platforms deployed dedicated VMs, running on top of Xen [51] or KVM [52] hypervisors respectively, to realize middlebox processing tasks. Both frameworks achieved packet switching between VMs at 10 Gbps line-rate for any packet size. More recently, OpenNetVM [53] showed that VM-based NFV deployments do *not* scale with increasing number of chained instances, hence opted for NFs running natively in lightweight Docker [54] containers. By interconnecting these containers with DPDK Tx and Rx ring buffers, OpenNetVM realized NFV service chains with higher throughput.

An alternative direction towards achieving better performance was to scale software middleboxes for modern, multi-core and sometimes heterogeneous hardware architectures. RouteBricks [55] took advantage of parallelism in the OS and hardware levels to introduce the first parallel Click router prototype, able to achieve 35 Gbps throughput. Most importantly, RouteBricks showed that performance was no longer the Achilles' heel of software routers. Moreover, PacketShader [56] exploits the massively-parallel processing power of a Graphics Processing Unit (GPU) to overcome the CPU limitations of software-based routers and achieve a forwarding rate of almost 40 Gbps for small frames. In the same context, NBA [57] introduced mechanisms to balance the load between CPUs and a GPU, in an attempt to further increase the router's performance. The results showed that an IP router could reach the line-rate of an 80 Gbps server. Finally, Barbette et al. in [58] proposed FastClick; a Click variant that combines tedious low-level configurations and techniques to turn Click into a fast user-space packet processor. The authors of FastClick compared their approach with a large variety of other Click-based works (e.g., PacketShader), showing that FastClick with DPDK or netmap-based I/O outperforms other state of the art approaches.

3.4 Middleboxes and Middlebox-aware Policy Enforcement

In [12], Qazi et al. introduced an SDN-based policy enforcement layer for steering traffic in a middlebox-aware way. Using flow correlation techniques to derive knowledge about middlebox modifications, the goal was to deal with dynamic packet modifications and provide a unified switch and middlebox resource management scheme. A key element of this approach is that it does not place any implementation constraints on middleboxes.

Following the same principles, Tracebox [59] inspected packets crossing the network to locate middleboxes and identify middlebox interference. FlowTags [60] tackled the policy enforcement problem by infusing a tagging module into

middleboxes. This approach requires a standardized API to expose the processing logic of middleboxes and to assign appropriate tags.

Header Space Analysis (HSA) [61] provided a network model to compose transfer functions of network elements, including middleboxes. Finally, OpenNF [62] proposed a programming API to safely and quickly migrate the state of operational middleboxes in virtualized environments.

3.5 Middlebox Consolidation Architectures

CoMb [63] presented a new middlebox architecture that explored opportunities for application-level consolidation by decoupling software from hardware and providing a logically centralized point for managing groups of middleboxes. Bremler-Barr et al. introduced the DPIaaS platform to virtualize the DPI function [64]. The goal of DPIaaS is to share this heavy and costly service among multiple instances to reduce the load within an NFV environment.

Deeper in the network stack, Open Middleboxes (xOMB) [65] proposed 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. Besides its packet processing abilities analyzed above, Slick [34] also introduced a consolidated control plane for middleboxes by sharing common elements among multiple middleboxes.

More recently, an advanced NFV framework called E2 [66] has been proposed by Palkar et al. Inspired by data analytics frameworks, such as the Apache Hadoop [67], the authors proposed an architecture that allows the definition and deployment of NFV jobs in the cloud. E2 provides (*i*) generic techniques to implement NFs, (*ii*) service composition using multiple NFs, (*iii*) resource allocation for each service, (*iv*) instantiation of the required number of NFs for a given service (elastic scaling), (*v*) load balancing among a service's instances, and (*vi*) placement of these instances across the servers of the NFV infrastructure.

The latest NFV consolidation work is OpenBox [68]. The authors applied the SDN control and data plane separation paradigm to the OpenBox framework, allowing network-wide deployment and management of NFs. OpenBox applications input different NF specifications to the OpenBox controller via a north-bound API. The controller communicates the NF specifications to the OpenBox Instances (OBIs) that constitute the actual data plane, 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. Another interesting observation is the need to apply traffic classification at the chain-level (i.e., 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.

3.6 Scheduling Techniques in SDN and NFV

Limited but essential contributions have been recently proposed in a different layer of modern networked systems.

Inspired by the programmable flow tables of the SDN paradigm, Sivaraman et al. [69] enabled similar programmability in the packet scheduler of SDN switches. Their study focused on controlling the order and departure time of packets; these parameters cover the needs of a broad set of packet scheduling schemes.

Mittal et al. [70] envisioned a universal packet scheduler that could potentially match the results of any scheduling algorithm. According to their study, there is no universal packet scheduler, but the best approximation of such a scheduler is offered by the Least Slack Time First algorithm. They support their findings by showing how the proposed scheduler can minimize average flow completion times and tail latencies, while maintaining per flow fairness.

Along the same lines, a technical report from Rizzo et al. [71] proposes an architecture to run software-based packet schedulers in modern high-speed and highly-concurrent virtualized environments. This work isolates scheduling decisions from the data plane allowing (*i*) the scheduler to make very fast (i.e., over 20 millions) scheduling decisions, while (*ii*) packet transmissions can run in parallel exploiting multi-core capacities.

3.7 Performance Monitoring Tools

The NFV vision is to turn the hardware-based network processing into software running in commodity hardware and OSs. Therefore, it is crucial to study the most relevant tools and research works with respect to system profiling to investigate how one could monitor the performance of NFV software stacks and identify potential problems.

System benchmarks

An NFV infrastructure can be seen as a network OS, since it is essentially comprised of interconnected commodity servers that run modern multi-core OSs such as Linux. The main source of data about such a system's performance can be collected by tools that reveal the hardware's performance. lmbench [72] is a benchmark suite designed to measure bandwidth and latency of critical building blocks of OSs. For example, lmbench provides tests for detailed memory (i.e., cache and main memory), networking (e.g., connection establishment, pipe, User Datagram Protocol (UDP), TCP), file system, process, signal handling, and context switching measurements.

Similarly, Intel's memory latency checker [73] can quantify the latency when transferring data of variable sizes across different hardware components (i.e., registers, caches, and main memory) of an Intel chipset, hence benchmarking the hardware's memory performance.

Code Profilers
Modern code profilers, such as OProfile [74] and Perf [75], access low-level performance counters at run-time, providing statistics about applications or the entire OS. Such tools can detect "expensive" functions, allowing developers to focus their efforts on optimizing critical parts of a software stack.

Data Profilers
CProf [76], callgrind's KCachegrind tool [77] (based on valgrind), and Intel's Performance Monitoring Unit (PMU) [78] are popular tools that can track applications' cache utilization. Likewise, likwid [79] provides modular tools that allow programmers to measure the performance of either an entire application or specific parts of an application. Finally, DProf [80] helps programmers understand cache miss costs by associating these misses with the data types instead of the code.

Chapter 4

Research Problem and Challenges

The advances in NFV are numerous and substantial, hence NFV is moving towards commercialization [20, 13, 21]. However, there still remain open problems that appear to be extremely challenging and require scientific answers. § 4.1 describes **a major problem** in the area of modern networked systems, with a focus on NFV. § 4.2 describes the challenges associated with the problem. § 4.3 formulates a research question, which is boiled down to a set of hypotheses in § 4.4, and a research methodology used to tackle the research problem in § 4.5.

4.1 The Problem

The main motivation of early NFV efforts was to eliminate the overhead of traversing the network stack of commodity OSs. However, even when using more lightweight network stacks, such as Click [33], performance problems remain since the communication between the NFV applications and the NICs still goes through unmodified network drivers. For this reason, researchers developed high-performance network drivers that bypass the OS's kernel, allowing a NIC to directly interact with user-space NFV applications. Details about the state of the art were presented in Chapter 3. Our goal is:

> **Goal:** Reveal the problems of NFV service chains by measuring a state of the art network stack using:
> (*i*) unmodified Linux network drivers (see § 4.1.1), and
> (*ii*) a highly-optimized state of the art network driver (see § 4.1.2).

In both cases, we use two identical machines, each with a dual socket 16-core Intel® Xeon® CPU E5-2667 v3 clocked at 3.20 GHz [81]. The machines are directly interconnected using two dual-port 10 Gigabit Ethernet (GbE) Intel 82599 ES NICs. One machine generates and sinks traffic (using different cores), measuring

the end-to-end latency and throughput. The second machine acts as the NFV host, where we deploy service chains on top of a state of the art NFV framework called Click [33]. More details about our testbed are provided in Chapter 5.

4.1.1 Commodity Network Drivers

We begin by measuring the end-to-end latency of chained user-space Click NFs, using unmodified Linux network drivers. As shown in Figure 4.1, we created chains of 1-8 routers that run (*i*) as individual processes in Linux containers interconnected with either the Kernel-based Open vSwitch (OVSK) [43] software switch, or Back-to-Back (B2B) and (*ii*) in a single process (denoted as OneProc in the legend).

We used one dedicated CPU core to execute the NFs, and in the case of the OVSK-interconnected chains, another CPU core for the switch. We injected 5

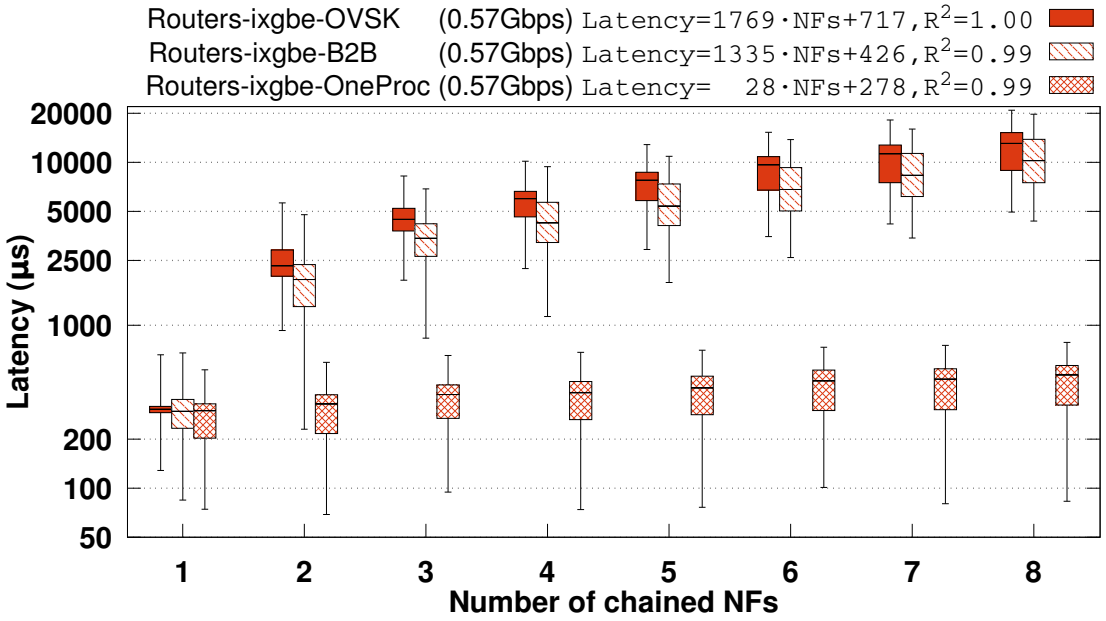


Figure 4.1: End-to-end latency (μs), plotted on a logarithmic scale, versus the chain's length for user-space Click routers based on the native Linux network driver. The chains run (*i*) as individual processes in containers interconnected with either OVSK, or B2B and (*ii*) in a single process. The chains run in a single core and in the case of the OVSK chains, OVSK is scheduled in a different core in the same socket. The input rate is 0.82 Mpps with 64-byte frames (resulting in a bitrate of 0.57 Gbps). The linear fit to the median latencies, stated in the legend, begins from the chain with 2 NFs.

million frames at an input rate of 0.82 Millions of Packets Per Second (Mpps). The frame size is 64 bytes without counting the trailing frame Cyclic Redundant Check (CRC), that we assume will be computed by the NIC itself. This packet rate corresponds to a bitrate of 0.57 Gbps and is the maximum rate that this software router can sustain without dropping packets. We chose a small frame size to impose more work on the CPU. The boxplots of Figure 4.1 show the end-to-end latency of each chain versus the chain's length.

To quantify the latency of each chain, we fitted the median latencies of the boxplots, leading to the equations shown in the legend of Figure 4.1. The fitting starts from the chains with 2 NFs. Based on these equations, each additional router in the OVSK chain adds 1769 μs of median latency, while the B2B-interconnected routers exhibit 1335 μs of median latency for each additional router. This difference (i.e., 1769-1335=434 μs) quantifies the overhead of using OVSK as a means to interconnect the chains.

The routers that are chained together in the same process (denoted as OneProc in the legend) exhibit a considerably lower latency than the multi-process chains. The main reason is the reduction in the number of I/O operations and the number of context switches required to realize a single process chain. Specifically, 28 μs of additional latency is imposed by these chains, for each additional NF; this latency is roughly 1.5 orders of magnitude lower than the multi-process chains. It is also worth noting that the 99$^{\text{th}}$ percentile of the latency increases to almost 1 ms for a single-process chain with 8 NFs.

Observation 1: Service chains realized as multiple processes face severe performance degradation with increasing chain length when compared to single-process service chains, when both use a commodity network driver.

4.1.2 State of the Art Network Drivers

In § 4.1.1, we observed excessive overhead when realizing user-space service chains, using unmodified Linux network drivers. For brevity, we refer to these chains as ixgbe-based chains (since the native Linux network driver for our Intel NICs is called ixgbe). The goal of this subsection is to assess whether the latest NFV techniques can realize these service chains without performance degradation.

Recent state of the art works improved the performance of software-based NFs by utilizing fast network drivers such as netmap [25] or DPDK [24]. Despite this fact, some of these works, such as ClickOS [49] and NetVM [50], showed that NFV service chains still face performance problems when chaining more than 3 NFs together. In particular, Figures 10 and 12 of the ClickOS and NetVM

papers respectively, show that throughput decreases 30-90% with increasing service chain's length.

These frameworks rely on hypervisors (i.e., Xen [51] for ClickOS and KVM [52] for NetVM) to schedule and interconnect VMs. Despite using fast I/O technologies, the overhead of deploying dedicated VMs to run NFs is still high, even though ClickOS uses a lightweight micro-kernel VM instance called miniOS.

One might imagine that service chaining using more efficient NFV frameworks might not face the same performance degradation. To examine this, one of the fastest NFV frameworks to date, called FastClick [58], was deployed with the same chains shown in § 4.1.1. FastClick is an extension of Click that realizes user-space NFV pipelines using netmap or DPDK-based I/O mechanisms, coupled with I/O and computation batching and multi-core capacities. In § 4.1.1, we showed that the single-process chains comprise the fastest way of service chaining; therefore, the same single-process chains are realized using FastClick. As can be seen in Figure 4.2, this service chaining is much faster than ClickOS or NetVM. This occurs because the chains run *natively*, thus eliminating the hypervisor costs. In these experiments, each router has two 10 Gbps NICs and one dedicated core per NIC is used to perform I/O at line-rate. Moreover, multiple CPU cores are utilized, by dispatching frames from the two I/O cores to another two cores that perform packet processing. All cores belong to the same socket enabling data exchange with low latency.

4.1.2.1 Low Input Packet Rate

To compare the state of the art DPDK-based chains with the ixgbe-based chains presented in § 4.1.1, the same 64-byte long frames are injected at the same rate (i.e., 0.57 Gbps) in order to measure the end-to-end latency. The top entry in the legend of Figure 4.2 shows the latency of these chains with an increasing number of routers.

Using this setup, the median latency of a single DPDK-based FastClick router is 8 μs. This latency is almost 40x (i.e., 8 vs. ~300 μs) lower than the latency of the same router using the native Linux network driver. Secondly, as denoted by the top entry in the legend of Figure 4.2, an additional router in the DPDK-based chain imposes only 1.4 μs of additional latency. Compared to the same chains realized with the ixgbe network driver, this latency is 20x lower. Indeed, the coefficient (28 μs) of the third equation (from top to bottom) in the legend of Figure 4.1 confirms this observation.

The reasons of these differences are (*i*) the effectiveness of FastClick's I/O mechanisms and (*ii*) the fact that the DPDK-based FastClick chains use dedicated cores for I/O and processing, whereas the ixgbe-based chains run in a single core. Two observations can be made from the results of these experiments so far:

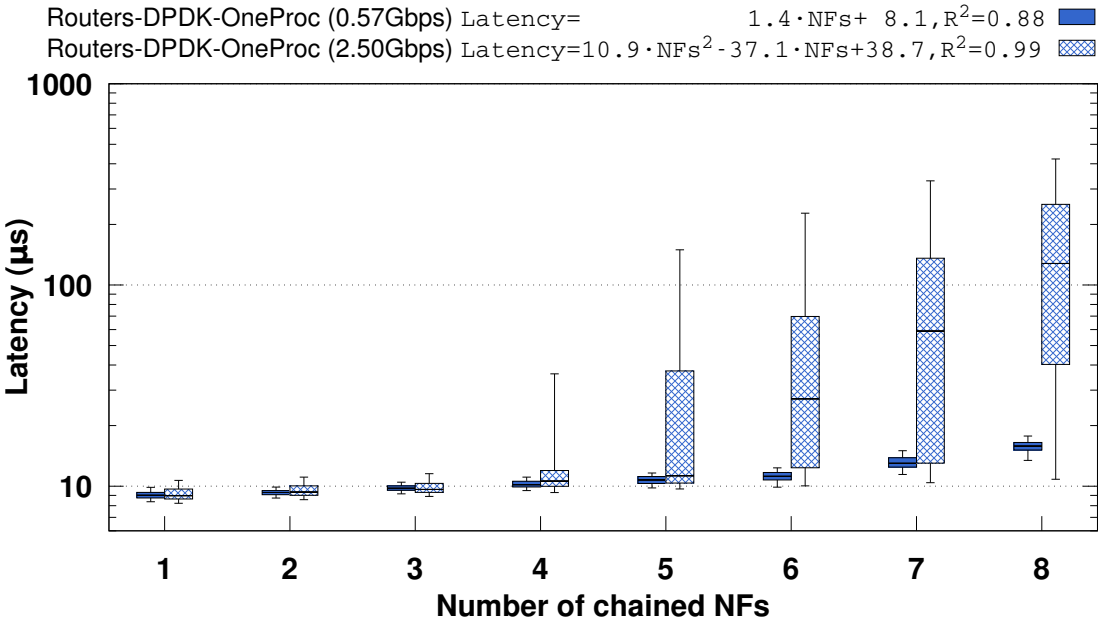


Figure 4.2: End-to-end latency (μs), plotted on a logarithmic scale, versus the chain's length for user-space FastClick routers based on the DPDK network driver. The chains run natively in a single process using 8 CPU cores (4 cores for I/O and 4 cores for processing). The input traffic consisted of 64-byte frames at 0.57 and 2.5 Gbps.

Observation 2: State of the art NFV frameworks, assisted by state of the art kernel-bypassing network drivers, can realize some long service chains with negligible performance degradation at low input rates (i.e., below 1 Gbps), even for small frame sizes.

Observation 3: Commodity network drivers inflict at least 10 times more latency on a state of the art NFV framework, compared to state of the art kernel-bypassing network drivers.

4.1.2.2 High Input Packet Rates

Several studies have shown that 10 GbE is widely used in today's datacenter networks, while a 100 GbE era will be established by 2020 [82, 83]. To this end, industry leaders, such as Cisco [84] are launching replacement infrastructure for 10 Gbps networks with 4x more capacity. QLogic also demonstrated end-to-end

interoperability between 25 and 100 GbE [85]. In large scale environments, service chains need to keep up with increasing input rates. To examine the scalability of state of the art NFV frameworks, two additional experiments are performed.

First, the input bitrate injected to the DPDK-based FastClick chains shown in Figure 4.2 is increased by 5x (from 0.57 to 2.5 Gbps). The latency of the DPDK-based router chains, under this increased rate, is quantified by the second legend (from top to bottom) in Figure 4.2. Despite the 5-fold increase in the input rate, the chains of 1-3 routers still perform well, with only 8-12 μs of median latency and nearly zero latency variance. However, after this point there is a substantial median latency increase that results in a chain of 8 routers requiring almost 500 μs to deliver packets through the chain. More interestingly, the latency variance follows a similar trend with latency, with the 1st and 99th percentiles of the latency up to 1.5 orders of magnitude apart.

This experiment is a first sign that state of the art NFV frameworks cannot realize long service chains without performance degradation under high input rates. Even though these chains consist of rather simple NFs, such as routers, and the input rate is less than the maximum rate we could inject.

The second experiment aims to test the limits of FastClick. Using the same FastClick router chains and the same frame size, we perform a throughput test at the maximum capacity of our testbed at 40 Gbps.* Figure 4.3 shows the throughput of the router chains. In this experiment all 4 NICs of the service chains are utilized and all the CPU cores of our socket (i.e., 4 cores for I/O and 4 cores for processing) are exploited. Note that the maximum achievable throughput for the 64-byte frame size is 31.5 Gbps, since this is the limit of our NICs [58]. In this experiment FastClick can operate at the maximum throughput only for a chain of 1 or 2 routers. The equation fitted to the median throughput for each chain shows that there is a quadratic throughput degradation that results in a chain of 8 routers achieving only 10 Gbps of throughput. This degradation is more than 2 times lower than the line-rate throughput of our testbed and one would expect an even greater degradation if more complex NFs are chained together.

From the last two experiments, which stressed a state of the art NFV framework at high input rates, we conclude:

Observation 4: State of the art NFV frameworks, using state of the art kernel-bypassing network drivers and multi-core capacities, exhibit serious performance degradation, when realizing service chains at tens of Gbps.

*A precise latency experiment at 40 Gbps might require harware-based timestamping. Software-based timestamping is used in this thesis instead, hence such an experiment was not possible.

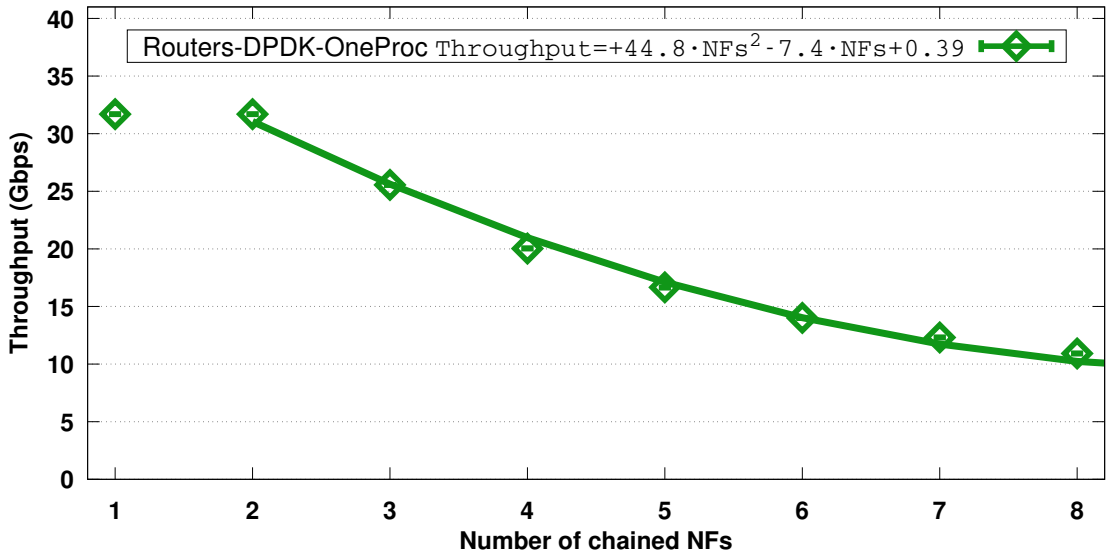


Figure 4.3: Throughput (Gbps) versus the chain's length for user-space FastClick routers based on the DPDK network driver. The chains run natively in a single process using 8 CPU cores (4 cores for I/O and 4 cores for processing). The input traffic consisted of 64-byte frames at 40 Gbps (10 Gbps per NIC with 4 NICs in total). The fit to the median throughput, stated in the legend, begins from the chain with 2 NFs.

4.1.3 Problem Definition

The four observations made above lead to the definition of a problem in the area of telecommunications systems:

> **Problem Definition:** State of the art NFV frameworks, exhibit *serious performance degradation*, when realizing service chains either using (*i*) commodity or (*ii*) state of the art network drivers.

Obviously, this problem is related to the underlying technology (i.e., the network driver) that is used to realize the NFV service chains. Chapter 1 highlighted the importance of NFV service chains in modern networked systems, hence this problem is of *major importance* for relevant stakeholders.

4.2 The Challenges

This section discusses the key challenges that emerge from the problem defined above. In response, the research approaches are highlighted, in an attempt to address these challenges, hence solve the problem.

4.2.1 Challenge 1

As described in 4.1.1, we observed high latency when chaining NFs that utilize the native Linux network driver. This was the primary reason that led researchers to develop fast, kernel-bypassing network drivers such as DPDK. Indeed, in § 4.1.2, we saw how such a fast driver can dramatically decrease the latency of service chains when the input data rate is low. However, the replacement of a commodity network driver with a custom one has several implications: (*i*) the interest of the researchers' is diverted from the performance problem of the commodity network driver and (*ii*) popular services to date, offered by e.g., Amazon [86], still rely on commodity OSs and network drivers.

Challenge 1: An in-depth analysis of the system's state, when commodity NFV applications are executing, would reveal the exact root causes of the observed performance. NFV stakeholders could benefit from tools that thoroughly analyze "hot" parts of NFV software stacks and draw attention to those functions that heavily utilize system resources, hence offer the greatest potential for performance improvements. Such tools could solve (part of) the performance problem, having a direct impact on popular services used by a large number of end users.

Approach to Challenge 1: I address this challenge in Chapter 6, by introducing an NFV profiler. To the best of my knowledge, this is the first NFV profiler in the literature. The proposed profiler uncovers performance issues in both standalone and chained NFs by accessing low-level hardware and software performance counters.

4.2.2 Challenge 2

Chaining NFs in a multi-process context (i.e., when each NF is a different process) is costly as shown in § 4.1.1. I/O is a critical factor in the observed latency, as also reported by earlier works [26, 24, 48, 25]. However, there is very little related work proposing alternative accelerations on different levels of the OS, except for I/O.

In response to this need, as discussed in § 3.6, Sivaraman et al. [69] revised the abstractions of modern switches, by proposing programmable packet scheduling techniques, while Mittal et al. [70] introduced packet scheduling algorithms that can approximately meet the requirements of a universal packet scheduler. These approaches can affect the order and timing of packet departures from a queue in a switch or NF.

Challenge 2: We believe that scheduling is a very powerful mechanism to improve the performance of processes, such as those occurring in a service chain. The works above inspired us to study scheduling schemes for NFV service chains. However, in contrast to [69, 70], a chain of NFs might require a global scheduler to make chain-level decisions, rather than an internal scheduler that executes local switch policies. This idea is inline with Amazon's attempts to integrate custom schedulers in their cloud services [87]. Therefore, the challenge is to design and implement a task scheduler that fits into the NFV ecosystem, taking into account the characteristics of the software-based packet processors, especially when an NFV provider wishes to schedule multiple NFs together.

Approach to Challenge 2: I address this challenge in Chapter 6, by proposing an NFV service chain scheduler. To the best of my knowledge, this is the first NFV service chain scheduler in the literature. The proposed scheduler employs techniques to adjust the frequency of I/O operations, in tandem with adjusting the priority and time quanta allotted to each NF, to maximize the effective run-time of the service chain, by reducing I/O and scheduling overheads.

4.2.3 Challenge 3

As shown in § 4.1.2, the quadratic performance degradation of state of the art NFV frameworks at high packet rates suggests the existence of redundancy when chaining the code of several NFs together. As described in § 3.5, several NFV consolidation frameworks have been proposed since 2012 [63, 65, 64, 34, 66, 68], mainly targeting the reduction of this redundancy. However, none of these works managed to consolidate a service chain or entirely eliminate the redundancy of the operations within the chain.

For example, Slick [34] falls short in cases where middlebox chains must be consolidated in the same machine and specifically in the same process in order to avoid context switching. Slick is unable to exploit the locality and speed of transferring packets *within* a single CPU core context; thus, Slick cannot deliver traffic through the chain with low latency.

Challenge 3: We believe that it is crucial to drastically consolidate a chain of NFs, to achieve the low latency necessary for fifth generation [88, 89, 90] communications. Specifically, meeting the target latency of 1 ms set by various fifth generation initiatives is extremely challenging, mainly because of the laws of physics; therefore any solution to this problem must minimize the number of hops of the chain (i.e., ideally achieving a single hop) and must eliminate all redundancies across the internal operations of the chain.

Potential challenges while implementing such an NFV consolidation framework would be (*i*) how this framework can effectively maintain the states of the participating NFs in the chain, (*ii*) whether it is feasible to consolidate these states, and (*iii*) how the traffic classification process can be realized at scale when multiple and complex NFs are chained.

Approach to Challenge 3: We address this challenge in Chapter 7, by proposing a highly-optimized NFV service chain synthesis framework. Our approach minimizes the number of elements that apply read, write, and discard packet/flow operations, allowing long, stateful, and complex service chains to be realized at 40 Gbps.

4.3 Research Question

Having introduced the research area (see Chapter 1), acquired enough background (see Chapter 2), studied the state of the art contributions (see Chapter 3), and highlighted the current problems and challenges (see § 4.1 and § 4.2), it is high time to formulate the research problem that this thesis project tackles. That said, we are ready to state a concrete research question as follows:

> Is it possible to maintain the high performance of a service chain (or chain of NFs) despite its length and complexity?

4.4 Hypotheses

Currently, the problem identified in § 4.1.3 has clear evidence that the null hypothesis H_0 is true for a set of implementations of some service chains.

> H_0: Service chains inherently exhibit performance degradation that depends upon the length and complexity of the chain.

In this thesis we use the Socratic method [91, 92], also known as "maieutic method" ("μαιευτική μέθοδος") to eliminate hypothesis H_0. The ultimate goal of the Socratic method is to increase understanding through inquiry. This is done using creative questioning to dismantle and discard preexisting ideas, allowing the respondent to rethink the primary question under discussion.

That said, we (*i*) "question" existing NFV approaches by conducting relevant experiments using service chains (see § 4.1), (*ii*) propose solutions that aim

to solve the performance degradation problems stated in H_0 (see Chapters 6 and 7), (*iii*) formulate an alternative hypothesis, and (*iv*) evaluate our solutions to provide evidence that steadily show that H_0 leads to contradictions, thus H_1 holds (see § 7.5).

The alternative hypothesis we want to support is as follows:

> H_1: Some service chains can be realized without their performance deteriorating despite the length and complexity of the chain.

4.5 Research Methodology

This thesis project follows a quantitative research approach as specified by John Creswell in [93]. In short, this approach is based on the closed loop depicted in Figure 4.4. First we design, then we implement, and finally we experimentally evaluate our solutions to the targeted problems. Next, we use the observations from the evaluation phase and the technical details from the implementation phase to provide feedback to and refine the design choices of our solutions.

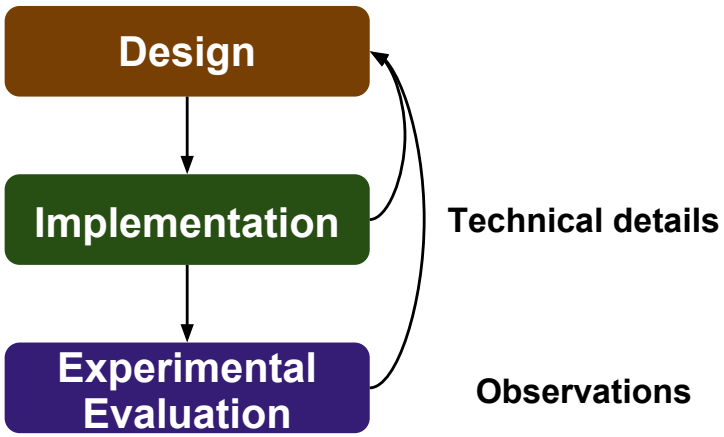


Figure 4.4: The research methodology followed by this licentiate thesis.

Chapter 5

Experimental Setup

This chapter describes the experimental setup followed to deploy, run, and measure the performance of chained NFs.

The testbed consists of six identical machines, each with a dual socket 16-core Intel®Xeon®CPU E5-2667 v3 [81] clocked at 3.20 GHz. The cache sizes are: 2x32 KB L1 (instruction and data caches), 256 KB L2, and 20 MB L3. Hyper-threading is disabled and the OS is the Ubuntu 14.04.3 distribution with Linux kernel v.3.13. Each machine has two dual-port 10 GbE Intel 82599 ES NICs. For the majority of the experiments in this thesis, two out of the six machines are used as shown in Figure 5.1. Machine 1 generates and sinks bi-directional traffic (using different cores), while machine 2 acts as the NFV host, where we deploy our NFV service chains and tools. In machine 2, an entire CPU socket is isolated to ensure that the measurements will not be affected by other competing processes, while all of the system's other functions use the CPUs in the other socket. The total capacity of the testbed shown in Figure 5.1 is 40 Gbps, when bi-directional traffic is injected.

Next, the two main components of the testbed are described in § 5.1 and § 5.2 respectively. The low-level configuration of the testbed is included in this thesis as Appendix A.2.

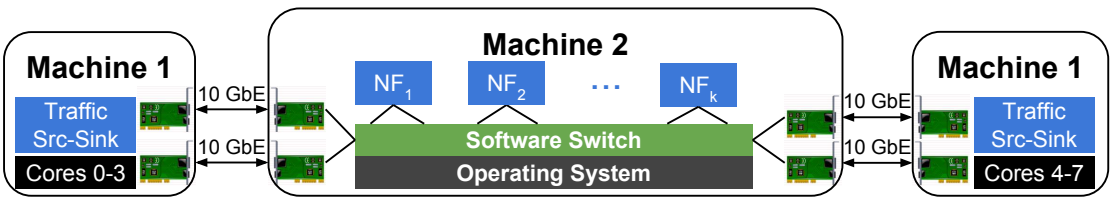


Figure 5.1: The main NFV experimental setup. Machine 1 generates and sinks bi-directional traffic, while machine 2 realizes the chained packet processing.

5.1 Traffic Generator and Sink

Traffic is generated and sunk in machine 1, as shown on the left and rightmost parts of Figure 5.1. Depending upon the target of the experiment, the traffic modules work in different modes as explained below.

Throughput mode is used to measure the throughput of the NFV deployment under test. In this case, machine 1 uses MoonGen [94] to generate and sink the traffic, ensuring that the appropriate number of CPU cores will be allocated to fulfill the capacity requirements of the test. MoonGen is a DPDK-based traffic generator that can saturate a 10 GbE NIC with frames of any size using only one core. Therefore, even for high throughput tests (e.g., at 40 Gbps), machine 1 can use 4 cores to generate traffic and the remaining 4 cores of the same socket to receive the traffic, as shown in Figure 5.1.

Latency mode is used to measure the end-to-end time required to send traffic from machine 1, traverse the NFV deployment under test in machine 2, and receive the traffic back to machine 1. Click is used as a traffic source and sink for this purpose. Before the packets leave the traffic generator, they are annotated with a timestamp that is written in the packet's payload. When the packets return to the sink module (after being processed by machine 2), their timestamp annotation in the packet payload is updated, packets are stored in memory, until being dumped to a pcap file after the end of the experiment. The timestamps are subsequently used to calculate the end-to-end latency.

Regardless of the mode, in which the traffic generator and sink operate, they can be parameterized to handle various traffic patterns using different protocols, such as TCP, UDP, or Internet Control Message Protocol, or even replay traces from pcap files. After the traffic generator has sent all the required packets, it produces a report that denotes the exact rate that packets were pushed to the NIC(s), along with the number of packets sent. The same metrics are reported by the traffic sink, hence by comparing these metrics we assess if the chain exhibited throughput degradation or packet loss.

5.2 Traffic Processor

The packets sent by the traffic generator of machine 1 are received by the NICs of machine 2 as shown in Figure 5.1. In the tests for this thesis we consider two different network drivers to interact with the NICs: (*i*) the standard Linux network driver for Intel 10 GbE NICs, specifically ixgbe version 3.19.1 and (*ii*) the DPDK

network driver version 2.2. As shown in Figure 5.2, five different service chains are considered for the tests of this thesis.

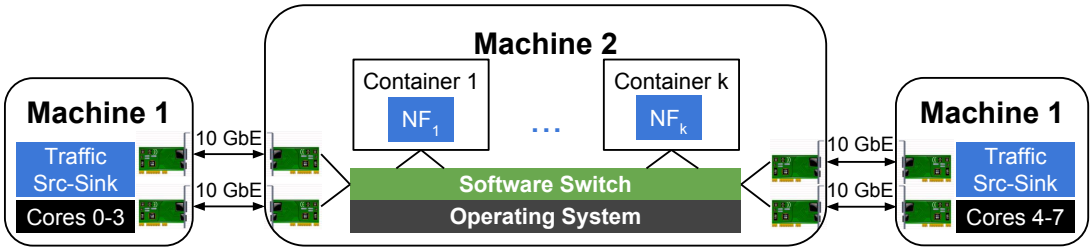
Three of the chain deployment types run on top of a software switch. In these cases, the NFs are user-space processes that run in isolated Linux containers, reading and writing frames from/to virtual ports that are attached to the software switch. Therefore, the switch acts as a backbone network that forwards frames across the chain. If each NF is a separate process, multiple containers are chained as shown in Figure 5.2a. This type of chain is called a "Multi-Process" chain. In an attempt to avoid the I/O communication cost among NFs in a chain, the entire chain is realized in a single process, hence one container. This case is shown in Figures 5.2b and 5.2d, where the difference is whether the process hosts a normal ("Single-Process") or a synthesized ("Synthesized Single-Process") version of the chain respectively. A "Single-Process" chain runs exactly the same code as the "Multi-Process" chain, but all in one process. A "Synthesized Single-Process" chain runs code equivalent in functionality with the other two chains, but from which the redundant operations have been removed. We discuss the way we synthesize service chains in Chapter 7.

Finally, Figures 5.2c and 5.2e show the same single process chains (i.e., normal and synthesized), running natively, without a backbone software switch interconnect. In this case, we remove the virtualization ("V") from NFV, aiming for better performance.

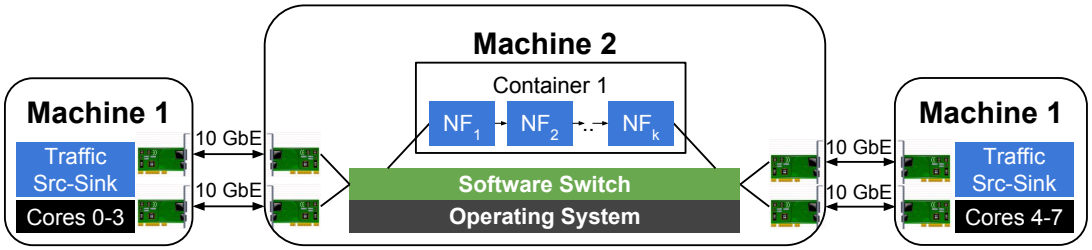
At the end of each chain, an additional pair of NICs sends the frames out of machine 2, back to the origin machine. These NICs act as the gateways of the processed frames to the traffic sink. Depending on the packet rate injected to the chain, only one set of NICs or both sets of NICs can be used by the chains in machine 2. For example, an experiment at 10 Gbps might use the top left and top right NICs, illustrated in Figure 5.1. Another experiment at 20 Gbps might either inject bi-directional traffic to the same two NICs or use all of the four NICs to inject uni-directional traffic. In this thesis we follow the former approach to examine the full capacity of the NICs, when both Tx and Rx operations are applied at the same time. Next, we present the way we implement NFs in the traffic processing machine.

5.2.1 Software-based Packet Processing Framework

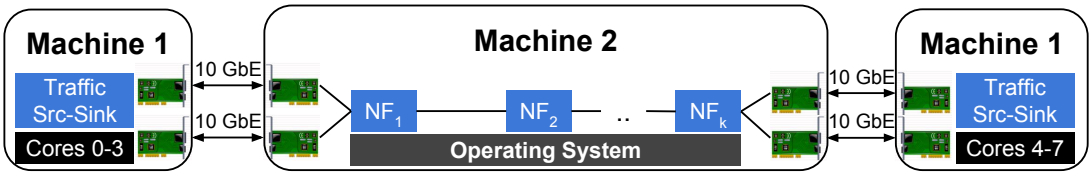
A service chain might consist of one or more NFs. In an NFV context, these NFs are running in software, hence we need a framework to facilitate the realization of NFs, while maintaining high performance. Click [33] is selected as such a framework, because it is the most popular software-based packet processing architecture in the academia.



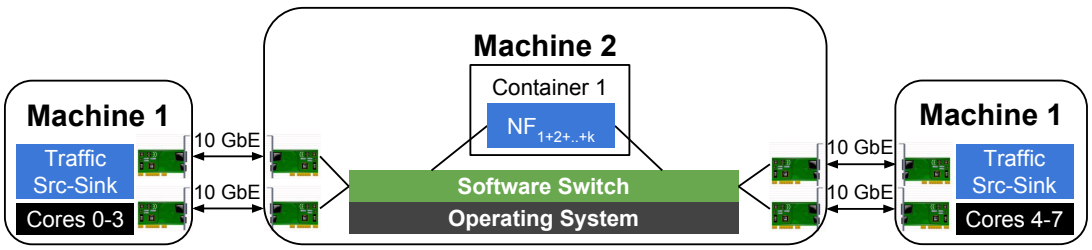
(a) Multi-Process chain of NFs, each running in a different container, on top of a software switch.



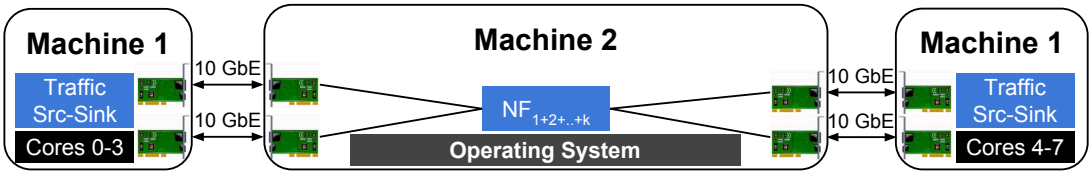
(b) Single-Process chain of NFs running in one container, on top of a software switch.



(c) Single-Process chain of NFs running natively.



(d) Synthesized single-process chain of NFs running in one container, on top of a software switch.



(e) Synthesized single-process chain of NFs running natively.

Figure 5.2: Five deployment types for chained NFs.

As introduced in § 3.1, Click is a modular C++ platform for implementing advanced packet processors by combining primitive packet computation elements. Click elements are combined by forming a Directed Acyclic Graph (DAG) that describes the way packets are read, processed, and written.

The basic NF implementation in our experiments is the Click router depicted in Figure 5.3. We use a slightly modified version of the router implemented by Kohler et al. in [33]. As shown in Figure 5.3, our router does not contain ARPQuerier elements, as we assume that the interconnections from this router to other nodes are static. Hence, we can directly encapsulate (using an EtherEncap element) the IP packet using a predefined gateway's MAC address as a destination field in the Ethernet frame. Note that the router in Figure 5.3 contains two interfaces (i.e., pairs of I/O elements); hence in order to realize the chains shown in Figure 5.2 we need to deploy either two of these routers or a similar router with 4 I/O pairs instead of two.

To implement more advanced NFs, we depart slightly from this implementation by placing a few Click elements at the correct position within a Click DAG. For example, a FW requires an IPFilter element (with one or more traffic filtering rules) between CheckIPHeader and GetIPAddress elements. Overall, we implemented the following four NFs:

1. IP Router;
2. L4 FW;
3. NAPT; and
4. L3 LB;

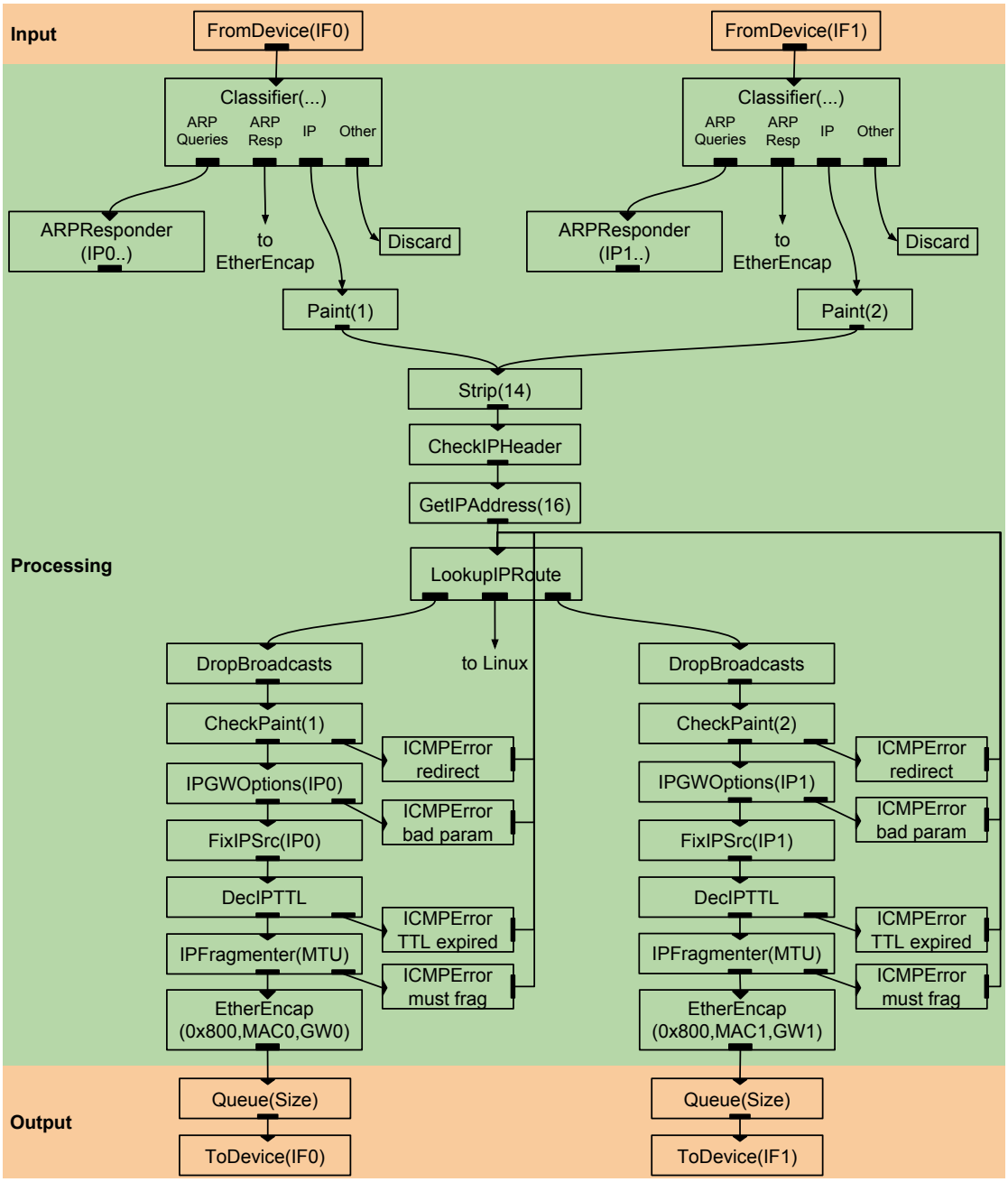


Figure 5.3: A Click implementation of an IPv4 router.

Chapter 6

Profiling and Accelerating Commodity NFV Service Chains with the Service Chain Coordinator

In § 4.1.1 we observed that chained NFs that use the native Linux network driver exhibit high latency. Given the popularity of this driver, we derived the first challenge of this thesis in § 4.2.1. This challenge is to deeply analyze an NFV system that uses the native Linux network driver, during the execution of service chains, to reveal the root causes of the high latency.

In response, the first contribution of this chapter* is an NFV profiler; a tool that collects data from low-level performance counters from the underlying NFV infrastructure to track packets as they move from the NICs to the processors (and vice versa) through the different levels of the system's memory hierarchy. The proposed profiler decomposes the observed per packet latency into components mapped to the involved hardware components (e.g., caches, main memory) and associates these components with their cause(s) (i.e., the responsible pieces of code that cause this latency).

Today, service chains are used by modern services to enrich their data plane functionality. For instance, Amazon offers services that allow tenants to build their own virtual infrastructure by combining functions such as filtering, routing, slicing, and load balancing [86]. In such an environment, even state of the art frameworks, such as ClickOS [49] and NetVM [50], cannot achieve high-performance as reported in § 4.1.2.

*The work described in this chapter is based on the journal article "Profiling and accelerating commodity NFV service chains with SCC" [95] (the authors of the article retained the copyright and give their joint approval for parts of this material to appear in this thesis).

Unfortunately, these latest advancements have not yet been adopted by cloud providers and it is unlikely that this will happen soon, as cloud providers continue to rely on commodity OSs, I/O drivers, and switching fabrics. Although techniques such as single root I/O virtualization (SR-IOV) can bypass the hypervisor and pass packets from the NICs to the VMs [96], cloud applications still use costly system calls to interact with the NICs. These interactions are frequent and consume a large fraction of the execution time of an NFV instance.

In the context of chained services, according to our profiler, I/O is not the only problem, as the length of a service chain imposes serious scheduling overheads. This was the second challenge defined in § 4.2.2.

To address these two challenges, we designed and implemented the Service Chain Coordinator (SCC). SCC employs techniques to adjust the frequency of I/O operations in tandem with adjusting the priority and time quanta allotted to each NF by the scheduler, to maximize the effective run-time of the chain. In contrast to earlier efforts [69, 70], SCC employs a global NFV scheduler to make chain-level decisions, rather than an internal scheduler that executes local switch policies.

In § 6.1, we provide an overview of SCC by formulating the key problem that SCC addresses and quantitatively summarizing our contributions. The testbed used in this chapter corresponds to Figure 5.2a.

6.1 SCC Overview

To solve the first part of the problem tackled by this thesis, as defined in § 4.1.3, we state a key question and the way to address this question.

Key Question: What are the reasons that cause user-space NFV service chains, using commodity OSs and network drivers, to exhibit low performance?

Methodology: In § 6.2 we describe an NFV profiler that (*i*) utilizes low-level hardware and software performance counters to track packets as they move across the system's memory hierarchy, (*ii*) measures the per packet latency of the involved hardware components (e.g., caches and main memory), and (*iii*) associates this latency with the cause(s) (i.e., the responsible pieces of code).

First, we leverage the profiler's power to reveal problems in NFV service chains and quantify their effects in § 6.3. Then, in § 6.4 we accelerate NFV service chains by solving those problems identified by the profiler via an automated run-time called SCC. We illustrate the problems and the solutions realized by SCC in Figure 6.1.

The bottom part of this figure, labeled as "No SCC", shows a typical way user-space NFV applications based on unmodified network drivers interact with the NICs via the OS's kernel. As we show in § 6.3, this causes two major problems related to the key question stated above.

Problem 1: The service chain at the bottom part of Figure 6.1 requires frequent, usually per packet, system calls that cause the service chain to yield the CPU to the OS in order that the latter can perform the necessary I/O operations.

Problem 2: The default Linux scheduler is inappropriate for NFV service chains because it grants short time quanta to the NFV processes and treats them as any other process in the system. As a result, the default Linux scheduler imposes excessive scheduling contention, the latency of which is greater than the actual run-time of a service chain.

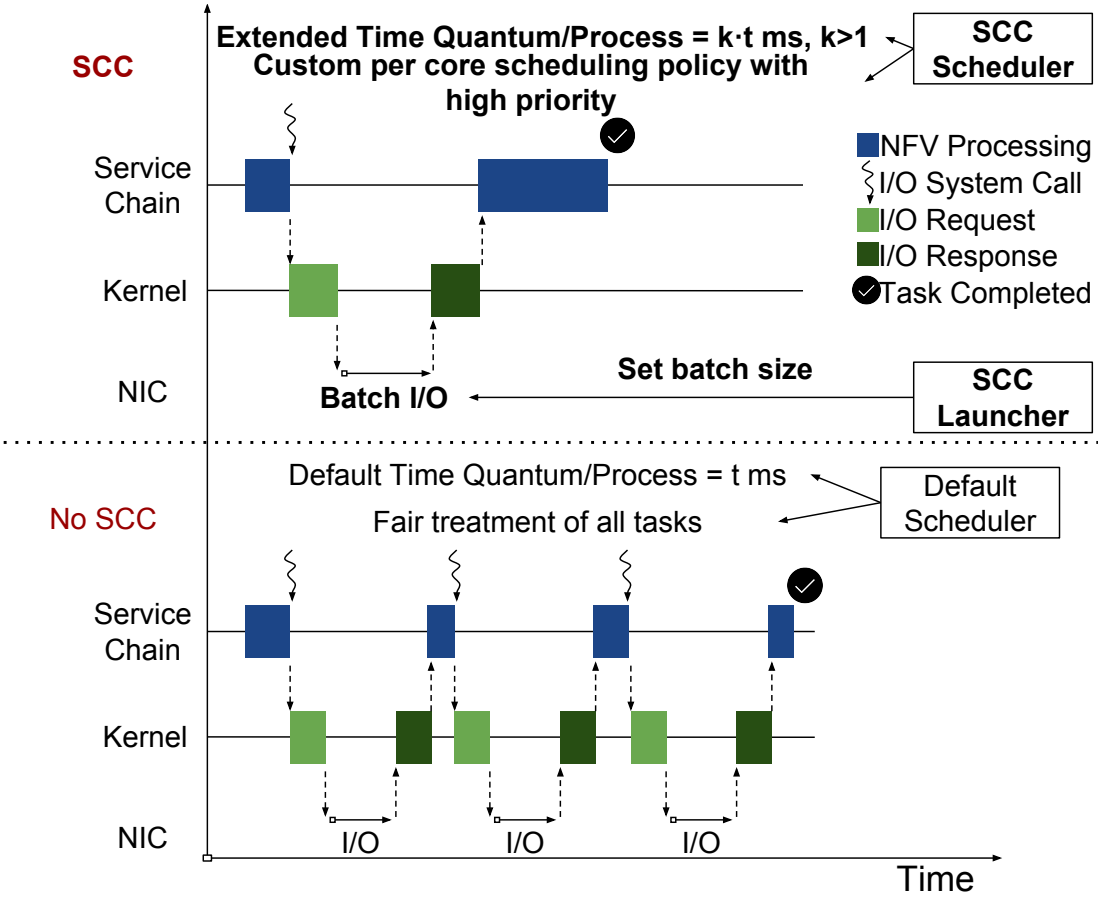


Figure 6.1: The SCC run-time combines (*i*) tailored scheduling for NFV service chains via the SCC Scheduler with (*ii*) fewer (but longer) user to/from kernel-space interactions by multiplexing I/O-related system calls via the SCC Launcher. SCC achieves faster completion time, hence lower latency, than the "No-SCC" case.

These I/O and scheduling problems cause NFV service chains deployed on commodity OSs and network drivers to exhibit high end-to-end latency and latency variance (see § 6.3). To solve these problems, we employ SCC, presented in detail in § 6.4, (labeled as "SCC" in the top part of Figure 6.1) as follows.

Solution to Problem 1: SCC reduces the number of times the path from user to kernel-space and the reverse are used by multiplexing multiple packets into one system call via the SCC Launcher component (see § 6.4.1). Our implementation (available from [97]) builds upon the popular FastClick NFV framework to address the first challenge of this thesis, introduced in § 4.2.1.

Solution to Problem 2: The SCC Scheduler component realizes a suitable scheduling plan to dramatically reduce the end-to-end latency and latency variance of NFV service chains. To do so, it implements single or multi-core scheduling policies for the entire service chain that grant longer time quanta and high priority to the involved processes (see § 6.4.2). This solution addresses the second challenge of this thesis, introduced in § 4.2.2.

We evaluate SCC in § 6.5, but we provide a summary of our findings in Table 6.1. The first column states the comparisons we made throughout this chapter among (*i*) standalone NFs that use different network drivers in user or kernel-space and (*ii*) chained user-space NFs, interconnected either with OVSK or B2B. The second column of Table 6.1 quantifies our observations made in § 6.3 and § 6.5. Note that the first row of Table 6.1 does not show all of the kernel-space overhead as it was not fully quantified in § 6.3.1.

Table 6.1: A summary of the SCC contributions and findings, made in § 6.3 and § 6.5. The evaluation concerns standalone and chained FastClick routers, in different contexts (i.e., user or kernel-space), using different network drivers (i.e., the standard Linux ixgbe and the DPDK drivers), with or without an underlying software switch (either using OVSK or B2B interconnections).

Comparisons	**Findings**
Part of the kernel overhead for a single router using the ixgbe network driver compared to the same router using the DPDK network driver.	Locks (27% of the kernel router's time), 10x more context switches because interrupt-handling pre-emptions destroy cache coherency.
User to/from kernel-space time share with respect to the total time spent by a user-space router using the ixgbe network driver.	User-to-kernel for Tx (32.7%), kernel-to-user for Rx (40.5%) of the user-space router's time.
OVSK overhead, comparing a user-space router with and without OVSK, both using the ixgbe network driver.	14% overhead due to more function calls, lookup cost and additional trips to user-space.
I/O multiplexing benefits for a user-space router using the ixgbe network driver.	3x lower latency and up to 4x lower jitter.
I/O multiplexing benefits for user-space chains using the ixgbe network driver.	Not implemented for chains interconnected with OVSK. 10-40% lower latency and 2x lower jitter for B2B interconnected chains.
Scheduling benefits for user-space chains using the ixgbe network driver.	30-300% lower latency and up to 40x lower jitter for chains interconnected with OVSK. 10-25% lower latency and 2x lower jitter for B2B interconnected chains.

6.2 Profiling NFV Software Stacks

This section introduces the research methodology used for profiling the software stacks of NFV service chains. The NFV profiler presented in § 6.2.1 is used to answer two questions:

Question 1: How does the data flow through the hardware of a commodity NFV server?

Question 2: Which elements of a service chain are responsible for the observed latency?

6.2.1 The SCC Profiler

An NFV profiler must closely interact with both the underlying hardware and the OS, to accurately collect and translate relevant events. Although there are generic tools [75, 74, 77] for interacting with an OS, one has to employ vendor-specific tools to acquire (some of) the hardware events. Additionally, these tools vary between different hardware architectures from the same vendor. Taking into account these facts, we designed the SCC Profiler. This NFV profiler consists of four modules running atop Intel's Xeon architectures and Linux-based OS as shown in Figure 6.2. In the remainder of this chapter we will limit our discussion to the Linux OS and the Intel Xeon processor used in our testbed. In the following sections, we analyze how the SCC Profiler keeps track of data by establishing software and hardware bindings with the relevant counters of our testbed.

6.2.1.1 Software Monitoring

We use Perf [75] to access performance counters of various parts of the Linux kernel. The SCC Profiler passes the Process Identifiers (PIDs) of the NFV service chain to Perf asking for a variety of events (labeled as "Perf+Linux Kernel" in Figure 6.2).* By querying the counters of the devices' skbuffs and network I/O-related system calls, the SCC Profiler learns the number of packets sent/received by the devices and the number of system calls required for these I/O operations.

The Linux scheduler provides counters regarding the execution of each NF of the service chain. The SCC Profiler retrieves the number of CPU migrations and context switches as well as the active, waiting, and blocking times of each NF. As shown in Table 6.2, using our custom OS benchmarks (available at [98]), we found that the context switching time between two processes scheduled using the

*When the system's configuration indirectly involves CPU cores that are not used by the PIDs of the NFV processes, the SCC Profiler can be instructed to monitor these additional cores. For example, this might happen if an NF is pinned to a core, but the interrupts of the NICs used by this NF are served by another core.

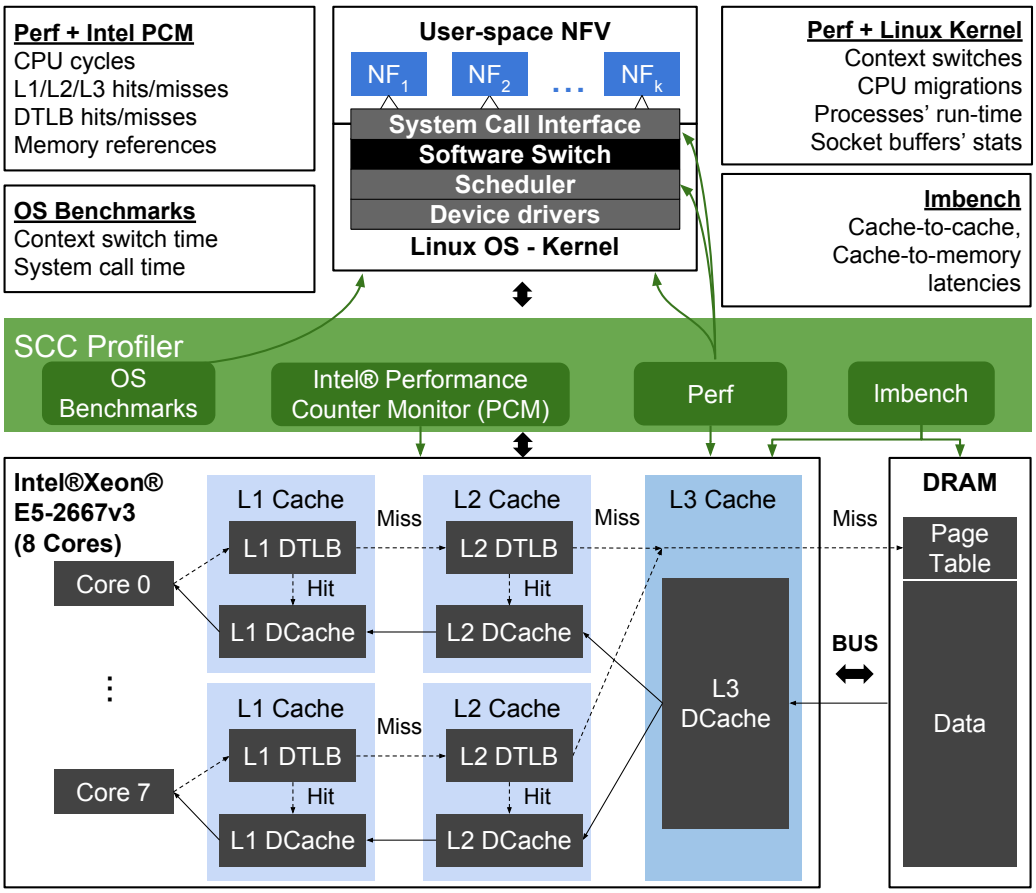


Figure 6.2: The SCC Profiler. lmbench measures the latencies to access each part of the memory hierarchy. The SCC Profiler combines the latencies from lmbench with (*i*) the hardware counters obtained by Intel's PCM and Perf and (*ii*) the software counters obtained by Perf and OS benchmarks, to measure run-time NFV performance and generate a report of costly operations.

default Completely Fair Scheduler (CFS) [99] (with the default priority) is roughly 1000 ns, while this time is 940 and 1140 ns when using the real-time First In, First Out (FIFO) & Round-Robin (RR) and batch scheduling policies respectively. Moreover, the Linux kernel requires 40 ns to execute a network I/O system call (i.e., a socket read or write) in our system.

Combining this information with the counters above, the SCC Profiler calculates the latency (per packet) due to the OS when providing basic I/O services (i.e., read and write system calls) to the NFV processes and to coordinate the execution (i.e., schedule) of the service chain. The next target is to capture how the underlying hardware executes the kernel's instructions and how efficiently these instructions pass the packets through the NFV pipeline.

Table 6.2: Latencies (ns) for a system call and a context switch under different scheduling policies (with default priorities) of the Linux kernel.

OS-level latency source	Latency (ns)
Context switch - Default CFS	1000
Context switch - Batch CFS	1140
Context switch - Real time Round-Robin scheduler	940
Context switch - Real time FIFO scheduler	940
Network I/O System Call	41

6.2.1.2 Hardware Monitoring

The bottom part of Figure 6.2 depicts the elements of one of the CPU sockets and the memory system of the NFV host machine.* There are two types of arrows in this part of the figure. The dashed arrows show the flow of the virtual address translation procedure in our processor's memory management unit. This translation occurs when a program requires a memory access. In this case the CPU passes the virtual address, used by the program, to the memory management unit asking for a mapping (stored in the OS's page table in the main memory) of this address to physical memory.

Going to memory for translation information before every instruction fetch or explicit data load/store would be prohibitively slow, therefore modern processors employ specialized hardware caches, known as Translation Lookaside Buffers (TLBs), that make a portion of the page table accessible at the speed of the processor, hence speeding up the address translation procedure for addresses with entries in the TLB. Our processor uses a hierarchy of TLBs at the first two (i.e., L1 and L2) cache levels†, hence a TLB miss will only cause an access to the page table in main memory if neither of the two TLBs contains the mapping. Upon a Data Translation Lookaside Buffer (DTLB) hit, the physical page and offset are fetched and the data moves from the respective cache (or the main memory) to the processor following the solid lines in the bottom part of Figure 6.2.

Moreover, our processor exploits the benefits of Intel's DDIO [28] technology as explained in § 2.5. The LLC portion used by DDIO can be up to 10% of the LLC's capacity, which in our case results in 2 MB [28]. However, as we will see from our measurements, doing so is a challenge for NFV services that rely on the usual Linux network stacks and drivers.

*The L1 instruction cache is omitted for readability reasons and because the miss rate of this cache was negligible in all of our experiments.

†The L1 cache also contains an instruction TLB.

To understand exactly what access delays an application will experience, it is crucial to measure the access costs to all the hardware components of Figure 6.2. Specifically our goal is to quantify the latency when data moves across the memory hierarchy, enabling us to pinpoint the bottlenecks of NFV software stacks. This will allow us to optimize the NFV implementation, to meet stricter latency requirements and to achieve high throughput.

The SCC Profiler uses lmbench [72] to measure the latencies of all the components of the underlying system's memory hierarchy. Figure 6.3 shows these latencies as measured in our testbed. lmbench initiates read and write transactions of progressively increasing array sizes (i.e., 1 KB-2 GB) that can eventually fill all of the caches and part of the main memory, to measure the latency of the following transactions:

Local: from a core to its local L1 and L2 caches.

On-chip: from a core to the shared cache (e.g., L3 cache in our case) or the L1/L2 cache of another core in the same socket.

Off-the-chip: from a core to main memory.

Note that the line size of our caches is 64 bytes. This is almost the size of the smallest Ethernet frame. A stride size equal to the cache line's size implies one hit per cache line, following this the latency to access the different parts of the same cache line is low. However, input data in reality might exhibit different access patterns in terms of size, hence we further increased the stride sizes, to the size of the standard Ethernet maximum transfer unit (i.e., 1500 bytes), up to the size of a jumbo Ethernet frame (i.e., 9000 bytes) to measure its effect on the latency. This way we emulate an skbuff that holds a frame equal to the size of our stride.

Figure 6.3 shows that L1 latencies are not affected by the size of the stride (with a constant access time 1.18 ns), while L2 latency exhibits low variance with respect to the stride size (between 1.25 and 5 ns). However, comparing the fastest (i.e., 64 bytes) and slowest (i.e., 1024 bytes) stride sizes we see that L3 cache and main memory latencies increase almost 10x (between 1.3 and 14.3 ns for the L3 cache and between 7 and 71.7 ns for main memory), although the smallest (i.e., 64 bytes) and largest (i.e., 9000 bytes) stride sizes exhibit a factor of 140.6x difference in the size. The reason behind this is that the hardware executes prefetch requests, in parallel with the current data processing, to bring cache lines from the next higher level store into the current cache before it is actually needed, thus allowing data access overlap with pre-miss computations. In addition, if there is no dependency between the data to be loaded, our CPU can issue multiple instructions to fetch independent chunks of data in parallel. These techniques hide part of the memory access latency, leading to decreased access latency as observed in Figure 6.3.

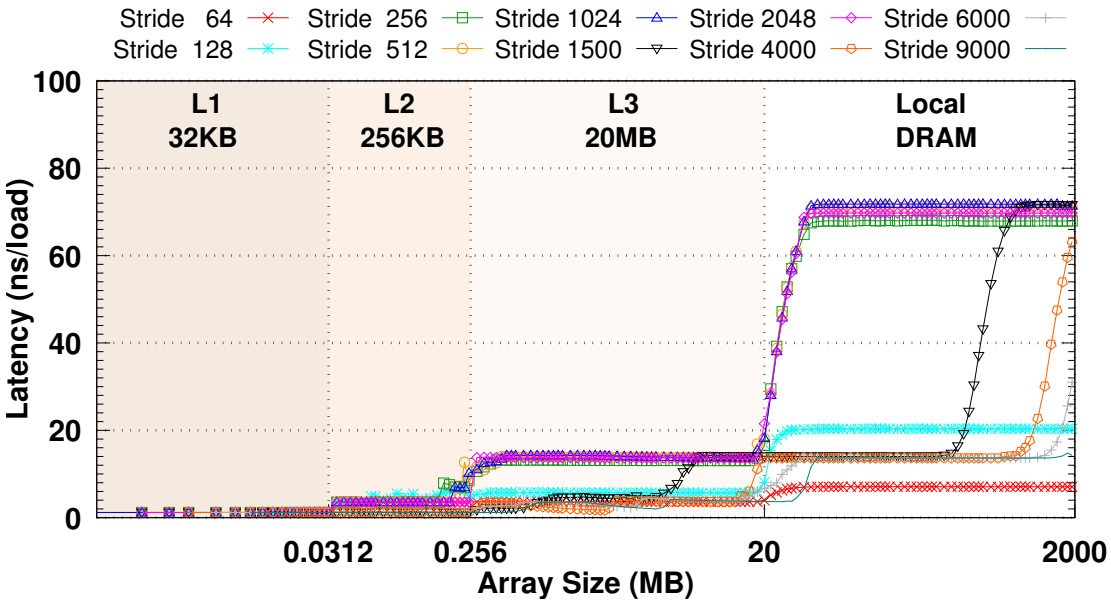


Figure 6.3: Latencies to access a progressively increasing array size (1 KB-2 GB) on different parts of the memory hierarchy versus different stride sizes in bytes for an Intel® Xeon® CPU E5-2667 v3 clocked at 3.2 GHz.

According to Larry McVoy and Carl Staelin [72], small stride sizes are expected to have higher spatial locality than large ones, hence prefetching is likely to bring useful data into the current cache. In contrast, the poor spatial locality of large stride sizes might cause the system to prefetch useless data, leading to increased latency. Although these statements sound instinctively correct, they do not hold for stride sizes that are not a power of two (i.e., 1500, 4000, 6000, and 9000 bytes), as we see in Figure 6.3. Specifically, we notice that the latency in ns/load for these stride sizes is (*i*) lower than the latency for a stride size of 128 bytes, for some array sizes beyond the L3 capacity (i.e., more than 20 MB), and (*ii*) comparable to the levels of the L3 cache access latency (i.e., around 14 ns) of smaller stride sizes, such as 256, 512, and 1024 bytes. This phenomenon persists for larger array sizes in main memory (i.e., for array sizes greater than the LLC size), as we increase the stride size to 4000, 6000, and 9000 bytes, but it does not happen at all for a stride size of 2048 bytes (which is a power of two).

We also notice that the largest stride size (i.e., 9000 bytes) exhibits latencies comparable to those of the smallest stride size (i.e., 64 bytes), while for some array sizes beyond the L3 capacity (between 22 and 36 MB) this latency is even lower that the latency of the 64-byte stride size. These results show that an increasing stride size does not always imply a higher memory access cost and that hardware prefetching and parallel fetching of multiple memory blocks might be equally or

more beneficial for large stride sizes that are not a power of two, than for (small) stride sizes that are powers of two. We believe that our observations complement the findings of [72], which are based on an older Intel processor than ours, showing that modern hardware architectures have substantially improved cache efficiency leading to lower memory access latencies.

Finally, having quantified the latency to access the hardware components of Figure 6.2, the SCC Profiler collects a set of run-time performance monitoring events from the underlying Intel processor. The complete list of available events is available at [100]. More information is provided as Appendix A.1. Specifically, during the execution of an NFV service chain we acquire CPU core, L1, and Dynamic Random Access Memory (DRAM) events using Perf, while L2 and L3 events are fetched using Intel's PCM tool. To capture the data movements in the bottom part of Figure 6.2, we monitor the number of hits and misses of load, store, and prefetch operations for all of the caches (including the DTLBs), and the number of accesses (load and stores) that occur in the DRAM. These events are labeled as "Perf+Intel PCM" at the top left box of Figure 6.2.

6.2.1.3 Latency Calculation

The software and hardware monitoring strategies of the previous sections provide enough data to the SCC Profiler for it to project the collected counters on a per packet scale and to calculate the total per packet latency incurred by our NFV server, with respect to the injected load.

Using the primitive latency values from Table 6.2 and Figure 6.3, we compose a variety of latency factors (shown in Table 6.3). To calculate the total per packet latency, we sum the (*i*) data access and (*ii*) address translation latency factors of each memory level (namely the L1, L2, L3 caches, and main memory), along with the context switching and system call latencies per packet of each component (i.e., process) of the service chain. The latter factor (i.e., system calls) does not sound as important as the other latency sources; however, in practice, an NFV service chain might be comprised of multiple NFs, usually each NF is deployed as a separate process (e.g., considering a chain as a set of VMs or containers), hence a read/write system call per packet per NF might add up a considerable latency.*

Note that according to Figure 6.3 the latency of a hit depends on the workload. Under realistic scenarios the incoming traffic will exhibit variable frame sizes, hence all these latencies are possible. For this reason, we instructed the SCC Profiler to use the worst case latency hit for each memory level as per Figure 6.3, because (*i*) these cases occur for several input data sizes (they are not corner cases), and (*ii*) their contribution is ten times greater than other input data sizes, hence only

*In this chapter we do not consider a synthesis of a single NF from a chain of NFs as we do in [101] (see also Chapter 7).

a few of these cases might contribute a large latency, the cause of which we do not want to ignore. The number of packets, processed by each NF and in total, are counted by the software monitoring process of the SCC Profiler (see § 6.2.1.1). Consequently, using the notation from Table 6.3, the mathematical formula to compute the total latency per packet is as follows:

$Latency/Pkt = L_{D1} + L_{D2} + L_{D3} + L_{DM} + L_{T1} + L_{T2} + L_{TM} + L_{CS} + L_{SC}$

This formula captures the latencies from all the involved hardware components of our system and a portion of the latency added by the OS. In § 6.3.2, we analyze a service chain and explain why there are other hidden costs, added by the OS, that are hard to accurately quantify on a per packet scale, although we manage to indirectly reveal their impact. Another important detail regarding the formula above is that when DDIO is utilized by the NICs, frames are exchanged directly with the LLC (i.e., L3 cache in our case), but this does not prevent a slow NFV system from interacting with the main memory as explained in § 6.2.1.2. In § 6.3 we show that NFV service chains based on unmodified Linux network drivers destroy cache coherency and eventually end up using main memory as they touch a larger number of memory locations than can stay in the LLC.

Finally we clarify that the SCC Profiler operates in counting mode which means that the counters are aggregated values collected during the execution

Table 6.3: Latency calculation formulas and notation for each source of latency in a service chain. The latencies of Table 6.2 and Figure 6.3 are used in the formulas.

Latency/packet		**Formula**	**Notation**
Data Access	**L1**	$L1Hits/Pkt \cdot L1DCacheHitLat$	L_{D1}
	L2	$L2Hits/Pkt \cdot L2DCacheHitLat$	L_{D2}
	L3	$L3Hits/Pkt \cdot L3DCacheHitLat$	L_{D3}
	DRAM	$L3Misses/Pkt \cdot MemHitLat$	L_{DM}
Address Translation	**L1**	$L1DTLBHits/Pkt \cdot L1DTLBHitLat$	L_{T1}
	L2	$L2DTLBHits/Pkt \cdot L2DTLBHitLat$	L_{T2}
	DRAM	$L2DTLBMisses/Pkt \cdot MemHitLat$	L_{TM}
Context Switching		$\sum_{i=1}^{n} ConSw_i/Pkt_i \cdot ConSwLat_p$, where n is the number of processes and p the scheduling policy	L_{CS}
System Calls		$\sum_{i=1}^{n} SysCalls_i/Pkt_i \cdot SysCallLat$, where n is the number of processes	L_{SC}

of an experiment. The SCC Profiler resets all the relevant counters before the experiment and collects their values at the end of the experiment. Hence these values do not involve sampling techniques or other approximation methods. Moreover, since we believe these counters of hardware-based events have high accuracy, we utilize their values as much as possible. For example, Perf does not query the memory controller to collect the main memory references, but rather uses a software-based event to estimate this number. Since all the L3 cache data misses in our system end up in main memory, so do the DTLB misses at the L2 cache, hence we can *infer* the number of main memory references by adding up these two hardware-based counters. However, we did not find a substantial difference between Perf's estimates and the values obtained from the hardware counters. As for the ability of the SCC Profiler to time the functions of the NFV stack, we exploit Intel's high-precision event timers [102] via Perf to acquire the entire list of functions together with their contribution to the total latency.

6.3 Uncovering NFV Performance Problems with the SCC Profiler

We examine the usability of the SCC Profiler by performing a measurement campaign for both standalone and chained NFs in § 6.3.1 and § 6.3.2 respectively.

6.3.1 Standalone NFs

We implemented an NF using FastClick. We focus on the basic router shown in Figure 5.3. We measured the performance of this router running in four different environments (see Figure 6.4) to establish a baseline, in terms of the resource requirements, of our NF. First, we deploy the router natively both in the Linux kernel as well as a user-space application, and tie its ports to the physical 10 GbE interfaces of our NFV server. Then, we measure the same router running as a user-space application in a Linux container. This container is attached to OVSK where it reads/writes frames from/to. Finally, although we target NFV applications that use the native Linux driver for I/O, we also deployed the same router using FastClick's DPDK [24] I/O elements (using the DPDK network driver) to examine the highest achievable performance.

Figure 6.4 shows the latency of the router in these four different environments, using a single CPU core. We injected 5 million frames at an input rate of 0.82 Mpps using a frame size of 64 bytes (without counting the trailing frame CRC that we assume will be computed by the NIC itself). We chose a small frame size to impose more work on the CPU core, and hence better stress the different I/O mechanisms. The reason behind the selection of this packet rate is because

at this packet rate we easily saturate a 10 Gbps link using a 1500-byte frame; hence, to maintain the same workload on the NF, regardless of the frame sizes we want to test, we use this same packet rate for all the different frame sizes in our experiments (see § 6.5). Also, this packet rate conveniently matches the maximum rate that our slowest router (the user-space router, attached to OVSK, using the native network driver) can sustain without dropping packets.

From Figure 6.4 we can distill several interesting findings:

Finding 1 The different network drivers clearly affect the performance of the router. A router with DPDK interfaces imposes almost 8x lower median latency than the kernel-space router with the native network driver, despite the fact that in the former case all of the NFs' code is running in user-space.

Finding 2 A user-space router imposes 4x greater median latency compared to its kernel-space counterpart when both use the same native network driver (i.e., ixgbe).

Finding 3 Attaching the user-space router with the native network driver to OVSK adds ~10% more median latency compared to the same user-space router without OVSK, with the lower latency percentiles of these two routers exhibiting a larger difference.

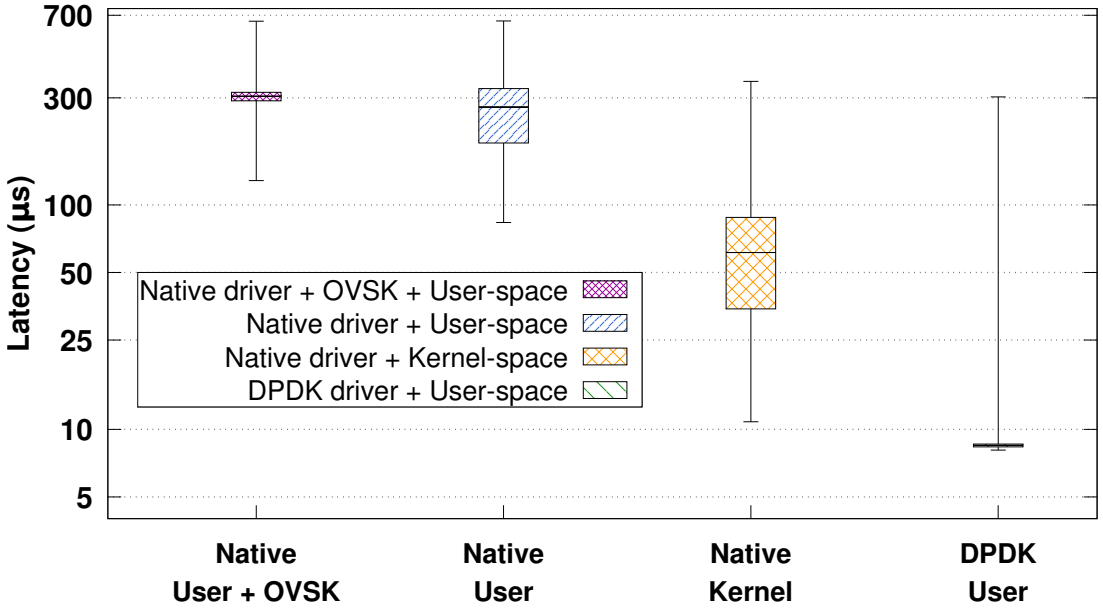


Figure 6.4: End-to-end latency (μs), plotted on a logarithmic scale, for 64-byte frames through four FastClick routers, each running in a different I/O context in a single core as stated in the legend. The input packet rate is 0.82 Mpps.

The first challenge when exploiting these results from the SCC Profiler is to pinpoint the exact cause(s) of these differences. Running the same experiment under the supervision of the SCC Profiler leads to the results shown in Table 6.4. The second column of this table depicts the total per packet latency calculated by the SCC Profiler, following the methodology described in § 6.2.1.3. Comparing these numbers with the results of Figure 6.4, we can safely state that the SCC Profiler reliably tracks the expended time, as the calculated latency falls within the range of the actual per packet latencies measured by the traffic sink. One interesting observation that arises from this comparison is that the latencies calculated by the SCC Profiler usually fall close to the lower percentiles of the actual latencies (except for the DPDK router where the median and low latency percentiles almost match), showing how "lucky" packets (i.e., those packets that experienced no additional delays) move across the different memories.

To associate the calculated latency with the caches and main memory, we also present the number of memory references per packet and the share (%) of the total latency spent in each memory level as the third and fourth columns of Table 6.4 respectively. Note that (*i*) DTLB statistics are not present, but the DTLB cost can be inferred by subtracting 100 from the sum of the percentages of the last column and (*ii*) we omitted the latencies imposed by the number of context switches and I/O-related system calls per packet (as per Table 6.3) to preserve the readability of the table. The former numbers are nearly zero because only one router is executed by this core; hence almost no context switches occur. The latter numbers make a negligible contribution to the overall latency as this router executes at most 2 I/O-related system calls (i.e., receive and send) per packet.*

*The memory of the DPDK router is mapped to user-space, hence the router does not apply the standard receive/send system calls.

Table 6.4: The latency in μs (column 2), calculated by the SCC Profiler, while tracking the packets injected during the experiment shown in Figure 6.4. Columns 3 and 4 show the number of memory references per packet and the share of the total latency imposed by each memory level.

Routers	Per Packet		
	Latency (μs)	L1/L2/L3/DRAM References	L1/L2/L3/DRAM Latency (%)
User +OVSK	259.14	3836/182/70/3557	1.71/0.25/0.37/97.06
User	216.97	3653/179/65/2964	1.99/0.29/0.41/96.60
Kernel	25.45	581/23/14/340	2.73/0.31/0.77/95.70
DPDK	8.01	3250/95/163/7	47.35/0.04/27.16/0.06

6.3.1.1 Root Cause Analysis

In this section we interpret the results shown in Table 6.4, seeking bottlenecks that can be overcome to achieve more efficient realizations. The reason that the DPDK-based router has the highest performance is that there is almost no data exchange between the core that executes the router and main memory. This router is clearly fetching/placing the DMAed packets from/to its L3 cache (since our processor uses DDIO) and it processes the majority of them using its L1 cache, as 47.35% and 27.16% of the total latency are due to hits in the L1 and L3 caches respectively. Note that 47.35%+27.16%=74.51%, hence 25.49% come from elsewhere - specifically 25.39 from DTLB hits, while only 0.04% from L2 and 0.06% from main memory. As explained in § 6.2.1.2, despite the usage of DDIO, main memory references are still possible. The DPDK router experiences DTLB misses at the L2 cache (i.e., the last DTLB) resulting in 7 main memory references per packet, which is only 0.06% of the total latency.

The stack of functions reported by the SCC Profiler showed that the router spent 73.7% of its total time executing I/O instructions, while the remaining 26.3% was dedicated to processing. Part of the I/O time was spent by memory-related functions, such as "clear_page_c_e" (8.4%) and FastClick's "Packet::make" (0.009%). The latter function is so cheap because DPDK pre-allocates a pool of frames that are immediately available to the application, hence the OS spends time only to reset the contents of the frame pool by calling the former function. The remaining I/O time was consumed by the FromDPDKDevice (83%) and ToDPDKDevice (8.6%) FastClick I/O elements. The "run_task" function of the FromDPDKDevice element calls DPDK's "rte_eth_rx_burst" function, which in turn calls the DPDK poll mode driver's "ixgbe_recv_pkts_vec" and "_recv_raw_pkts_vec" functions in order to load a batch of frames from the Rx queues of the NIC to the ring buffers that are mapped to the LLC. This batch is turned into a batch of FastClick frames (without a memory copy), then pushed to the output through the FastClick pipeline, following a "run-to-completion" model.

To realize this model, packet reception is separated from processing and transmission, running as an individual task. FastClick ensures that the reception task will immediately deliver input packets to the pipeline by polling the NIC. Even when the Rx queues are empty, the underlying DPDK framework executes pause instructions (via the "rte_delay_us") for a short period of time to keep the CPU (hoping that frames will arrive soon), thus reducing the number of context switches that might destroy cache coherency. This is why the Rx operations dominate the total I/O time. The pause instructions consumed 7% of the FromDPDKDevice element's time, while the remaining time was spent by the "_recv_raw_pkts_vec" (87.2%) and "ixgbe_recv_pkts_vec" (5.8%) functions to poll the NIC. After the FromDPDKDevice element, the FastClick driver calls the processing elements of

the pipeline to apply the routing NF with a batch of frames, expending 26.1% of the router's total time. After the last processing element, the driver calls the "push_batch" function of the ToDPDKDevice element, which simply places the batched frames into the Tx ring buffer.

The behavior of the kernel-based router is different from the DPDK router. The latency contributions of the different memory components of the kernel-based router reveal the major involvement of the main memory, since 95.7% of the total latency is due to the 340 references per packet to main memory. The kernel-based router involves two threads: one thread receives frames (stored in skbuffs) at interrupt time and stores their pointers in an internal queue (acting as a producer), while the second thread - that does not operate at interrupt time - emits the available frames of the internal queue into the subsequent elements of the pipeline (acting as a consumer) to realize the processing and output of the NF. In this execution model, the queue, and consequently the skbuffs where the queue points to, are the critical sections between these two threads. Moreover, the producer's thread executes at interrupt time, which means that it has a higher priority than the consumer's thread.

Looking into the SCC Profiler's report we found that the main sources of latency in the case of the kernel-based router were the kernel functions "_raw_spin_trylock_bh" and "_raw_spin_unlock_bh", as well as the network driver's function "ixgbe_xmit_frame_ring". The first two functions cause the CPU to execute a busy waiting loop in the kernel until a lock of the internal queue (which is initially kept into the LLC, but eventually evicts data to main memory as we explain below) is acquired or released respectively, consuming 4.26% and 23.1% of the total CPU cycles spent by the kernel-based router (almost 30% of the router's time). The reason that the lock takes so much time to unlock (i.e., 4.26% is 5.5x less than 23.1%) is because, in the meantime, the kernel thread of the router's driver processes and transmits (invoking the driver's "ixgbe_xmit_frame_ring" function) a set of frames, leaving free space for new frames to arrive. Processing and transmission procedures expend ~13% and 6% of the total time respectively, filling the time gap between the lock and unlock functions.

Since the router utilizes only one core, we believe that this execution model heavily involves main memory because the interrupt-based frame reception is frequently preempting the processing and transmission task, causing the newly-arrived input frames to evict the currently processed frames from the LLC to main memory, destroying the cache coherency. The SCC Profiler found that the kernel-based router imposes 10x more context switches than the DPDK router (i.e., 7710 vs. 841), supporting our reasoning. Taking into account that this router processes 32 packets each time the processing task is scheduled, the above number of context switches implies one context switch, thus one flush operation in the core's local

caches, every 19 sets of packets. The DPDK router requires only one context switch every 186 sets of packets.

The next challenge is to quantify the difference between the kernel and user-space routers when utilizing the same native driver. Looking at the user-space router without OVSK, we observe that each basic I/O function of the router comprises of a long list of function calls that start from the user-space FastClick I/O functions, dive into the kernel, and end up at the same driver functions used by the kernel-space router. Hence, we can directly calculate this extra overhead of crossing the user-space border when executing NFV tasks, by simply measuring the time spent by these functions. As we analyze below, this time is on average 1397 cycles per packet for a receive and 1120 cycles per packet for a send operation.

To transmit a frame, user-space FastClick begins by calling the RouterThread driver, which calls the "run_task" function of the ToDevice FastClick element, pulls a frame from the queue, and calls the "sendto" system call. This system call enters the libc system library and is translated into a sequence of 5 socket functions until the data is passed to another set of nested functions that eventually allocate an skbuff and call the driver's single-frame Tx function ("netdev_start_xmit"). In this path, locking (the same as occurs in the kernel-based router) and memory copy/allocation mechanisms (there is a copy from user to kernel-space memory) are present leading to a per packet transmission cost of 32.68% of the total number of cycles spent by the user-space router. Associating this percentage with the latency of the user-space router as measured by the SCC Profiler (216.97 μs), we can compute the user to kernel-space frame transmission overhead as 70.9 μs per packet. The overhead for the reverse path will be described next.

We followed the same methodology at the receive side of the router, by following the invoked function calls and accumulating their costs. We found that frame reception from kernel to user-space adds another 87.76 μs, with these calls corresponding to 40.45% of the router's total number of cycles. This suggests that 32.68% + 40.45% = 73.13% of the router's total number of cycles are spent on the overhead of executing read/write frame operations from/to user-space to/from the kernel-space driver. Frame reception is more expensive than frame transmission for two reasons: (*i*) the application has to allocate user-space memory in order to accommodate the received frame which has to be copied from the skbuffs hosted in the kernel's memory area, and (*ii*) FastClick computes a timestamp for each received frame. Both operations invoke extra, *per packet system calls* using the malloc, memset and ioctl commands. Out of the 7.77% difference between reception (40.45%) and transmission (32.68%) costs, the former memory-related reason occupied the vast majority of the time (91% versus 9% for timestamping).

The summary of the latencies of the reception (87.76 μs) and transmission (70.9 μs) operations of the user-space router is 158.66 μs. Subtracting this number

from the total latency of the user-space router (with the native driver) measured by the SCC Profiler (216.97 μs), we end up with 58.31 μs, which is almost exactly the median latency of the kernel-space router (61.34 μs from Figure 6.4). This leads us to believe that our profiler did an accurate job in identifying the functions involved in the user-space packet processing along with their individual costs.

Finally, the SCC Profiler's report, collected while profiling the user-space router attached to OVSK, showed a similar behavior as the user-space router without a software switch interconnect. Without delving into details, OVSK intercepts every packet heading/originating to/from the NIC, contributing to (*i*) a longer list of called functions, (*ii*) additional trip(s) from the kernel to user-space OVS module (when the packet does not exist in the caches kept in the kernel), and (*iii*) a lookup cost to find the destination port (i.e., virtual interface) of the packet. The SCC Profiler found that the lookup was performed entirely in main memory and quantified that all these factors add up a 19% more latency compared to the user-space router without OVSK. This difference is reflected by looking at the main memory references of the two routers (15% more references for the OVSK router than the user-space router without OVSK) in Table 6.4.

6.3.1.2 Lessons Learned

To summarize the above study, we quantified the performance difference between a state of the art NFV router using the DPDK network driver and the kernel-based router using the native Linux network driver. We found that locking mechanisms and interrupt handling in the kernel reduce the NFV performance; this is why FastClick adopted DPDK's "run-to-completion" approach, as polling requires at most LLC accesses, keeping the caches hot without involving time consuming locks. One could avoid the interrupt costs of the kernel-space router by using the PollDevice Click element. We did not use this element because it works only for a limited set of (old) network drivers.

As for the difference between the user-space and kernel-space routers, when both use the same native driver, we found that the memory allocation and copying between user and kernel-space are the main sources of latency. Despite the fact that user-space Click uses a smart polling mechanism to interact with the NICs, the per packet cost is still high as the polling is not very aggressive. These problems were discussed in earlier work by Luigi Rizzo [25], but without much evidence; thus one of our contributions is proving this evidence.

To accelerate the kernel-space router, one must employ polling of the NICs and avoid the kernel's locking mechanisms. To achieve better performance for a user-space router, while still using the native network driver, one has to minimize the interactions between the user-space application and the kernel, e.g., applying a system call to an entire batch of frames and to use pre-allocated pools of packet buffers to avoid the cost of dynamic memory allocations.

6.3.2 Chained NFs

Modern cloud services are comprised of multiple components, often chained together using an underlying switching fabric. Using the NF of the previous section as such a component, we create chains of 1-8 user-space routers, each running in a Linux container on top of a software switch. We chose OVSK (profiled in the previous section) as it is one of the most popular and generic software switches.

We injected the same amount of traffic as in the standalone NF case (see § 6.3.1) and pinned all of the routers to one isolated CPU core. We also scheduled OVSK in a different CPU core in the same socket. The boxplots of Figure 6.5 show the latency of each chain versus the chain's length, as measured by the traffic sink. The points of Figure 6.5, to the right of each boxplot, represent the latency of each chain as measured by the SCC Profiler. This result demonstrates that the SCC Profiler cannot only track a single NF (as shown in § 6.3.1), but is capable of accurately tracking a chain of NFV processes.

Next, we perform a root cause analysis to explain why the latency increases with the length of the chain.

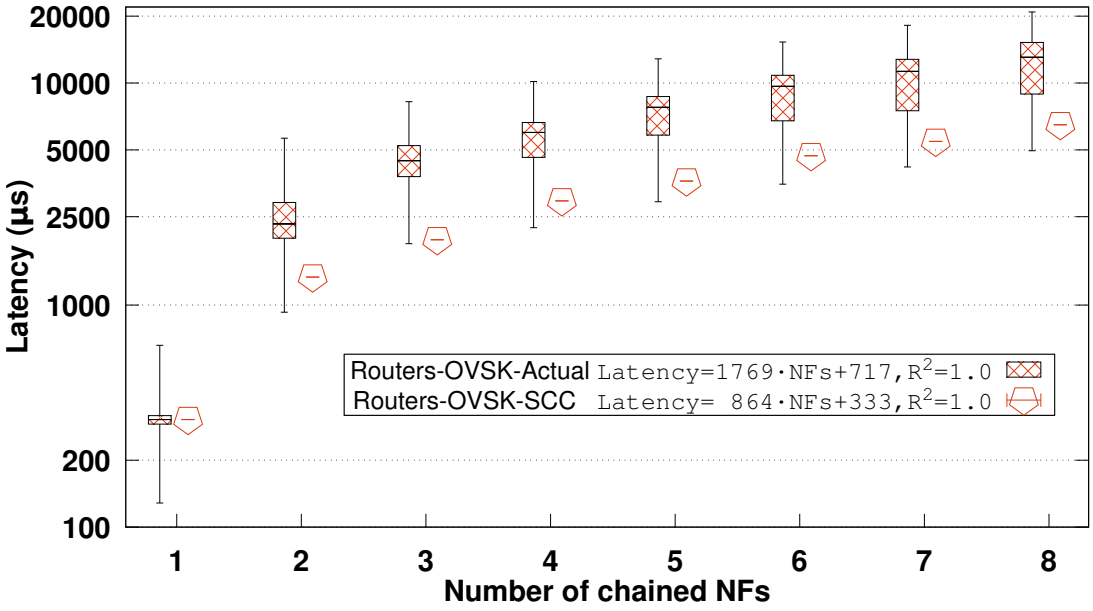


Figure 6.5: End-to-end latency (μs), plotted on a logarithmic scale, (i) measured at the traffic sink (boxplots) and (ii) calculated by the SCC Profiler (points), versus the chain's length for user-space FastClick routers, running in containers on top of OVSK. The routers run in a single core and OVSK runs in a different core in the same socket. The input rate is 0.82 Mpps with 64-byte frames. The linear fit to the median latencies, stated in the legend, begins from the chain with 2 NFs.

6.3.2.1 Root Cause Analysis

To visualize the latency of each chain, we fitted the median latencies measured by the traffic sink and the latencies calculated by the SCC Profiler, leading to the equations shown in the legend of Figure 6.5. The fitting starts from the chain with 2 NFs. Based on these equations, each additional router in the chain adds 1769 μs of (median) latency, while the SCC Profiler is able to account for roughly 864 μs of this latency, falling between the 1st and 15th percentiles of each chain's latency. Looking at the number of memory references per packet, performed by each chain, the equation that describes their dependence on the chain's length is as follows:

$MainMemoryReferences/Pkt = 11647 \cdot NFs + 4689, R^2 = 1.0$

According to Figure 6.3, a hit to main memory takes 71.7 ns, hence multiplying this latency with the coefficient from the equation above, results in 835 μs of latency for each additional router; this is almost exactly the latency increase with the chain length, according to the calculation of the SCC Profiler (as shown in the legend of Figure 6.5). This is not a surprising result; as in § 6.3.1, we showed that even a single user-space router was unable to keep its data in its processor's local cache(s) because the I/O operations involve memory allocation in user-space and data is copied from/to the kernel, touching a lot of memory locations; hence the data cannot stay in the cache causing data exchanges back and forth between main memory and the cache. Clearly this problem only becomes more severe with a chain of these routers.

However, we have not yet clarified, why the latency calculations of the SCC Profiler are below the actual median latencies when we chain NFs. This can be explained by the OS-level counters that the SCC Profiler obtained from each NF. Specifically, by querying the Linux scheduler, the SCC Profiler acquires information about the time a task (*i*) is executing on a CPU, (*ii*) is not runnable, including I/O waiting time, and (*iii*) is runnable but not actually running due to scheduler contention. We derive two metrics from these counters. First, we divide the chain's waiting time by its actual run-time and define the metric "Wait/RunTime". Since the waiting time of each of our NFs is mostly affected by the I/O operations, the "Wait/RunTime" metric captures the impact of yielding the CPU to execute I/O with respect to the effective run-time of an NF. Secondly, we define the metric "SchedContention/RunTime" as a fraction of the time spent due to scheduler contention relative to the chain's run-time. This metric reflects the overhead, added by the OS, to execute the chain.

Table 6.5 depicts the values of these two metrics for four different chain lengths, as obtained from the experiment shown in Figure 6.5. For a single router (i.e., with chain length equal to 1), we observe that the amount of time spent waiting (mostly for I/O) is almost 15x higher than the time spent executing useful instructions on the CPU, while there is no scheduling overhead since the CPU executes, thus the

OS schedules, only this router. Increasing the chain's length leads to increasing waiting and scheduling overheads. The processor spends 75x more time waiting than actually running a chain of 8 routers, while the time that this chain is runnable but does not execute on the processor due to contention in the scheduler is almost 4x higher than the chain's run-time. Our discussion so far has highlighted that both *I/O and scheduling overheads* appear in NFV service chains.

Table 6.5: Effect of the service chain's length on the (*i*) waiting time and (*ii*) time spent due to scheduling contention with respect to the effective run-time of the service chain.

Chain Length	Wait/RunTime	SchedContention/RunTime
1	14.76	0
2	18.80	1.25
4	30.20	2.88
8	74.36	3.78

6.3.2.2 Lessons Learned

The overheads shown in Table 6.5 are captured by the SCC Profiler, but not fully quantified. To clarify this issue, the formula that the SCC Profiler uses to compute the per packet latency (see § 6.2.1.3) includes the entire I/O overhead by following the data movements across the memory hierarchy, but only *partially* captures the scheduler's overhead by computing the per packet latency imposed by context switching (which is only a part of the scheduling overhead). This is because it is hard to accurately project this latter overhead to a per packet latency dimension. Despite this missing scheduling overhead, we believe that our latency calculation methodology provides enough accuracy to describe the per packet latency of NFV service chains. Moreover, *the results in Table 6.5 serve as a motivation for improving the performance of these NFV service chains by addressing these I/O and scheduling overheads.* In § 6.5.2, we quantify the total overhead of the default Linux scheduler by comparing the performance of NFV service chains scheduled by both the default and a more efficient task scheduler.

Next, we address both I/O and scheduling problems by presenting the run-time part of SCC.

6.4 The Service Chain Coordinator

In this section we utilize the knowledge mined by the SCC Profiler to increase the performance of both standalone and chained NFV applications. We designed SCC to integrate both the profiler and various acceleration techniques into the NFV framework illustrated in Figure 6.6. We explain each module of this framework in the following sections.

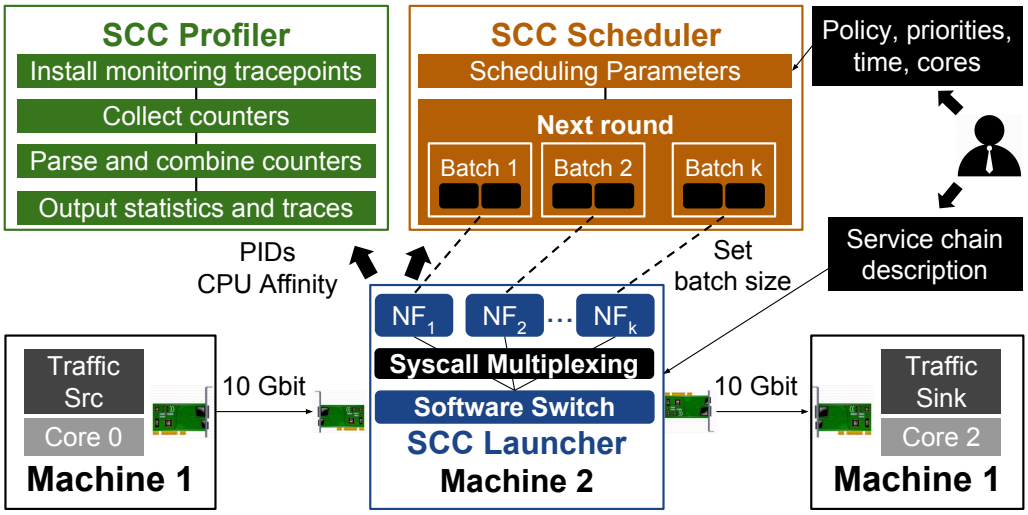


Figure 6.6: The Service Chain Coordinator in the context of our testbed. A system administrator inputs a service chain description and configuration (top right). The SCC Launcher identifies the service components, applies the requested configuration and deploys the chain (bottom center). Using the PID and CPU affinity of each NF, the SCC Profiler (top left) can profile the deployed NFs. The SCC Scheduler (top center) ensures that service components comply with the scheduling configuration specified by the system administrator.

6.4.1 The SCC Launcher

A system administrator specifies a service chain using a simple JavaScript Object Notation (JSON) format understood by SCC. This description is injected into two components of SCC: the Launcher and Scheduler. Specifically, the system administrator chooses those NFs that will comprise the service chain (e.g., a firewall followed by a router), the execution environment that will host each NF (i.e., a native or a container-based NF deployment), the I/O driver (to support a kernel or user-space Linux-based chain), the underlying switching fabric (e.g., OVSK, Linux bridges, etc.), the desired topology of the NFs, as well as the hardware components

that will be used by the chain (i.e., by selecting the CPU affinity of each service component). These NFs can be selected from a pre-installed library. While in our prototype we used FastClick as a packet processing library, our design is not limited to this library.

Next, the configuration parameters for each NF are passed to the SCC Launcher via a JSON-based configuration file. The SCC Launcher parses this input, composes the chain, creates the necessary interfaces (as needed), and launches all of the components (i.e., switches and NFs). The components are pinned to the requested CPU core(s) according to the CPU affinity mask included in the service chain description.

Once the components are launched, the system administrator can choose, via the configuration file, whether the chain will operate in "profile" or "run-time" mode. In "profile" mode the SCC Launcher passes the service chain's PIDs and configuration to the Profiler. As the configuration specifies the CPU affinity of each NF, the SCC Profiler establishes monitoring connections with the appropriate hardware components. Then, the SCC Profiler operates as described in § 6.2.1. In "run-time" mode the PIDs and some auxiliary data structures are passed to the SCC Scheduler. The reason for having these two modes is that the profiling itself occupies system resources, hence we believe that a system administrator would benefit from analyzing her NFV service chains offline, using the SCC Profiler, and then apply the knowledge from the profiling to deploy the accelerated chains online via the SCC run-time.

Before describing the Scheduler, we explain a key internal component of the SCC Launcher, the multiplexing of system calls.

6.4.1.1 Multiplexing of System Calls

The first pillar of SCC's acceleration techniques is an improved I/O mechanism for user-space NFV applications that use native Linux network drivers. This technique is integrated into the SCC Launcher to accelerate interactions of the NFs with the host OS and hardware by multiplexing network I/O-related system calls in order to reduce the number of times the path from user-space to kernel-space and the reverse are used.

In § 6.3.1 and § 6.3.2, we showcased that using per packet send/receive system calls to interact with the OS is costly for NFV tasks, especially when a chain of NFs is executed. The reason is that, instead of an NF utilizing its allocated CPU time for performing the actual packet processing, it yields the CPU to the OS in order that the OS can perform the necessary I/O operations each time a packet has to be received or emitted. Moreover, the time spent processing is a small fraction of the time spent for I/O, based on the experiments of § 6.3.2.

We solve this problem by multiplexing multiple packets into a single system call to allow batches of packets to enter/exit each NF using one receive/send transaction with the kernel. With careful engineering, this method increases the chances of each NF processing an entire batch of packets *uninterrupted*, the next time it gets the CPU. For this to happen, the SCC Launcher operates in three rounds sequentially. In the first round, each NF performs read operations in a batch style. Then, each NF performs its own processing during the second round, while in the last round packets are emitted out of the NFs. In the rest of this chapter, we use the term batching interchangeably with the term multiplexing, both refer to groups of packets being sent/received via one system call.

The system administrator can tune the number of multiplexed system calls via the configuration file shown in Figure 6.6. Based on this configuration, the SCC Launcher will *pre-allocate* a number of I/O vectors that will be used by the kernel to deliver and fetch packets to/from each NF. Ideally, SCC should be able to auto-tune i.e., to select the correct number of I/O vectors itself (based upon changes and feedback from measurements). However, for performance reasons, memory pre-allocation in SCC is static, hence auto-tuning the number of multiplexed system calls, using online feedback from the profiler, would require SCC to restart the NFs. We decided not to automate this process to maintain high performance and prevent service disruptions due to restarting the NFs.

Finally, a challenge when multiplexing I/O-related system calls is to ensure that traffic will not face unacceptable delays when the input rate is low. For example, imagine that one wants to multiplex 16 packets in one, e.g., receive, system call but the input packet rate is e.g., 1 pps. This means that a naive implementation of the multiplexing mechanism might cause the application to block until the entire batch of packets is received (after 16 seconds in this example). To avoid such a problem, the SCC Launcher operates in non-blocking mode, by reading or writing up to a certain number (e.g., 16 packets) of packets at once. If this number is not reached, the system call returns the available packets received/sent or zero if nothing was read/written. This choice allows us to exploit the merits of batching under high input packet rates, while still achieving low latency under low input packet rates.

6.4.2 The SCC Scheduler

In § 6.3.2 we observed increasing scheduling overheads when executing chained NFs. We attributed these overheads to the fact that the default Linux scheduler does not grant large enough time quanta per NF, leading to more frequent scheduling decisions and increased number of context switches. In this section, we allow a system administrator to modify the scheduling procedure of NFV service chains by using our SCC Scheduler.

The SCC Scheduler boots once the service chain has been deployed by the SCC Launcher, the PIDs of the NFs are available, and the chain operates in "run-time" mode. The SCC Scheduler reads the scheduling parameters from the input configuration, registers the PIDs with the appropriate scheduler based on the requested policy, adjusts the priorities*, and re-configures time-related scheduling parameters via system calls. We provide more details regarding the reconfiguration of the scheduler in § 6.4.2.1.

Depending on the input configuration and the selected data plane technology that interconnects the NFs, the SCC Scheduler can operate either in single or multi-core mode as illustrated in Figure 6.7. If the system administrator wants to deploy both the NFs and the underlying switch in the same core, then the SCC Scheduler executes the service chain as shown in Figure 6.7a. This uniprocessor task scheduling scheme invokes the software switch in the odd rounds (i.e., round 1, 3, etc.), and the NFs in the even rounds (i.e., round 2, 4, etc.).

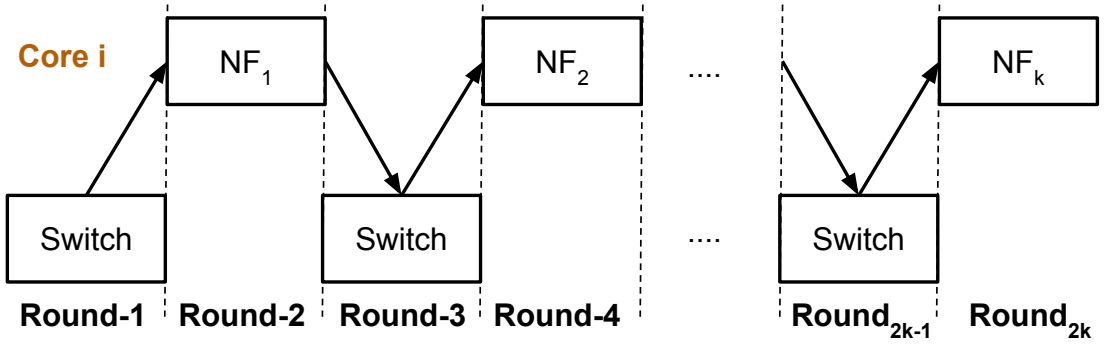
The system administrator might allocate a different core for the switch, to run the NFs in a dedicated core. In this case, the scheme depicted in Figure 6.7b can be used. Modern OSs maintain task queues per core, providing mechanisms to hand off the work from one task scheduler to another, hence better exploiting the hardware capacities. Therefore, to realize the multi-core scheduling plan shown in Figure 6.7b, two instances of the SCC Scheduler are required. One instance schedules the NFs and the other schedules the switch, while in each round the NFs and the switch are running in parallel.

If the system administrator wants to allocate more cores for the chained NFs, the latter scenario (and Figure 6.7b accordingly) can be generalized in a per core basis manner. This involves running one instance of the SCC Scheduler per core and each instance will coordinate its own processes (i.e., NFs/switch).

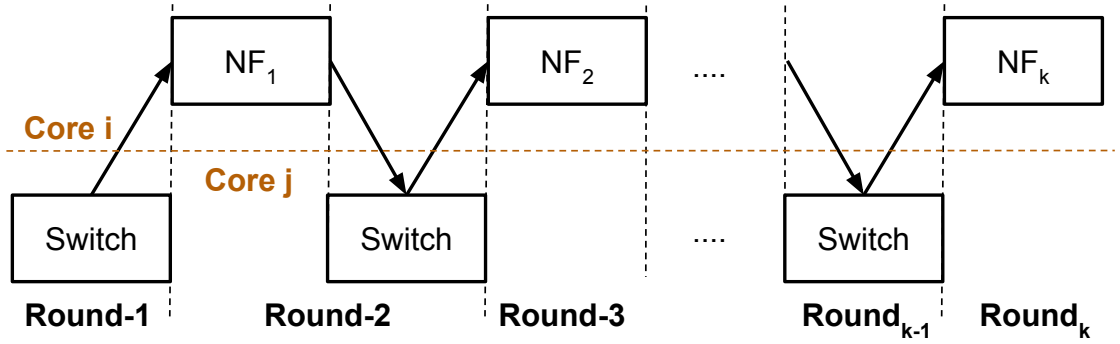
A key task of an NFV scheduler is to guarantee that the scheduling plan (e.g., as shown in Figure 6.7) requested by the system administrator will be executed meticulously. This is challenging because modern OSs employ task migration mechanisms to balance the load among all the available cores, when some of the cores are overloaded. Such a mechanism might cause continuous migrations of the chain's processes from one core to another, destroying the cache coherency.

To guarantee that SCC realizes the scheduling according to the input CPU affinity and scheduling properties, we take two measures. First, the SCC Scheduler reserves the cores requested by the system administrator (see the input of the SCC Scheduler in Figure 6.6), by excluding those cores from the candidate list of cores that undertake other processes in the system, hence ensuring dedicated resources for NFV processing. Second, as explained in § 6.4.1, the SCC Launcher explicitly pins the service chain's processes to the reserved cores based upon the same CPU

*If no priorities are given, default kernel values are used (see § 6.4.2.1).



(a) Uniprocessor scheduling, where the entire service chain runs in a single core.



(b) Example of a multiprocessor scheduling where all the NFs run in one core and the software switch runs in a different core.

Figure 6.7: Scheduling options for service chains (NFs and the underlying switch).

affinity input. These measures ensure that the reserved cores execute only NFV tasks and the NFV scheduling is solely orchestrated by the SCC Scheduler.

6.4.2.1 Tuning the Linux Schedulers

We studied the task scheduler of the Linux kernel v3.13 to identify knobs that will allow a developer to reconfigure key parameters for NFV service chains, such as the scheduling policy, the priority range of a given scheduling policy, and the time quantum granted to a task by the scheduler. Table 6.6 summarizes the important properties of these Linux schedulers.

This version of the Linux scheduler maintains 140 queues, each corresponding to a different priority level. Priority levels between 1 and 99 (1 is the highest priority) are static and can be used by processes scheduled by the real-time scheduler. All of the remaining 40 priority levels (i.e., [100, 139]) correspond to a single static priority 0, which is lower than any real-time priority; however, these tasks are mapped to a dynamic priority range in [-19, 19] (with -19 being the maximum dynamic priority) as shown in Table 6.6.

Table 6.6: Scheduling settings useful for NFV tasks in the Linux OS v3.13.

Scheduling Policy		Priority Range		Time Allocation
		Static	Dynamic	
CFS	**Default**	0	[-19, 19]	Dynamically selected based on (*i*) # of running tasks (*ii*) dynamic priority (see equation 6.1)
	Batch	0	[-19, 19]	
Real Time	**RR**	[1, 99]	-	Reconfigurable via: sched_rr_timeslice_ms
	FIFO	[1, 99]	-	Time-less scheduler

CFS is the default Linux scheduler that schedules tasks with static priority 0. As shown in Figure 6.8, the core data structure that strikes the balance of all tasks' virtual run-times in CFS is a time-ordered tree, where each node corresponds to a task and is associated with the task's virtual run-time. A task with a low virtual run-time value is stored towards the left side of the tree and has the gravest need for the CPU. Conversely, tasks with a high virtual run-time value (or less need for the CPU) are stored towards the right side of the tree. Therefore, the leftmost node

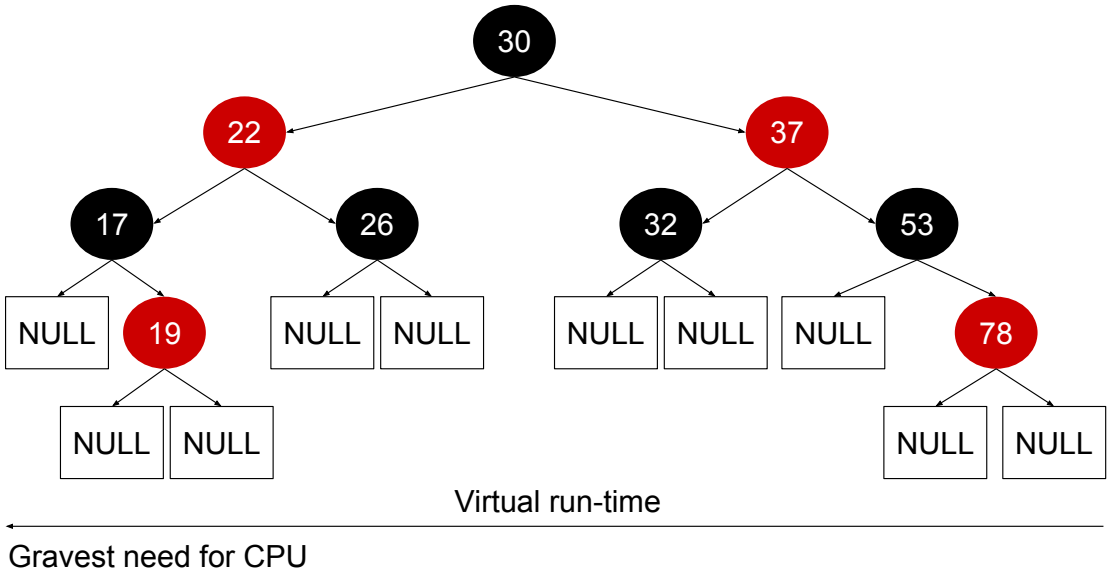


Figure 6.8: The Linux Completely Fair Scheduler's red-black tree data structure for selecting the next runnable task.

of this tree is the next task to execute on the CPU. CFS exposes system calls to modify a task's dynamic priority. CFS guarantees that the minimum time quantum granted to a task will be always greater or equal than *sched_min_granularity* and computes this value based on the following formula:

$$CFSTimeQuantum = \begin{cases} (140 - P) \cdot 20, & \text{if } P < 120 \\ (140 - P) \cdot 5, & \text{if } P \geq 120 \end{cases} \tag{6.1}$$

where $P \in [100, 139]$ is the task's dynamic priority mapped to a value in [-19, 19] (see Table 6.6). Based on this formula, the largest time quantum that can be granted to a process scheduled by CFS is 800 ms.

Note that, the time that CFS will finally allot to a task also depends upon run-time state variables in the kernel. For example, preemption might be triggered if a more deserving task is available, hence a task's slice might not be entirely consumed. To maintain longer execution times, CFS offers another scheduling policy for CPU-bound processes, called batch CFS. This policy prevents other processes from preempting the CPU as would occur under the default CFS policy, hence the processes run for longer time slices. A process scheduled with the batch scheme "lives" in the same data structure as the processes scheduled by the default CFS scheme, uses the same priority ranges, and the next process to execute is still chosen by CFS. These properties of the batch scheduling scheme are beneficial for NFV tasks as shown in § 6.5.2.

The Real-Time Scheduler provides two scheduling policies for interactive tasks: FIFO and RR. Tasks scheduled using either of these two policies will always be prioritized over any tasks scheduled by CFS. The scheduler maintains a list of runnable threads for each possible static priority value. In order to determine which thread runs next, the scheduler looks for the nonempty list with the highest static priority and then selects the task at the head of this list.

To illustrate the scheduler's functionality, consider the example shown in Figure 6.9. Let us assume that the system has two available CPU cores and there are 6 tasks active at the time. Tasks T1, T2, T3 run on CPU 0, while tasks T4, T5, T6 run on CPU 1. Each core has a static array of tasks and the size of the array is equal to the number of static priorities available for this scheduler (i.e., 99). This means that if T1 has a static priority 60, while T2 has a static priority 93; then, if CPU 0 currently executes T3 (as shown by the index in Figure 6.9), the next task to be selected by the scheduler is T2 since its static priority is higher than T1's static priority (i.e., $93 > 60$). Similarly, CPU 1 will prioritize T5 after the execution of T4, since static priority 32 is greater than static priority 2.

A process scheduled by the FIFO scheduling algorithm has no time slice, but instead runs until it blocks (e.g., for I/O), is preempted by a higher-priority real-

time task, or voluntarily yields the processor. Two or more FIFO tasks with equal priority do not preempt each other and tasks of lower priority will not be scheduled until the process relinquishes the CPU.

In contrast, the RR real-time scheduling policy is a timeful extension of the FIFO scheme. Unlike FIFO, each task scheduled with the RR algorithm is allowed to run only for a certain maximum time quantum. Upon the expiration of this time quantum, the task will be put at the tail of the queue for its priority. As depicted in Table 6.6, this scheme exposes a way to adjust the duration of the value of the time quantum via the proc filesystem, hence RR is an alternative scheduling policy for SCC. In contrast, FIFO's time-less approach could be beneficial for executing single-process NFV tasks, but provides limited control of the process execution time in multi-process NFV scenarios.

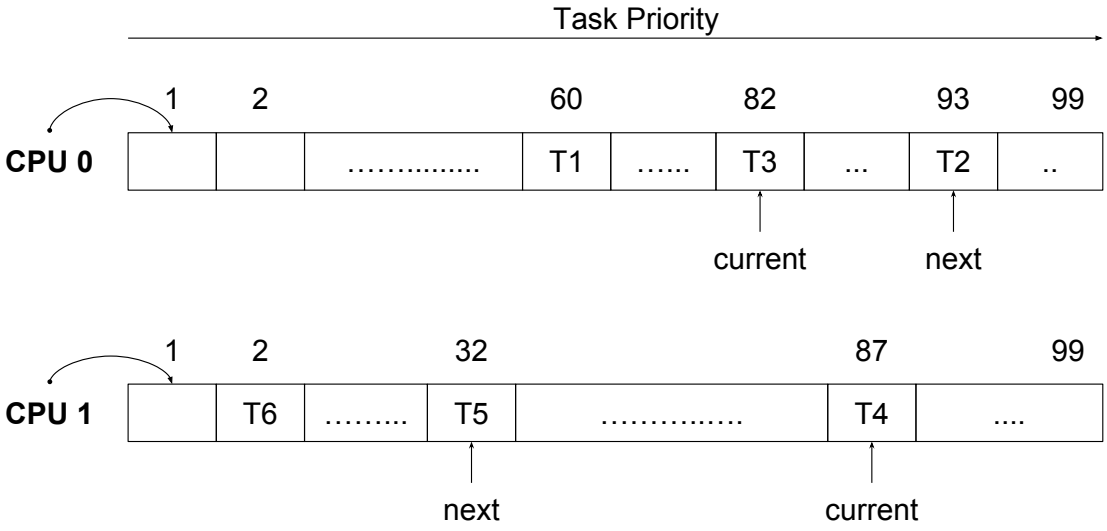


Figure 6.9: An example scenario of per processor real-time tasks queued on run queues. The current and next tasks to run are indicated by indices. The queues are static arrays and the indices denote the priority of each task.

6.4.3 The Entire SCC System

Separately employing the I/O and scheduling techniques above might not lead to the desired performance. For example, granting a short time quantum to a process that reads a large batch of packets might not fully reap the benefits of batching. In contrast, allocating a long time quantum for a process that applies per packet read/write operations cannot be fully exploited since, sooner or later, the process will yield the processor to perform I/O, thus "losing" the opportunity to exploit the long time quantum.

For these reasons, as shown in Figure 6.6, SCC builds a run-time that effectively combines these accelerations. As explained above, the system administrator can tune the number of multiplexed system calls, scheduling policy, scheduling priority and the time quantum of each process to achieve fast packet processing. As illustrated in Figure 6.1, correct selection of these parameters will allow a CPU to process an entire batch of packets per scheduling round, leading to fewer user to/from kernel-space paths being used, hence lower latency. We evaluate the effectiveness of SCC in § 6.5.

6.5 Performance Evaluation

This section evaluates the acceleration techniques of SCC. We used the standalone and chained NFV services, profiled and analyzed in § 6.3.1 and § 6.3.2 respectively, to assess the benefits of SCC. We deploy these NFV service chains on top of SCC (see § 6.4) and answer three key questions: (*i*) What is the effect of SCC's I/O multiplexing on the performance of individual user-space NFs (see § 6.5.1)? (*ii*) What is the impact of different scheduling strategies on the performance of chained user-space NFs (see § 6.5.2)? (*iii*) What are the benefits of SCC when both I/O multiplexing and scheduling are applied (see § 6.5.2.2)?

6.5.1 Impact of SCC's I/O Multiplexing

The goal of this section is to evaluate our first acceleration technique for user-space NFV chains: multiplexing multiple packets into one system call. We use the user-space FastClick router (based on the native network driver) from § 6.3.1 and deploy it on SCC. Then, we assess the impact of I/O multiplexing (as a function of the batch size) on the router's performance, by conducting a sensitivity analysis using an exponentially increasing batch size based on the formula: $batch\_size = 2^i|_{i=0}^{8}$. When the batch size equals 1, no batching is used, i.e., simply the standard FastClickI/O. We take this as the base of what we want to accelerate.

Figure 6.10 depicts the latency of a single router as a function of the batch size with four different frame sizes (i.e., 64, 128, 256, and 1500 bytes). We highlighted two areas in this figure: the left-most area, with a light red background, where batching is disabled, whereas the remaining area, with a light green background, shows the router's performance for different batch sizes. We input frames with different frame sizes at the same rate (i.e., 0.82 Mpps, which is the line-rate for the 1500 byte frame size) used in all of our experiments. As we see in Figure 6.10, the load imposed on the router is the same for all the frame sizes, since the router exhibits similar latencies, independent of the frame size. For this reason, we used the SCC Profiler to analyze the memory utilization of the router during one of these

experiments, with the smallest frame size (i.e., 64 bytes), as shown in Table 6.7. This frame size was also utilized to profile the same router (see § 6.3.1) without batching, hence it offers a clear comparison reference for our I/O acceleration.

Looking at the results of Figure 6.10, we see that our batching acceleration clearly outperforms the non-batching case, achieving 2-3x lower median latency for several batch sizes. Specifically, the best batch sizes, with respect to the end-to-end latency, are between 2 and 32 batched system calls, as the median latency for almost all frame sizes is in the range of 80-115 μs, whereas the non-batching cases achieve median latencies in the range of 270-310 μs. The lowest value of the latter group of medians is also visualized with the red horizontal dashed line shown in Figure 6.10. This difference in latency is reflected in a decrease in the number of references to memory with batch sizes greater than 1, as shown in Table 6.7. Batching decreases the number of main memory references by 2-3x. As main memory accesses are the main component of the latency, we can see the effect of batching is very beneficial in reducing latency (up to some point). More notably, the worst case latency (here the 99$^{\text{th}}$ percentile) for some batch sizes *is comparable or even lower (i.e., for 1500-byte frames) than the median latency of the router without batching.*

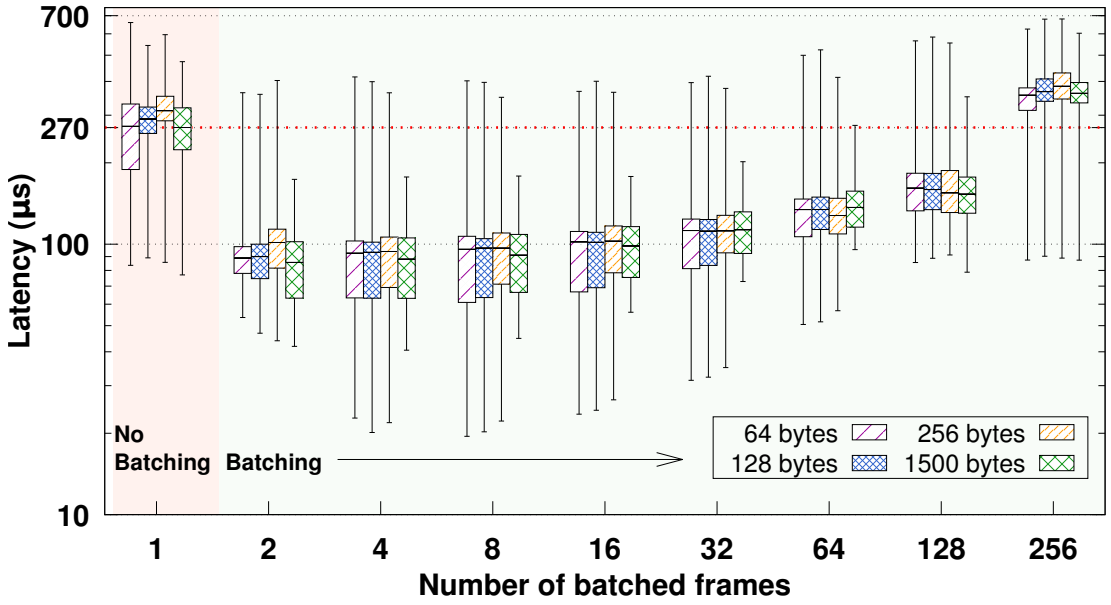


Figure 6.10: End-to-end latency (μs), plotted on a logarithmic scale, versus the number of frames multiplexed/batched into one system call for a user-space FastClick router using the native Linux network driver. The router runs in a single core and the input packet rate is 0.82 Mpps with 64, 128, 256, and 1500 byte frames. The corresponding bit rates are 0.57, 0.99, 1.84, and 10 Gbps.

Another benefit of batching is the reduction of the latency variance (also known as jitter). Without batching, the router exhibits a latency variance of 400-600 μs for the different frame sizes. With a batch size of e.g., 2 system calls, this variance is roughly 300 μs (i.e., 33-200% lower) for the 64, 128, and 256 byte frames and only ~130-140 μs (i.e., 2.5-4x lower) for 1500 byte frames. We believe that jitter-sensitive NFV applications will find this batching beneficial.

Batch sizes between 4 and 32 have some outliers at very low latency, comparable to the levels of a DPDK router (see Figure 6.4). Moreover all the latency percentiles for these batch sizes are shifted by roughly 3x compared to the non-batching case. Further increasing the batch size (i.e., batch sizes of 64 and 128 frames) achieves yet lower median latency than the non-batching case. However, using a batch size of 256 frames increases the latency and latency variance as both metrics are greater than the non-batching case. This is not surprising, since aggressive batching has a well-studied effect on latency and latency variance [48].

From a resource utilization perspective, Table 6.7 shows the router's per packet latency and different types of memory accesses, as computed by the SCC Profiler, for the different batch sizes when the frame size is 64 bytes. As stated earlier, a clear impact of multiplexing multiple frames into one system call is a reduction in main memory accesses. In the non-batching case, almost 3000 main memory references per 64 byte frame occur, resulting in a latency of 216.97 μs, as calculated

Table 6.7: The SCC Profiler's per packet latency calculation and memory utilization report while tracking the *64-byte* packets injected during the experiment shown in Figure 6.10. The latency in the second column is calculated using the collected performance counters introduced in § 6.2.1.3 and falls within the actual latency percentiles shown in Figure 6.10.

Batch Size	Per Packet	
	Latency (μs)	L1/L2/L3/DRAM References
1	216.97	3653/179/65/2964
2	126.94	2048/138/49/1731
4	117.47	1872/125/46/1603
8	101.05	1601/115/42/1378
16	95.23	1503/119/41/1298
32	84.20	1315/110/38/1148
64	85.88	1331/113/40/1170
128	94.31	1471/145/46/1284
256	155.0	2564/252/81/2109

by the SCC Profiler. Exponentially increasing the batch size from 2 to 256, leads the OS to transferring more data per batch, i.e., an entire batch of frames is transferred with one system call, and this transfer occurs less frequently (because we apply one system call every "batch size" number of frames). Consequently, batching exploits the spatial locality of virtual memory addresses; this means that multiple virtual addresses tend to fall into the same physical page, hence there are fewer references to main memory. This effect is shown in the third column of Table 6.7. For batch sizes between 2 and 64, the router makes 2-2.8x fewer main memory references than the router without batching, hence the latencies shown in Figure 6.10 are greatly affected by this phenomenon.

This analysis leads to three conclusions: (*i*) To benefit from I/O multiplexing, moderate batch sizes between 2 and 32 system calls decrease the end-to-end latency and latency variance for small, medium, and large frames; hence this degree of multiplexing appears attractive. (*ii*) When the goal is to fit more service chains into a given hardware capacity, choosing bigger batch sizes (i.e., between 8 and 64 system calls) leads to more than 2x better cache utilization (especially by reducing the number of main memory accesses). (*iii*) For jitter sensitive applications, batching 2 system calls gives the best results both from the latency and jitter perspectives.

6.5.2 Impact of SCC's Scheduling

In this section we evaluate the effects of different scheduling strategies on the performance of NFV service chains. In § 6.3.2.1, we observed increasing scheduling overheads with the length of the chain and attributed these overheads to the inability of CFS to grant large enough time quanta per NF. Here, we deploy NFV chains using SCC and utilize the SCC Scheduler to, ideally, eliminate this overhead.

Considering the analysis of the different schedulers in § 6.4.2, we see that the batch CFS and the real-time RR schedulers offer interesting properties that could be beneficial for NFV service chains. Here, we evaluate the former scheduler against the default Linux scheduler. To modify the time quantum of each NF in a chain we set its "niceness" value accordingly. Based on equation 6.1, without modifying the "nice" value, a process can run for up to 100 ms; we have increased this value to study its effect on the performance of the NFV chains.

In § 6.5.2.1 we evaluate a multi-core scheduling scenario without using our I/O multiplexing, while in § 6.5.2.2 we combine scheduling with I/O multiplexing in a single-core scenario.

6.5.2.1 Multi-core Scheduling without I/O Multiplexing

Figure 6.11 shows the latency of 1-8 user-space FastClick routers, chained together, on top of an OVSK instance. The chains are scheduled using the multi-processor scheduling option of SCC shown in Figure 6.7b, where one CPU core executes the OVSK, while another CPU core in the same socket executes the NFs. Specifically, the top set of chains in the legend is scheduled by the default CFS (with the default time quantum up to 100 ms), while the other two sets of chains are scheduled by the batch CFS with two different time quanta configurations. The former configuration's time quantum is roughly 4x greater than the default (i.e., 420 ms), while the latter is 8x greater than the default (i.e., 800 ms). To achieve this configuration we set the "nice" value to -1 and -19 respectively (or 119 and 100 based on equation 6.1). Although CFS does not guarantee to exhaust its assigned slice, it acts as an upper bound.

In this experiment, SCC uses only scheduling acceleration, i.e., without I/O multiplexing. This is because while we implemented the I/O multiplexing in FastClick, OVSK still relies on its standard I/O mechanism, without using

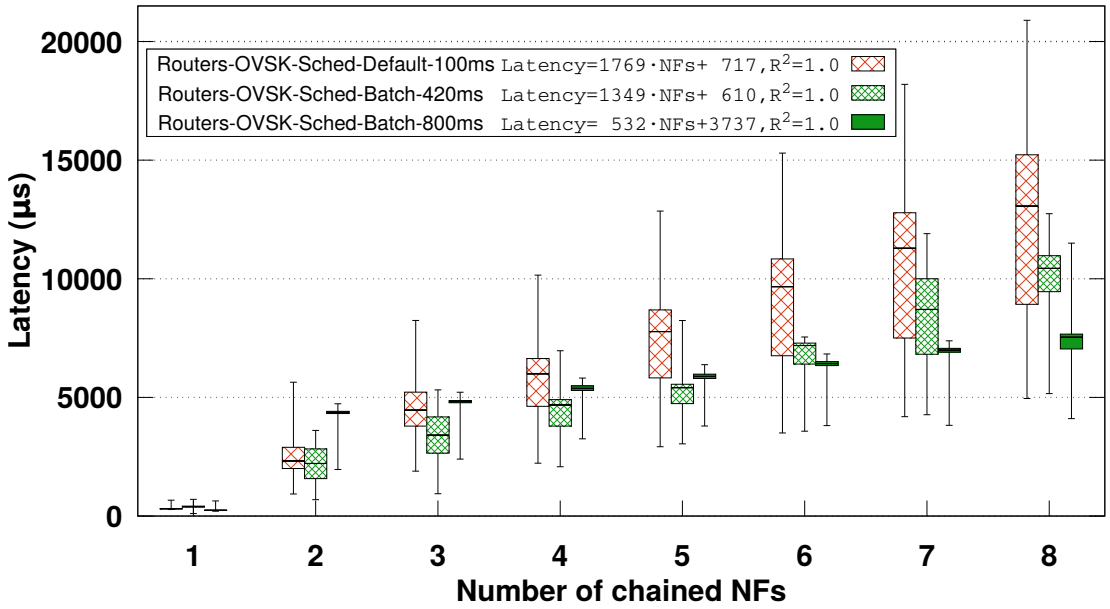


Figure 6.11: End-to-end latency (μs) versus the chain's length for FastClick routers, running in containers on top of OVSK. The chains are scheduled either with the default or batch CFS policies, the latter with different time quanta allocations. The routers run in a single core, OVSK runs in a different core in the same socket, and the input rate is 0.82 Mpps with 64 byte frames. The fit to the median latencies, stated in the legend, begins from the chains with 2 NFs.

batching. Using I/O multiplexing only in the NFs is counter-productive, since packets have to be batched and un-batched multiple times while moving from OVSK to NFs along the chain, hence no multiplexing is used in this test.

Based on Figure 6.11, we can see that the SCC Scheduler realizes the chains with a considerably lower latency compared to the default scheduler. Starting from the chains with 2 NFs, we fit a linear equation to the median latencies for each scheduling scheme to determine the cost of additional NFs in each chain. This cost is 1769 μs per additional NF, when we use the default Linux scheduler (as was previously reported in § 6.3.2). Using the batch scheduler, the latency of the same chains is 1349 or 532 μs (30-300% lower) depending upon the size of the time quantum. In the case of the batch CFS with a time quantum of 420 ms, the latency is *always* lower than the default CFS and the scheduling benefits continue to increase with the chain's length. In contrast, using the maximum time quantum (i.e., 800 ms) appears to be an overkill for short chains, as the latency is actually higher than the default CFS. However, if an NFV provider wants to deploy chains with more than 5 NFs, this larger time quantum achieves substantially lower latency, below that of the other two cases.

SCC greatly reduces latency variance. Looking at Figure 6.11, the edges of the latency boxplots (i.e., 25$^{\text{th}}$ to 75$^{\text{th}}$ percentiles) for the batch scheduling cases fall close to each other, hence these chains deliver the majority of packets with low variance. Especially when using batch CFS with the maximum time quantum, the 25$^{\text{th}}$ to 75$^{\text{th}}$ percentiles almost match. In contrast, when using the default scheduler, the chains exhibit a huge latency variance, that is orders of magnitude greater than the batch scheduling cases for long chains. For example, the variance between the 25$^{\text{th}}$ and the 75$^{\text{th}}$ latency percentiles, for a chain with 7 routers, when using the default scheduling scheme is 6327 μs. The same percentiles for the same chain, when scheduled by the batch CFS, differ by 3180 μs when using the 420 ms time quantum, and only 156 μs (40x less variance than the default CFS) using the maximum time quantum.

6.5.2.2 Single-core Scheduling Combined with I/O Multiplexing

Next, we combine the benefits of the above scheduling scheme with our I/O acceleration (see § 6.4.1.1) to exploit the full capacity of SCC. Hence, we deployed the same chains of routers on SCC, interconnected B2B, without an underlying software switch. This configuration avoids the slow I/O of OVSK and can operate using a batched I/O mode. The chains are scheduled using the uni-processor scheduling option of SCC shown in Figure 6.7a, where one CPU core coordinates the execution of all the routers.

Figure 6.12 shows different variants of these chains. The top set of chains in the legend are scheduled by the default CFS, while the other four are scheduled

by the batch CFS. In this experiment we use the maximum time quantum for the batch CFS, as we found that it does not exhibit the negative impact observed in § 6.5.2.1 for short chains. This is because, when combined with SCC's I/O multiplexing, this scheduling allows the NFs to better exploit this large time slice. To quantify the effects of both the batch CFS and the I/O multiplexing, we tested four different cases. From top to bottom in the legend, the second set of chains use the batch CFS without I/O multiplexing, while the last three sets of chains use I/O multiplexing with batch sizes 2, 16, and 32 respectively.

Looking first at the latencies of the chains scheduled by the default and batch CFSs' (the latter without I/O multiplexing), we notice a similar trend to that shown in Figure 6.11. This means that batch CFS is beneficial regardless of the interconnect of the chains. The benefits of the batch CFS can be quantified by looking at the equations fitted to these two cases. Although the per NF latency cost (i.e., the slope in the equations) of the batch CFS is only ~10% lower (1268 μs versus 1335 μs), one should pay attention to the intercepts of these functions (75

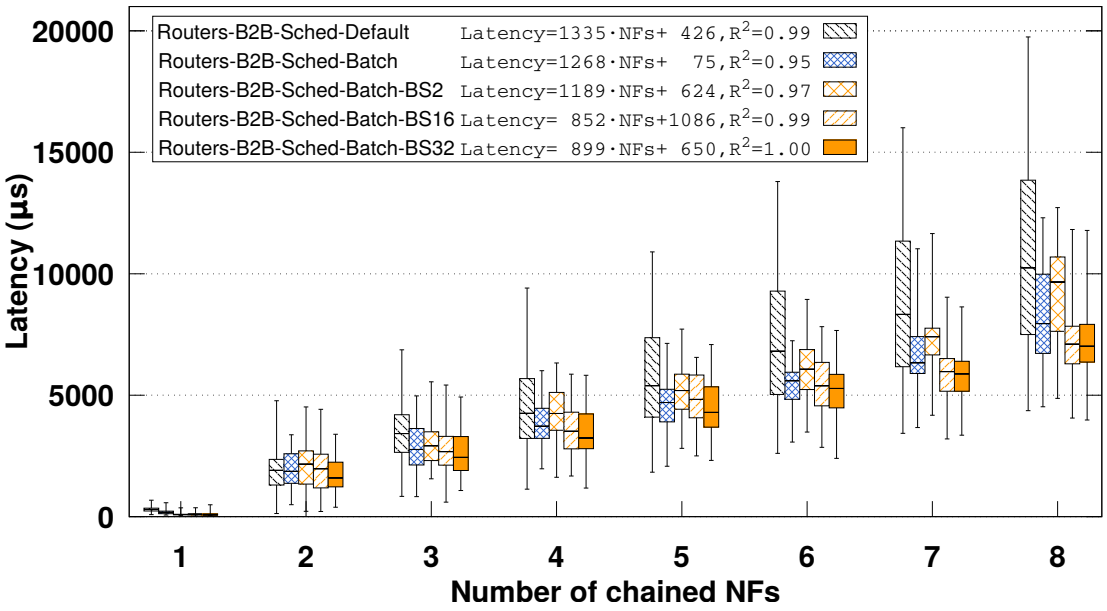


Figure 6.12: End-to-end latency (μs) versus the chain's length for FastClick routers, running in B2B chained containers. The top set of chains in the legend are scheduled by the default CFS. The other four chains are scheduled by the batch CFS; the first of them does not use I/O multiplexing, while the remaining use I/O multiplexing with batch sizes 2, 16, and 32 (from top to bottom in the legend). Note that the maximum time quantum is granted to the NFs by the batch CFS in this experiment. The routers run in a single core and the input rate is 0.82 Mpps with 64 byte frames. The linear fit to the median latencies, stated in the legend, begins from the chains with 2 NFs.

versus 426 μs for a zero length chain). These values indicate the basic processing cost of each chain, which in the case of the batch CFS is almost 7x lower than the default CFS. The batch CFS also reduces the latency variance by up to 2x.

Section 6.5.1 showed the benefits of multiplexing multiple packets into one system call for a standalone NF. Now, we quantify the benefit of this technique when combined with the increased time quanta per NF allocated by the batch CFS in a scenario with chained NFs. The remaining three sets of boxplots in Figure 6.12 show the latencies for these three batch sizes. With a batch size of 2 system calls, the best results are achieved in the standalone case (see § 6.5.1), but this does not have similar performance in a chained scenario. We attribute this fact to the mismatch between the execution time granted by the batch CFS and the frequency of system calls made by the NFs in this case. In other words, batching only two packets at a time for a long chain of NFs requires frequent system calls, which leads to yielding the CPU long before a NF's time quantum expires.

We observe that when using larger batches, the NFs seem to better exploit their time slices. A close look at the fitted equations proves this fact. A batch size of 2 system calls incurs 1189 μs of latency per additional NF, but with a much greater basic processing cost than the batch CFS without I/O multiplexing, rendering the I/O multiplexing technique as a worse option. However, batch sizes of 16 and 32 packets reduce the per NF latency by up to 50%, while also greatly reducing latency variance. For a chain of 8 routers with the batch CFS using a batch size of 32 system calls the latency reduction is 4x greater than the default CFS. Indeed, comparing the variance between the 25$^{\text{th}}$ and the 75$^{\text{th}}$ latency percentiles in Figure 6.12, half of this reduction is due to the scheduler, while the other half stems from I/O multiplexing.

Finally, we correlate the above latency measurements with the SCC Profiler's data, gathered during the execution of both the OVSK and B2B interconnected chains. Table 6.8 shows the values of the "SchedContention/RunTime" metric, defined in § 6.3.2.1. This metric captures the time spent due to scheduler contention relative to the chain's run-time. For a single router (i.e., chain length equal to 1), there is no corresponding cost because this router is the only process to be scheduled. However, under the default CFS policy, for a chain of 4-8 NFs the time that the chain is runnable but does not execute due to contention in the scheduler is 3-4x greater than the actual run-time of the chain for both OVSK and B2B cases. The batch CFS scheduler employed by SCC reduces this overhead by ~50%. However, for a chain of 2 routers we observe an increased scheduling overhead compared to the default CFS, especially for the OVSK case. That is confirmed by the increased latency of this particular chain, as depicted in Figure 6.11. As explained above, the presence of OVSK does not allow an NF to fully exploit its longer execution time, because of OVSK's ineffective I/O. This

is not the case for the B2B chain of 2 routers, since our I/O multiplexing better exploits the time available from the CPU.

As for the metric "Wait/RunTime", also defined in § 6.3.2.1, this is mostly affected by the I/O mechanism of the chain. We measured a ~40-60% reduction of this overhead when we used SCC's I/O multiplexing, compared to a chain without this acceleration.

Table 6.8: Effect of the batch CFS scheduler on the time spent due to scheduling contention with respect to the effective run-time of the service chain for four chain lengths. The last case of B2B chained NFs, labeled as "Batch+MUX", also uses I/O multiplexing of 32 packets into one system call.

Chain Length	SchedContention/RunTime				
	OVSK Chains		B2B Chains		
	Default	Batch	Default	Batch	Batch+MUX
1	0	0	0	0	0
2	1.25	4.34	1.39	2.83	2.82
4	2.88	2.73	3.11	2.92	2.91
8	3.78	2.58	4.17	2.82	2.98

6.6 Originality and Open Source Contributions

Here we highlight the originality of SCC with respect to earlier efforts, related to network I/O (see § 3.3), scheduling (see § 3.6) and system profiling (see § 3.7).

6.6.1 System Benchmarks

lmbench [72] and Intel's memory latency checker [73] are tools for measuring a system's performance by benchmarking the hardware's performance capabilities. To do so, they measure the latency for intra and inter-memory transactions. For example, lmbench and memory latency checker can precisely quantify the latency when transferring data of variable sizes between two caches or between a cache and main memory.

SCC uses lmbench as a system benchmark. Apart from memory latencies, SCC requires kernel-level benchmarks to measure scheduling and system calls overhead. Although there are relevant available tools such as [103], we developed our own benchmarks to acquire these metrics. These benchmarks are available at [98].

6.6.2 Code Profilers

OProfile [74] and Perf [75] are code profilers that provide statistics about applications or the entire OS, by accessing low-level performance counters. Such tools can draw a developer's attention to those functions that exhibit high utilization of system resources, hence these functions offer the greatest potential for performance improvements. A great deal of effort is required to understand how the applications under test use the system's resources. Moreover, this knowledge is needed, to instruct these code profilers which particular subset of relevant events, out of a large pool of potential events, are actually relevant.

We performed a study to find crucial NFV performance counters and incorporated Perf in SCC, as we found that it can access these counters. These counters are illustrated in Figure 6.2 and quantified in Table 6.3.

6.6.3 Data Profilers

Various cache profiling tools have been proposed, such as CProf [76], callgrind's KCachegrind tool [77] (based on valgrind), and Intel's PMU [78]. These tools can track applications' cache utilization allowing a developer to build a map of the system's caches and how they are used. Moreover, likwid [79] provides a broader and modular performance monitoring suite. One can either wrap an entire application to measure its performance with respect to key hardware counters, or enclose a particular piece of code within an application between likwid start and stop functions.

DProf [80] helps programmers understand cache miss costs by associating these misses with the data types instead of the code. DProf provides clear insights into which objects of an application's data structures incur expensive cache loads; however, DProf mostly focuses on the LLCs and in particular how data moves in and out of LLCs. This focus results from the tool being designed to optimize cross-CPU data exchanges.

The SCC Profiler extends these earlier profilers by keeping track of the entire memory hierarchy (including TLBs and main memory), hence our profiler quantifies both data movements and address translations from a processor all the way to the main memory and correlates this data with OS-level counters.

6.6.4 Scheduling

Sivaraman et al. [69] envisioned future switches with programmable boards that allow network administrators to deploy custom packet scheduling schemes. They introduced a Push-In First-Out abstraction model that controls the order and departure of packets, capturing the needs of several packet scheduling schemes. Mittal et al. [70] explored the possibility of designing a universal packet scheduler that can match the results of any scheduling algorithm, concluding that the Least Slack Time First algorithm best approximates a universal scheduler.

Scheduling a multitude of processes that comprise a service chain is not addressed by prior scheduling works since these solutions operate at the packet and not at the task level. In contrast, we study the performance of task scheduling in NFV to identify ways to achieve better resource utilization. In § 6.4.2.1, we studied all the available schedulers of our OS and provided useful insights on their properties with respect to NFV. In § 6.5.2, we evaluated the benefits of the batch CFS scheduler and compared it to the default Linux scheduler. Although the real-time RR scheduler has attractive properties for NFV tasks (see § 6.4.2), our experiments showed that this scheduler outperforms the CFS schemes only in the case of standalone NFs. The focus of this work is on chained NFV scenarios, hence we chose the batch CFS.

6.6.5 Network I/O

Netmap [25], DPDK [24], PFQ [26], and PF_RING [104] are network I/O mechanisms that boost NFV performance by providing direct access to the ring buffers of a NIC, using custom network drivers. The time required for these tools to be widely adopted in the market motivated a solution that could be immediately adopted by cloud providers. The Linux kernel has significantly evolved over the past decade and today provides sufficient tools to speed up NFV applications running on top of *unmodified* network drivers.

We found that the vectorized I/O technique [41] introduced in version 2.5 of the Linux kernel permits reading/writing frames from/to multiple buffers using a single transaction. This technique is supported by the ixgbe driver in Linux and can be exploited by activating the scatter/gather feature of the NIC. Our open source implementation [97] is built on top of FastClick. Earlier efforts have successfully applied similar techniques [25, 48, 24] to amortize the system calls' overhead. Our work advances these works by combining batch I/O with scheduling to further improve the performance of NFV applications.

Chapter 7

Synthesizing High Performance NFV Service Chains

As discussed in § 3.5, several consolidation attempts were made to improve the performance of chained NFs. Contemporaneously with the attempt called OpenBox [68], we implemented the mechanisms specified in [105] as the next logical step in our high-performance NFV research. A detailed comparison with OpenBox is given in § 7.7.

This chapter*, describes the design and implementation of Synthesized Network Functions (SNF), our approach for dramatically increasing the performance of NFV service chains. The idea behind SNF is simple: create spatial correlation in order to execute service chains at the speed of the CPU cores operating on the fastest, i.e., L1, cache of modern multi-core machines. SNF leverages the ever-continuing increases in numbers of cores of modern multi-core processor architectures and the recent advances in user-space networking.

Packets in a traffic class are all processed the same way. SNF automatically derives traffic classes of packets that traverse a provider-specified service chain of NFs. 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 practice, SNF allocates multiple CPU cores to execute different sets of traffic classes in isolation (see § 7.1).

SNF's consolidation 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

*The work described in this chapter is based on the journal article "SNF: Synthesizing high performance NFV service chains" [101] (the authors of the article retained the copyright and give their joint approval for parts of this material to appear in this thesis).

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 from a packet processing chain to packet classification, as one component of SNF has to classify each incoming packet into one of the pre-determined traffic classes, and pass it to the synthesized function. Existing open-source software was extended to improve the performance of software-only packet classification. In addition, in one set of experiments an OpenFlow [14] switch was employed as a packet classifier to demonstrate the performance that would be possible with a sufficiently powerful programmable NIC. The benefits of SNF for network operators are multi-fold: (*i*) SNF dramatically increases the throughput of long NF chains, while achieving low latency, and (*ii*) it preserves the functionality of the original service chains.

The SNF design principles were implemented into a modified version of the Click [33] framework. To demonstrate SNF's performance, a comparison between SNF and FastClick is being made (see § 7.5). To show SNF's generality we tested its performance in three use cases: (*i*) a chain of software routers, (*ii*) nested NAPTs [23], and (*iii*) Access Control Lists (ACLs) using actual NF configurations taken from ISPs [106].

The evaluation in § 7.5 shows that software-based SNF achieves 40 Gbps, even with small Ethernet frames, across up to 10 NFs, even with stateful chains. SNF chains show 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 frame sizes), while bounding the median chain latency to below 100 μs.

An SNF overview is provided in § 7.1. The synthesis approach is introduced in § 7.2 and a motivating example is presented in § 7.3. Implementation details and performance evaluation are presented in § 7.4 and § 7.5 respectively. Verification aspects are discussed in § 7.6. Finally, § 7.7 shows the originality of SNF with respect to the state of the art.

7.1 SNF Overview

The idea of synthesizing network service components consorts with a powerful property: *data correlation in network traffic*. In a network system, this property is mapped to *spatial locality with respect to the receiver's caches*. SNF aggregates parts of the flow space into Traffic Class Units (TCUs) (a detailed definition is given in § 7.2.1). These TCUs are mapped into sets of (re)write operations. Then, by carefully setting the CPU affinity of each TCU, this aggregation enforces a high degree of correlation in the traffic (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, SNF is designed based on Figure 7.1. In the example shown in this figure we assume that a network operator wants to deploy a service chain between network domains 1 and 2. For simplicity we 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) attempts 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 typically has substantial capacity in modern hardware architectures.

Once a processing core acquires a frame, it executes SNF as shown in Figure 7.1. First the core classifies the frame (green rectangles in Figure 7.1) in one of the chain's TCUs and then applies the required synthesized modifications (blue rounded-rectangle in Figure 7.1) that correspond to this TCU. Both classification and modification processes are highly parallelized as different cores can simultaneously process frames that belong to different TCUs. Both processes are detailed in § 7.2.2.

The key point of Figure 7.1 is that a core's pipeline shares nothing with any other pipeline. We employed the symmetric Receive-Side Scaling (RSS) [107] scheme by [108] to hash input traffic such that bi-directional flows are always served by the same SNF rewriter, hence the same processor. This scheme allows a core to process traffic for a TCU at the maximum processing speed of the machine.

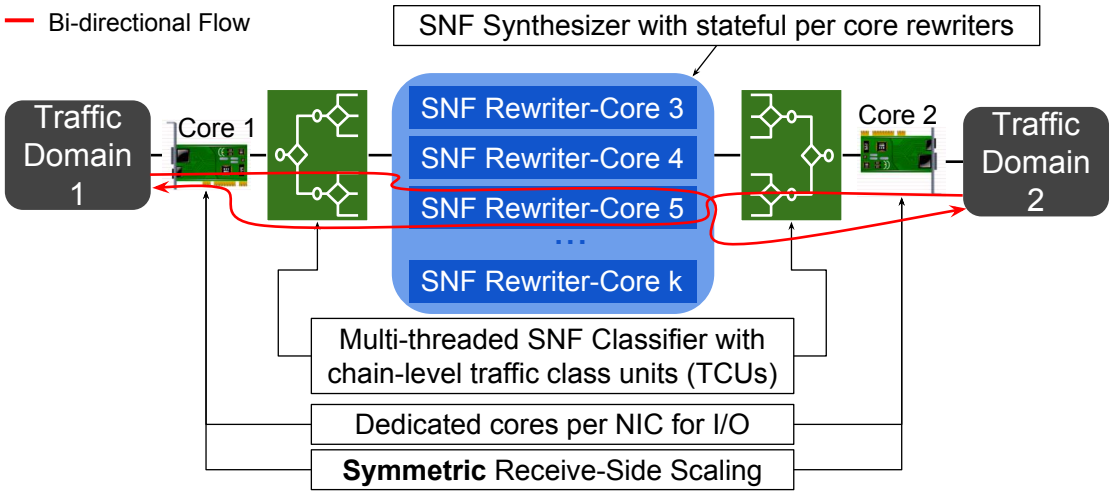


Figure 7.1: SNF running on a machine with k (k>5 in this example) CPU cores and 2 NICs. Dedicated CPU cores per NIC deliver bi-directional flows to packet processing CPU cores via Symmetric RSS. Processing cores concurrently classify traffic and access individual, stateful SNF rewriters to modify the traffic.

7.1.1 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:

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

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

Multiple read operations on the same field occur because each NF contains its own decision elements. A typical example is an IP lookup in a chain of routers. While SNF is parsing the initial chain, it collects the read operations and constructs traffic classes encoded as paths of elements in a DAG. Then, SNF synthesizes these elements into a *single* classifier to realize both routing and filtering.

Multiple write operations on the same field overwrite previous values. For example, the IP checksum is modified twice when a decrement 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.

We address these issues in order to face the third challenge of this thesis, introduced in § 4.2.3. To this end, the next section describes in detail how SNF *automatically* synthesizes the equivalent of a service chain.

7.2 SNF Architecture

Taking into account the main objectives listed above, this section presents the design of SNF: § 7.2.1 defines the synthesis abstraction, § 7.2.2 presents the formal synthesis steps, and § 7.2.3 describes how stateful functions are realized.

7.2.1 Abstract Service Chain Representation

The crux of SNF's design is an abstract service chain representation. First, § 7.2.1.1 describes a mathematical model to represent packet units.

Next, § 7.2.1.2 models an NF's behavior in an abstract way. Finally, § 7.2.1.3 defines the target service-level network function.

7.2.1.1 Packet Unit Representation

Inspired by the approach of [61], we represent each packet as a vector in a multi-dimensional space. However, a protocol-aware approach is followed by dividing a packet according to the unsigned integer value of the different header fields. Thus, if p is a TCP segment encapsulated in an IPv4 packet, it is represented as:

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

From now on, P is the space of all possible packets. For a given header field f of length l bits, a field filter F_f is defined 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 called a *packet filter*, defined as a logical expression of the form:

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

where $(F_1, ..., F_n)$ 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 ϕ can be assimilated to $(F_1, ..., F_n)$.

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

7.2.1.2 Network Function Representation

Network functions typically apply read and write operations to traffic. While the packet unit representation presented in § 7.2.1.1 allows us to compose complex read operations across the entire header space, we still need the means to modify traffic. For this, a packet operation is defined as a function $\omega : P \mapsto \Phi$ that associates a set of possible outputs to a packet. An additional constraint is added such that for any given packet operation ω, there is $\omega_1, ..., \omega_n \in \mathbb{N}^{\mathbb{N}}$ such as:

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

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-NAPT). *Therefore, SNF supports both deterministic and conditional operations.*

Defining Ω as the space of all possible operations, a *Processing Unit (PU)* can be expressed as a conditional function that maps packet filters to operations:

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

where $(\omega_1, ..., \omega_m) \in \Omega^m$ are operations and $(\phi_1, ..., \phi_m) \in \Phi^m$ 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:

$$\begin{aligned} NF_{ROUTER} : PU\{Lookup\} &\rightarrow PU\{DecIPTTL\} \\ &\rightarrow PU\{IPChecksum\} \rightarrow PU\{MAC\} \end{aligned}$$

with 4 PUs: an IP lookup PU is followed by decrement IP TTL, IP checksum update, and source and destination MAC address modification PUs.

7.2.1.3 The Synthesized Network Function

The previous section laid the foundation to construct NFs as graphs of PUs. Now, at the service level where multiple NFs can be chained, a TCU is defined as a set of packets, represented by disjoint unions of packet filters, that are processed in the same fashion (i.e., undergo the same set of synthesized operations), hence are part of a flow or similar flows. This definition allows us to construct the service chain's $SynthesizedNF$ function 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 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 $O(n \cdot \log m)$.

7.2.2 Synthesis Steps

Leveraging the abstractions introduced in § 7.2.1, the steps that translate a set of NFs into an equivalent SNF are detailed in this section. The SNF architecture is comprised of three modules, shown in Figure 7.2. Each module is described in the following sections.

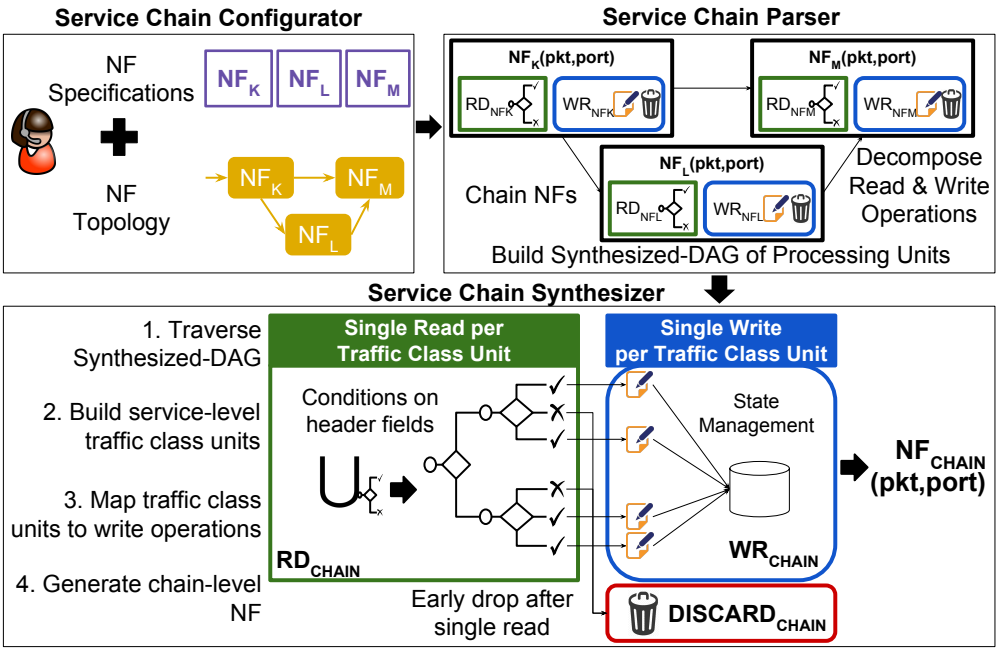


Figure 7.2: The SNF framework. The network operator inputs a service chain and its topology (top left part). SNF parses the chained NFs, decomposes their read and write parts, and composes a Synthesized-DAG (top right part). While traversing the Synthesized-DAG, SNF builds the TCUs of the chain, associates them with write/discard operations, leading to a synthesized chain-level NF.

7.2.2.1 Service Chain Configurator

The top left box in Figure 7.2 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 [68, 34, 33]. 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 7.2). 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, hence two NFs are interconnected as follows:

$$NF_1[interface_1] \rightarrow [interface_0]NF_2$$

No loops: As 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.

7.2.2.2 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 7.2, 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, and 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 7.2) 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 § 7.2.1.2. 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.

7.2.2.3 Service Chain Synthesizer

After building the Synthesized-DAG, the next target is to create the $SynthesizedNF$ introduced in § 7.2.1.3. To do so, the SNF's TCUs need to be derived. To build a TCU the following steps are executed: 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 of the Synthesized-DAG.

Now let us consider a TCU t, defined by its packet filter ϕ and its packet operation ω, that traverses a PU U using the adapted depth-first search. 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 packet 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 packet operation by composing ω and ω_i (line 4).

Algorithm 1 Building the SNF TCUs

```
1: function TRAVERSE(t = (φ, ω), U = {(φ_i, ω_i)_{i≤m}})
2:     for i ∈ (1, m) do0
3:         φ' ← INTERSECT(t, φ_i)
4:         ω' ← ω_i ∘ ω
5:         t' = (φ', ω')
6:         TRAVERSE(t', U.successors[i])
7:     end for
8: end function
```

Algorithm 2 Intersecting a TCU with a filter

```
1: function INTERSECT(t = (φ, ω), φ^0)
2:     φ' ← P
3:     (ω_1, ..., ω_n) ← ω.COORDINATES
4:     (φ_1, ..., φ_n) ← φ.COORDINATES
5:     (φ^0_1, ..., φ^0_n) ← φ^0.COORDINATES
6:     (φ'_1, ..., φ'_n) ← φ'.COORDINATES
7:     for i ∈ (1, n) do
8:         if ω_i = id_ℕ then φ'_i ← φ_i ∩ φ^0_i
9:         else
10:            if ω_i(φ_i) ⊂ φ^0_i then φ'_i ← φ_i
11:            else φ'_i ← ∅
12:            end if
13:        end if
14:    end for
15:    return φ'
16: end function
```

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.

7.2.3 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 ensure that NFV service chains be realized without redundancy, hence single-read and single-write operations must be applied per packet per header field.

To highlight the challenges of maintaining the state in a chain of NFs, consider the example topology shown in Figure 7.3. 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 likely to happen in practice, as an 8-bit network prefix contains less than 17 million addresses and recent surveys have predicted that 50 billion devices will be connected to the Internet by 2020 [109].

Consolidating this chain of NFs into a single SNF instance poses a problem.

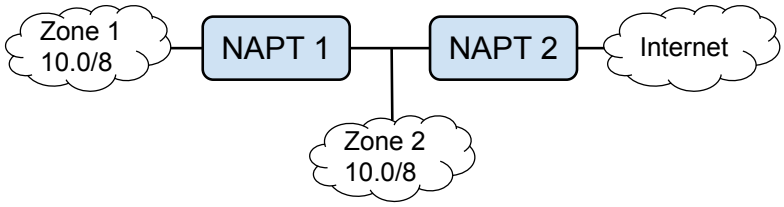


Figure 7.3: Example NAPT chains, where two zones share the same IPv4 prefix.

That is, traffic originating from zones 1 and 2 share the same source IP address and port range, but to ensure that all the traffic is translated properly, the corresponding synthesized chains 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 originates 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:

Property 1: The uniqueness of flow mappings is enforced 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 7.4 illustrates how SNF handles stateful configurations with three egress interfaces. "Property 1" is applied by having exactly one stateful (re)write element (denoted as Stateful RW) per egress interface. "Property 2" is applied 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 link between a flow entry and its origin interface.

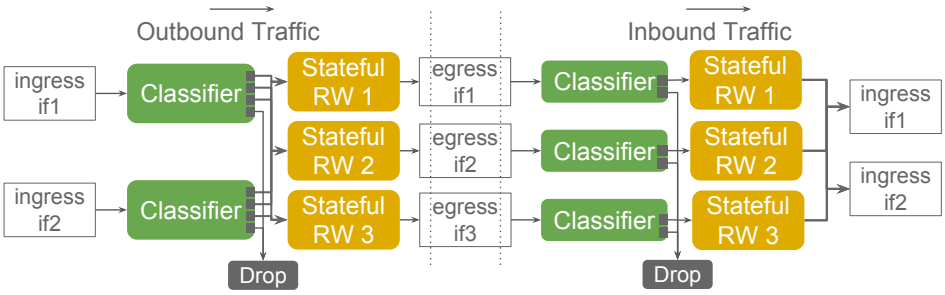


Figure 7.4: State management in SNF.

7.3 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. Click is used to specify the NF DAGs of this example, but SNF is applicable to other frameworks. The example chain consists of a NAPT, a layer 4 FW, and a layer 3 LB that process TCP and UDP traffic as shown in Figure 7.5.

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

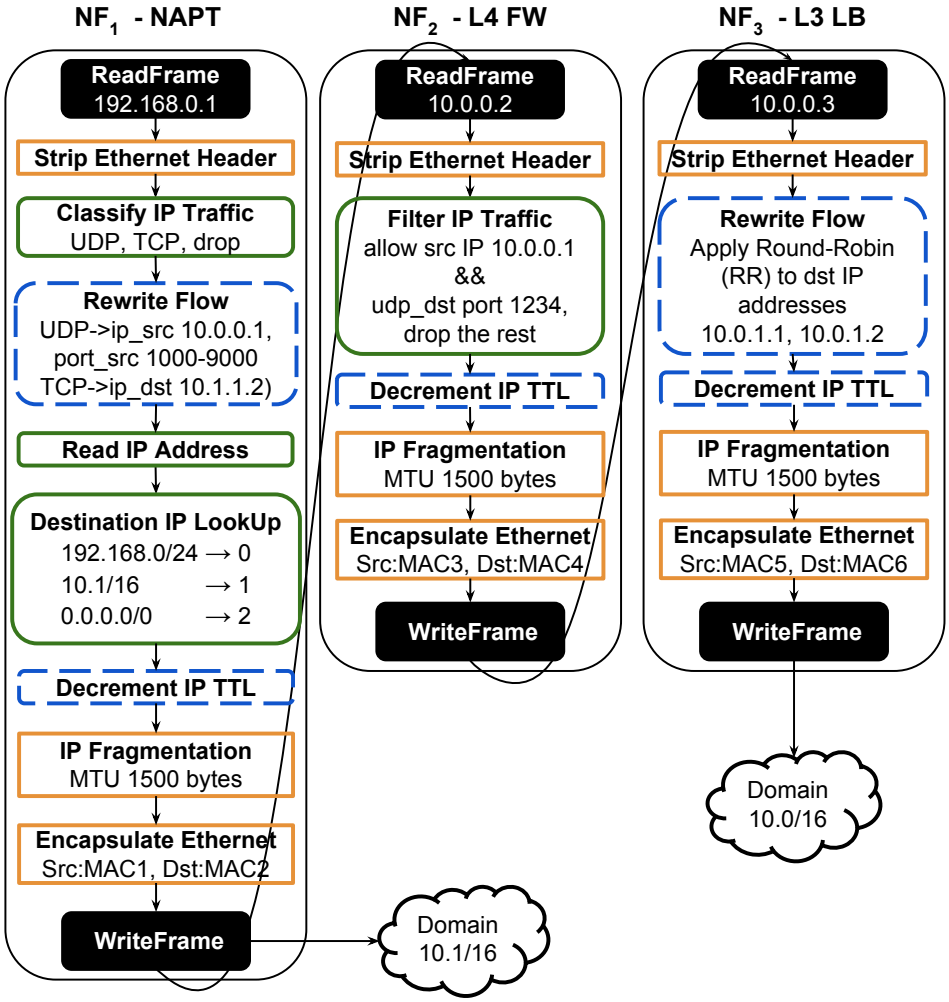


Figure 7.5: The internal components of an example NAPT-L4 FW-L3 LB chain.

The operations of each NF in Figure 7.5 are colored and outlined to highlight their scope, with those filled in black color representing I/O operations, while the unfilled operations do the actual NF processing.

The rectangular operations in Figure 7.5 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"). Redundancy was found 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. The "Rewrite Flow" operation of the first NF modifies the source IP address and port of UDP/IP packets, destination IP address of TCP/IP packets, as well as the network and transport layer (for UDP) checksums. The "Rewrite Flow" operation of the third NF modifies the destination IP address and IP checksum fields of the allowed packets (the rest are dropped by "Filter IP Traffic") while the three "Decrement IP TTL" operations (one per NF) modify the IP TTL and IP checksum fields. The minimal set of rewrite operations that must be applied to the UDP packets by this chain performs a single modification of all these fields. The initial chain calculates the TTL 3 times and IP checksum 5 times.

Based on our measurements on an Intel Xeon processor (see Table 7.1), by comparing Original and no checksum (NoCS) "Decrement IP TTL"/"Rewrite Flow" operations in this table, the checksum calculations cost is 10-40 CPU cycles/packet. Thus combining the "Rewrite Flow" with the "Decrement IP TTL" operations into one synthesized operation and enforcing the checksum calculation only once (CSOnce), saves 237 CPU cycles/packet in this example.

Figure 7.6 depicts a synthesized version of the NF chain shown in Figure 7.5. Following the SNF paradigm presented in § 7.2, the synthesized chain forms a graph with two main parts.

Table 7.1: Median CPU cycles per packet spent by the Click elements used in Figures 7.5 and 7.6 to realize the example chains on an Intel®Xeon® E5-2667 v3 processor. The input rate is 200 kpps and the packet size is 1500 bytes.

Operation	Click Element	CPU Cycles/pkt
Strip Ethernet Header	Strip	59
Encapsulate Ethernet	EtherEncap	70
Read IP Address	GetIPAddress	55
Classify IP Traffic	IPClassifier	150
Filter IP Traffic	IPFilter	155
Destination IP LookUp	RadixIPLookup	150
Decrement IP TTL (Original)	DecIPTTL	81
Decrement IP TTL (No CS)	DecIPTTL	70
Rewrite Flow (Original)	IPRewriter	365
Rewrite Flow (No CS)	IPRewriter	327
Rewrite Flow+Decrement IP TTL (CS Once)	IPRewriter (extended with TTL decrement)	368
IP Fragmentation	IPFragmenter	48

The left-most part (rounded rectangles with solid outline in Figure 7.6) 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 the allowed UDP packets

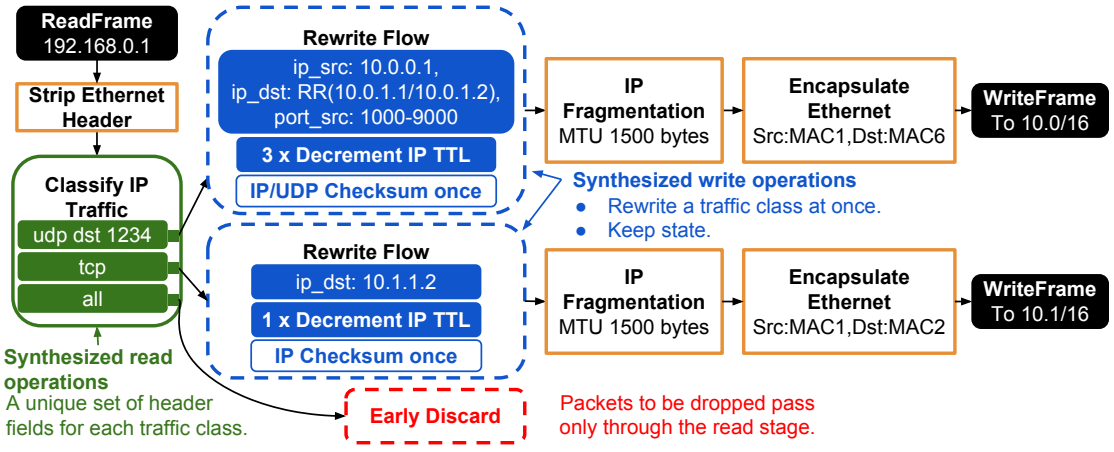


Figure 7.6: The synthesized chain equivalent to Figure 7.5. The SNF contributions are shown in floating text.

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 7.6) associated with this traffic class (right-most part of the graph). As shown in Figure 7.6, 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 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 only calculate the checksums once IP addresses, ports, and the IP TTL fields are written. Therefore, in this example four unnecessary operations (3 "Decrement IP TTL" and 1 "Rewrite Flow") and four checksum calculations (3 IP and 1 IP/UDP) are saved. Moreover, integrating all decisions (i.e., routing, filtering) in one classifier caused the classifier 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 chain. The initial chain implements the same functionality using 18 processing operations and two additional pairs of I/O operations. Based on Table 7.1, the total *processing* cost of the initial chain is 2014 CPU cycles/packet, while the synthesized chain requires 3x less (roughly 695) CPU 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 key role in limiting the system's performance; therefore, the advantages of synthesizing more complex service chains than this simple use case are expected to be even greater.

7.4 Implementation

As 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 maximizes SNF's impact as it allows direct comparison with state of the art Click variants such as RouteBricks [55], PacketShader [56], Double-Click [48], SNAP [110], ClickOS [49], and FastClick [58].

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, NUMA support, and fine grained CPU core affinity techniques, FastClick can realize a single router achieving line-rate throughput at 40 Gbps [58]. *SNF aims for similar performance for an entire service chain.*

7.4.1 FastClick Extensions

SNF was implemented in C++11. The modules depicted in Figure 7.2 are 14376 lines of code. The integration with FastClick required another 1500 lines of code (including modifications and extensions). Although FastClick improves a router's throughput and latency, it lacks features required for broader NFV applications; therefore, the following extensions are made to target a service-oriented platform:

Extension 1: Stateful elements that deal with flow processing (such as IP/UDP/TCPRewriter) were not originally equipped with FastClick's accelerations 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 accelerations.

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, the IP/UDP/TCP checksum calculations were improved by calling the respective functions only once all the header field modifications are applied. Moreover, the IP/UDP/TCPRewriter elements were extended 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 (as shown in Figure 7.1). 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 keeping the per core cache(s) hot, the RSS mechanism of DPDK (see Figure 7.1) was extended using a symmetric approach proposed by Woo and Park [108].

Extension 4: To make software-based classification more scalable, we implemented the lazy subtraction algorithm introduced by HSA [61]. With this extension, SNF aggregates common IP prefixes in a filter and applies the longest one while building a TCU, thus producing shorter TCUs.*

The SNF prototype supports a large variety of packet processing libraries, fully covering both native FastClick and hypervisor-based ClickOS deployments. This 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. The FastClick extensions are available at [111].

7.5 Performance Evaluation

The problems of state of the art NFV frameworks stated in § 4.1.2 hinder large-scale hypervisor-based NFV deployments that could reduce network operators' expenses and provide more flexible network management and services [112, 113].

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.

The experimental setup described in § 7.5.1 is used to measure the performance of the above three types of chains and answer the following questions: Can (stateful) chains with an increasing chain length be synthesized *without* sacrificing throughput (see § 7.5.2 and § 7.5.3)? What is the effect of different packet sizes on a system's throughput (see § 7.5.3)? What are the current limits of purely software-based packet processing (see § 7.5.4.1) and how can we overcome them (see § 7.5.4.2)?

7.5.1 Testbed

Our testbed consists of 6 identical machines. The technical characteristics of these machines were described earlier in Chapter 5. Unless stated otherwise, two

*This extension is not a direct part of FastClick, since the compressed classification rules are computed by SNF beforehand; then, SNF passes these rules to FastClick's classification elements.

machines are used to generate and sink bi-directional traffic. The traffic modules were described in detail in § 5.1. To gain insight into the performance of the service chains, the throughput and end-to-end latency to traverse the chains are measured 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. The two different setups utilized by our software-based and hardware-assisted deployments are:

Software-based experiments
In § 7.5.2, § 7.5.3, and § 7.5.4.1 we stress different purely software-based NFV service chains that run in one machine following the execution model of Figure 7.1. This machine has two dual port 10 GbE NICs connected to the two traffic source/sink machines (two ports per 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 experiments
For the complex NFV service chains, presented in § 7.5.4, we deployed a testbed where the traffic classification is offloaded to a NoviFlow 1132 OpenFlow switch with firmware version 300.1.0. The switch is connected to two 10 GbE NICs via each of the two senders/receivers, and with one 10 GbE link to each of the four processing servers in our SNF cluster. This testbed has a total of 40 Gbps capacity (the same as the software-based setup above), but the processing is distributed to more machines in order to show how our SNF system scales.

7.5.2 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 [55, 56, 48, 110, 49, 58], achieves scalable performance at tens of Gbps.

As shown in this section, a naive chaining of individual, fast NFs does not achieve high performance. To quantify 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 in Figure 7.7 show the throughput achieved by these chains as a function of the increasing length of the chains, when 60-byte frames (excluding the CRC) were injected. As stated earlier in § 4.1.2.2, the maximum throughput for this frame size is 31.5 Gbps and this is the limit of our NICs [58].

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

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

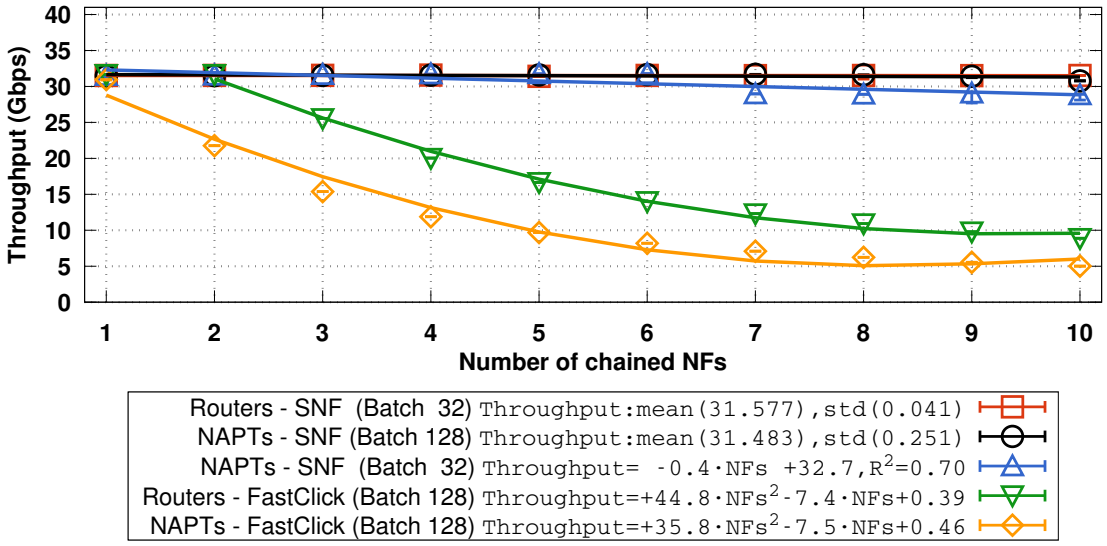


Figure 7.7: Throughput (Gbps) of chained routers and NAPTs using (*i*) FastClick and (*ii*) SNF versus the numbers of chained NFs (60-byte frames are injected at 40 Gbps). Bigger batch sizes achieve higher throughput.

7.5.3 Stateful Service Chaining

The problem of Service Function Chaining has been investigated by Quinn and Nadeau [22] and several relevant use cases [23] 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/Internet Protocol version 6 (IPv6) traffic might either be directed to a NAT64 [114] or a Carrier Grade Network Address Translator (NAT) [115]. In more extreme cases, this traffic might originate from different access networks (such as fixed broadband, mobile, datacenters, or cloud customer premises), thus causing the nested NAT problem [116].

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 7.7 show that the chains of FastClick NAPTs suffer a steeper (according to the fitted equation) quadratic degradation than the FastClick routers. Although FastClick was extended 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 NAPT chains at the hardware speed as shown by the black circles of Figure 7.7. The curve with the 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, the effect of different frame sizes on the chains of routers and NAPTs is explored. We run the longest chains (i.e., 10 NFs) for frame sizes in the range of [60, 1500] bytes. Figure 7.8 shows that SNF matches the NICs' performance and achieves line-rate 40 Gbps throughput for frames larger than 128 bytes. FastClick only achieves similar performance for frame sizes greater than 800-1000 bytes.

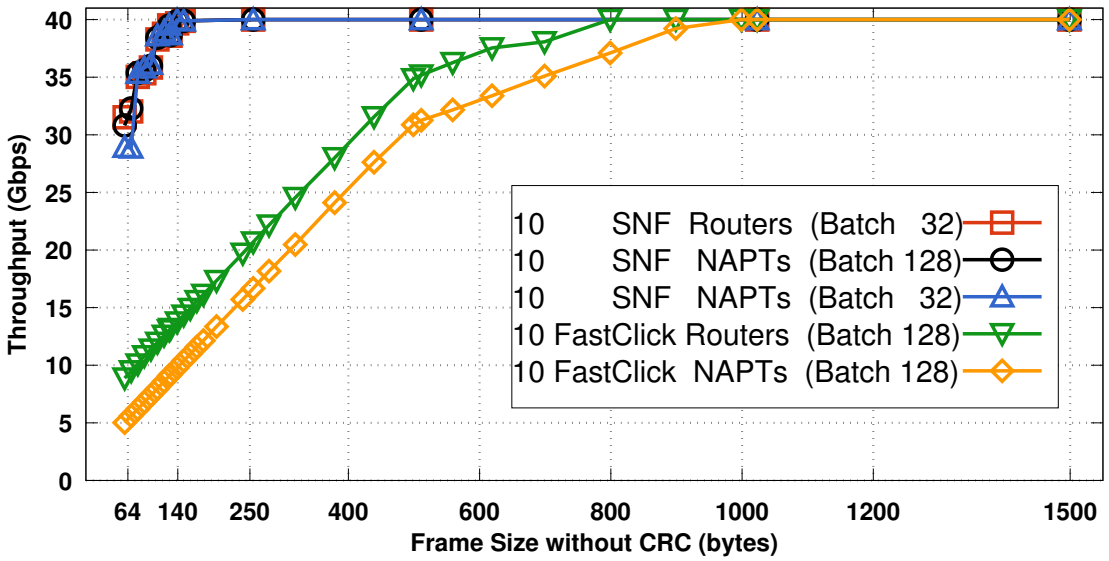


Figure 7.8: Throughput of 10 routers and NAPTs chained using (*i*) FastClick and (*ii*) SNF versus the frame size in bytes (without CRC). The different frames are injected at 40 Gbps.

7.5.4 Performance Analysis of Real Service Chains

A common use case for an ISP is to deploy a service chain of a FW, a router, and a NAPT as depicted in Figure 7.9. The FW of such a chain may contain thousands of rules in its ACL causing serious performance issues for NF implementations.

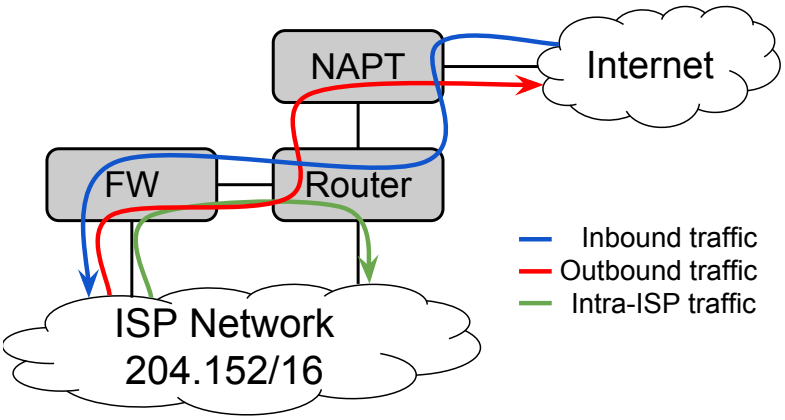


Figure 7.9: An ISP's service chain that serves inbound and outbound Internet traffic as well as intra-ISP traffic using three NFs.

We utilize a set of three ACLs [106], taken from ISPs, to deploy the service chain of Figure 7.9. In these tests 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. The above ACLs were used to generate traces of variable-length frames that systematically exercise all of their entries.

In the following sections the performance of SNF is measured in two different testbeds: § 7.5.4.1 shows how the limits of an Intel® Xeon® CPU E5-2667 are explored using a software-based implementation of the above chains, while in § 7.5.4.2, a hardware-assisted variant of the each of the same chains is employed to assess the performance of SNF in a realistic deployment.

7.5.4.1 Software-based SNF

Figure 7.10 depicts a topology that emulates the service chain of Figure 7.9. Two machines are used as traffic sources/sinks to emulate the ISP and Internet domains, both connected to the service chains via two 10 Gbps NICs.

The service chains are hosted by the machine in the middle, where we instantiate a software-based SNF across all 8 CPU cores of an Intel® Xeon® CPU E5-2667, according to Figure 5.2e. The service chain has two pairs of NICs, each

pair connected to a different domain (i.e., ISP and Internet). For each NIC we use one CPU core to perform I/O, hence 4 CPU cores of the same socket are left for packet processing. The symmetric RSS scheme introduced in § 7.1 is employed to allow a CPU core to drive an SNF TCU at the maximum processing speed of the machine. The same chains were also implemented in FastClick [58], to have a state of the art performance reference for SNF, according to Figure 5.2c.

In the following sections a detailed performance analysis of both (i.e., SNF and FastClick) software-based service chains is provided. First we evaluate the per packet latency imposed by the *read* and *write* stages of SNF and FastClick (see § 7.5.4.1). Then, the overall systems' performance is measured following the setup illustrated in Figure 7.10 (see § 7.5.4.1).

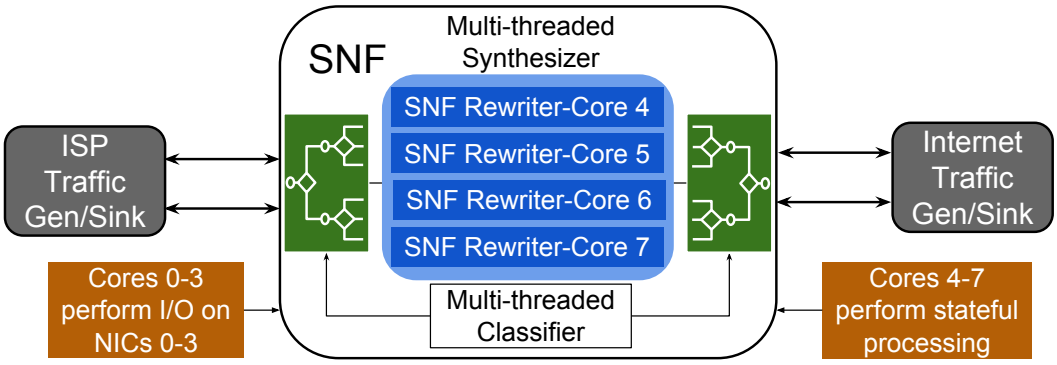
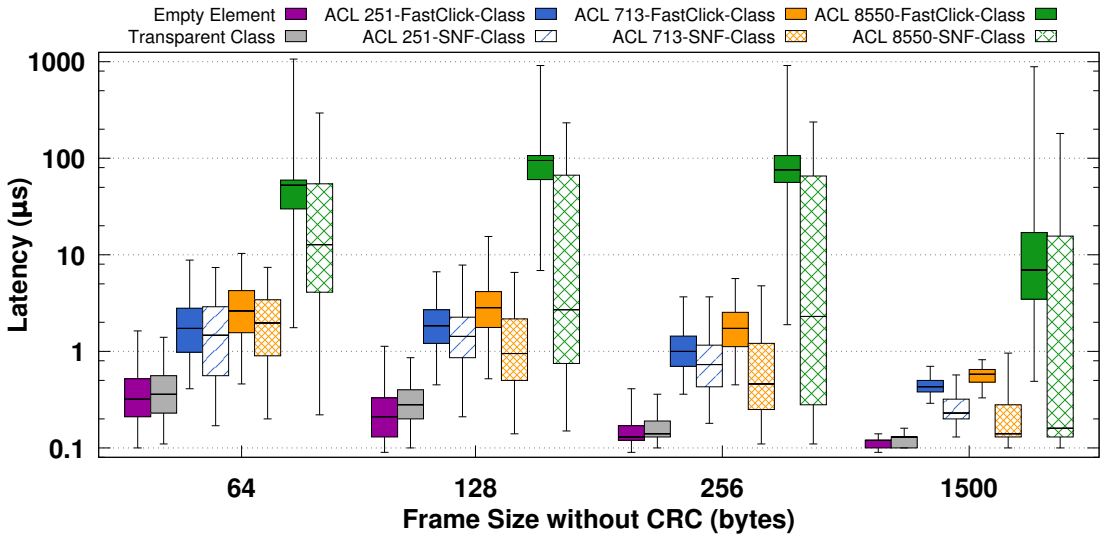


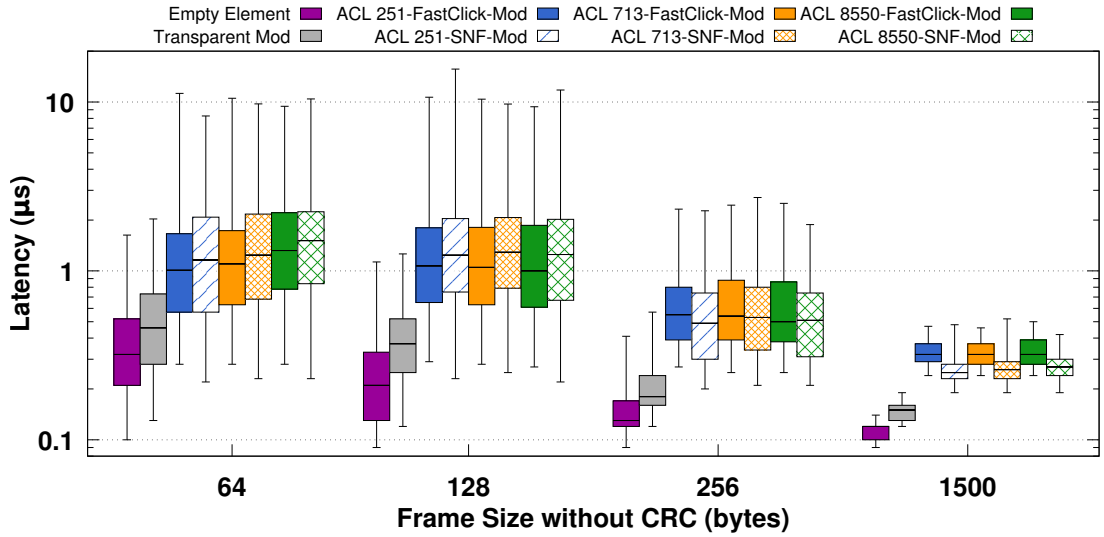
Figure 7.10: Software-based SNF testbed. The ISP and Internet domains are connected to the service chain using 2x10 Gbps NICs each. The service chain has 4x10 Gbps NICs. The machine that executes the service chain uses 4 CPU cores for I/O (one per NIC) and 4 cores (in the same socket) for stateful processing.

Performance of Internal Stages

A set of latency microbenchmarks was conducted within the internal stages of SNF and FastClick. The main target of this measurement campaign is to quantify the exact cost, in terms of latency, of the read and write operations. To do so, the read and write parts of the SNF and FastClick pipelines were isolated and a set of new Click elements were placed before and right after these parts. These elements were used to timestamp each packet with a nanosecond precision, hence when a packet exits the target stage, its' payload contains the delta latency calculated as a time difference between the first (i.e., before the target stage) and the second (i.e., after the target stage) timestamp. For example, in Figure 7.10 we put the SNF synthesizer (rounded blue rectangle inside the SNF box) between two timestamping elements in order to measure the latency of SNF's write operations. Figure 7.11 shows the latencies obtained from these experiments.



(a) Classification.



(b) Modification.

Figure 7.11: Latency (μs), plotted on a logarithmic scale, versus frame sizes (for frame sizes of 64, 128, 256, and 1500 bytes) of the classification (read) and modification (write) stages 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 32 packets. Input rate is 5 Gbps across all the input links.

Four frame sizes (64, 128, 256, and 1500 bytes) were used to inject 100,000 frames per frame size. The latency per frame was measured and plotted as boxplots. Each boxplot corresponds to three latency percentiles (25^{th}, 50^{th}, and 75^{th}) illustrated as bottom, middle, and top horizontal lines respectively. The whiskers correspond to the 1^{st} and 99^{th} latency percentiles.

Figure 7.11a depicts the latency of SNF and FastClick when they execute the read-part (i.e., classification) of the three service chains (i.e., each one corresponding to a different ACL), as a function of four different frame sizes. As a reference, we also measure the latency to traverse an empty Click element (first boxplot of each frame size highlighted with purple color) and the latency to traverse a read element (second boxplot of each frame size highlighted with gray color) that does not contain any traffic conditions (i.e., a classifier with one wildcard entry).

A first observation in Figure 7.11a is that even an empty Click element introduces variance in the per frame latency which spans from almost $100\,ns$ to $2\,\mu s$ for small frames. The reason of this variance is the design of the FastClick platform, as its main target is to provide packet processing functions at very high throughput using kernel bypassing (via DPDK) together with I/O and computational batching. The latter technique has a well known effect on the latency even though we used the minimum batch size (i.e., 32 packets) allowed by DPDK to conduct these experiments above.

Secondly, since the input rate of this experiment is 5 Gbps, but different frame sizes are tested, the input load in terms of the number of packets per second is different among these frame sizes. The corresponding packet rates are 7.44, 4.22, 3.55, and 0.41 Mpps for frame sizes of 64, 128, 256, and 1500 bytes. This is the reason that the latency for small frames is greater than the latency observed for 1500 bytes frames. A complementary observation is that an increasing input load (i.e., in pps) increases the latency variance.

Third, the increasing complexity of the three ACLs leads to increasing latency for both SNF and FastClick. SNF uses a single classifier to consolidate all read operations, as opposed to FastClick that uses a set of read operations per NF in the service chain. This difference results in a substantial latency reduction if we look at the latency of the largest ACL (i.e., 8550 rules). To further clarify this, we see that the 1^{st}, 25^{th}, and 50^{th} percentiles of SNF's latency are 10-20x lower that the respective percentiles of FastClick's latency. Additionally, the 75^{th} percentiles of SNF are lower but comparable to FastClick, while the 99^{th} percentiles exhibit a difference of 2-3x.

Despite the compression that SNF applies to the traffic classes of a chain, latency variance is still observed. For the largest ACL, comparing the 1^{st} and the 99^{th} latency percentiles of Figure 7.11a results in 3 orders of magnitude of

variance (FastClick shows a variance of 4 orders of magnitude for the same ACL). The reason for this variance is the cost of searching the binary trees of SNF's TCUs; based on our formal complexity analysis in § 7.2.1.3, this cost is linear with the number of packet filters in the worst case. Some TCUs of the largest ACL have more than 2000 packet filters, hence some packets might need to traverse all these conditions while being classified. This explains the measured variance of the software-based SNF for the large ACL.

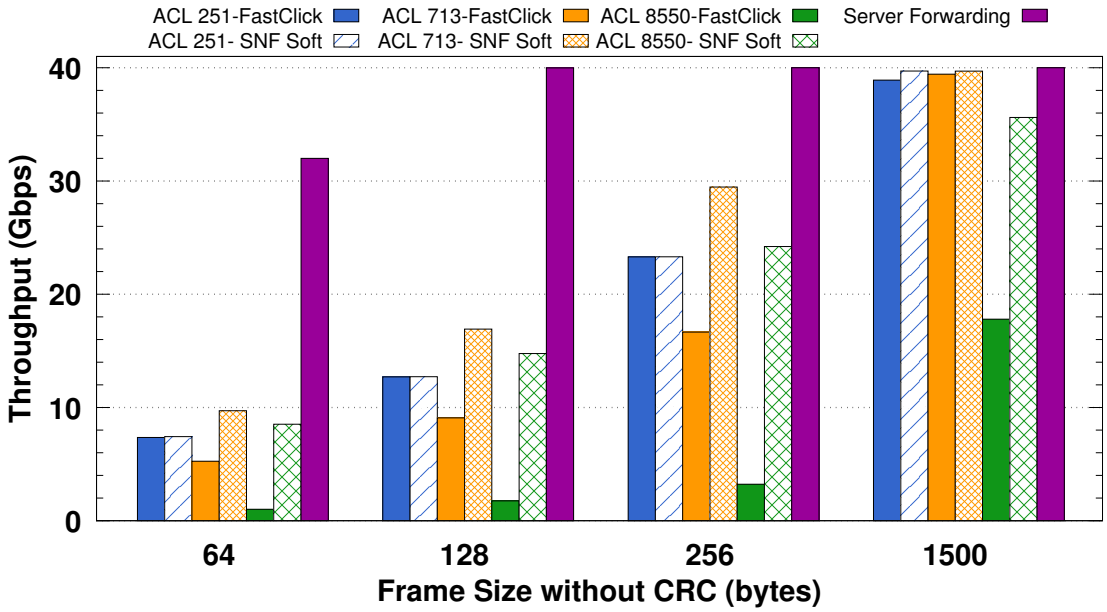
Figure 7.11b shows the latency imposed by SNF and FastClick when applying packet modifications. Generally, the cost of modifying the traffic is lower than the classification cost since the 25^{th} and 75^{th} percentiles span between $100\,ns$ and $2\,\mu s$. This cost is almost 4x greater than the cost of an empty Click element and *independent* of the complexity of the ACLs (which reside in the FW), since the write operations of these chains occur in the router and NAPT. Similarly to the classification case, latency variance is observed; this variance causes the 99^{th} percentiles to be 10x greater (around $10\,\mu s$) than the median latencies.

Finally, although SNF applies write operations with zero redundancy, it appears to incur similar latency to FastClick. The reason is the thread-safe design of the SNF's IPSynthesizer element. This element allows multiple cores to apply synthesized write operations on *independent* TCUs in parallel. However, in this use case there are only four TCUs (bi-directional Intra-ISP and Internet traffic), hence only two TCUs (i.e., one per domain) can run at the same time. This design trades a small performance overhead for safety, although use cases with more TCUs might achieve better performance.

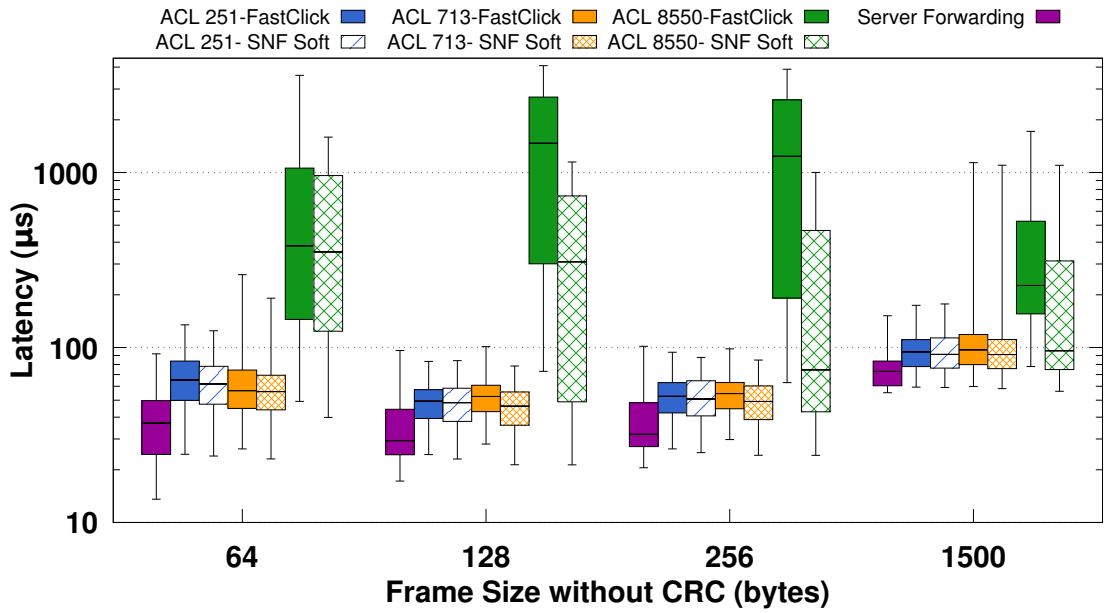
Overall Performance

In this section the focus is shifted towards the overall system's performance. To do so, the throughput and end-to-end latency of the software-based service chains, as observed by the traffic receivers, are measured. Figure 7.12 presents the performance of the 3 chains versus the same four frames sizes. To accurately quantify the cost of each service chain, throughput and latency are measured when the server is simply forwarding traffic between the senders and receivers. This experiment is marked with the label "Server Forwarding" in Figures 7.12a and 7.12b and shows what the underlying hardware can achieve when no processing takes place.

Figure 7.12a 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 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 7.12b, the median latency is also bounded below $100\,\mu$s. Looking at the latency of the "Server Forwarding" case, we notice



(a) Throughput (Gbps).



(b) Latency (μs) on a logarithmic scale.

Figure 7.12: Overall performance of the software-based SNF and FastClick 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. Both frameworks 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 rate is 40 Gbps for the throughput and 5 Gbps for the latency experiments across all the input links.

that this time is dominated by the fact that our traffic is initiated in one machine but performs two hops before being sunk at the destination.

However, for the ACLs with 713 and 8550 rules the combination of all possible traffic classes among the FW, router, and NAPT NFs causes the chain's classification tree to explode in size, hence *synthesis is a powerful yet necessary solution*. This causes three problems for FastClick: (*i*) the throughput when executing the last two ACLs (713, and 8550 rules) is reduced by 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 7.12b), and consequently (*iii*) the 99$^{\text{th}}$ 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 the throughput is better than the 251-rule case. This was possible because the lazy subtraction technique of SNF (see § 7.4.1) achieved high compression rate for the ACL with 713 rules, while the respective compression rate for the small ACL (i.e., 251 rules) was not very high. Regarding latency, SNF demonstrates 1.1-12x lower median latency (bounded below 300 μs) and 1.8-3.5x lower latency variance (the 99$^{\text{th}}$ percentiles are slightly above 1 ms in some cases). These results are generally inline with the latency microbenchmarks shown in § 7.5.4.1 and indicate that FastClick (which is the basis of SNF) is the main cause of the latency variance. However, there is one difference between the latencies shown in Figure 7.12b and the latency microbenchmarks shown in Figure 7.11. That is, in the experiment that involves multiple hops (Figure 7.12b), large frames (i.e., 1500 bytes) exhibit almost two times greater median latency than the smaller frames. This result is in contrast with the findings of Figure 7.11, where the larger a frame is, the lower the latency to classify or modify this frame. We believe that this result is explained by the propagation delays of longer frames. Finally, the throughput of SNF is up to 8.5x greater than for the FastClick implementation of the chain.

7.5.4.2 Hardware-assisted SNF

The results presented in § 7.5.4.1 show that even highly optimized software-based service chains cannot handle packet processing at a high rate for small frames when the NFs are complex, also exhibiting latency variance. To overcome this problem, additional experiments were conducted, in which packet classification is offloaded 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 and hint at the performance that is potentially achievable by offloading packet classification to a programmable network interface.

The topology of this new setup is shown in Figure 7.13. The ISP and Internet domains are connected to the SNF classifier using two 10 Gbps NICs each. The switch classifies the packets and forwards them across four SNF servers that are connected by 10 Gbps links to the switch. Finally, each server forwards the modified traffic back to the traffic receiver module of the origin machine.

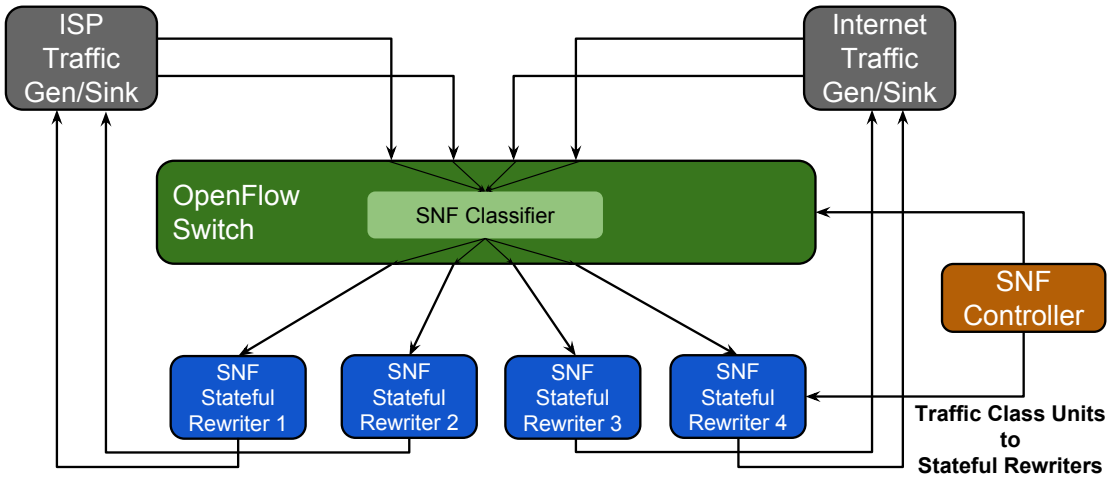


Figure 7.13: Hardware-assisted SNF testbed. Two interfaces per domain (i.e., ISP and Internet) send packets to the hardware-based classifier (ports at the top) of the service chain, realized by an OpenFlow switch. The switch classifies and dispatches input traffic to 4 different output ports connected with a cluster of 4 SNF machines. Each machine uses two NICs: One NIC receives traffic from the switch, while the other NIC forwards the modified traffic back to the ISP or the Internet.

This extended version of SNF includes a script that converts the classification rules computed by the original SNF to OpenFlow v1.3 rules. This translation is not straightforward because the switch rules are less expressive than the rules accepted by the software-based NFs. Specifically, rules that match on TCP and UDP port

ranges are problematic. While OpenFlow only allows 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 within 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 a top-to-bottom order, the same behavior is emulated by assigning decreasing priorities to the OpenFlow rules.

In the following sections the same sets of ACLs as before are used to measure the throughput and latency of the hardware-assisted SNF, showing again the classification, modification, and overall performance of the system.

Classification and Modification Cost

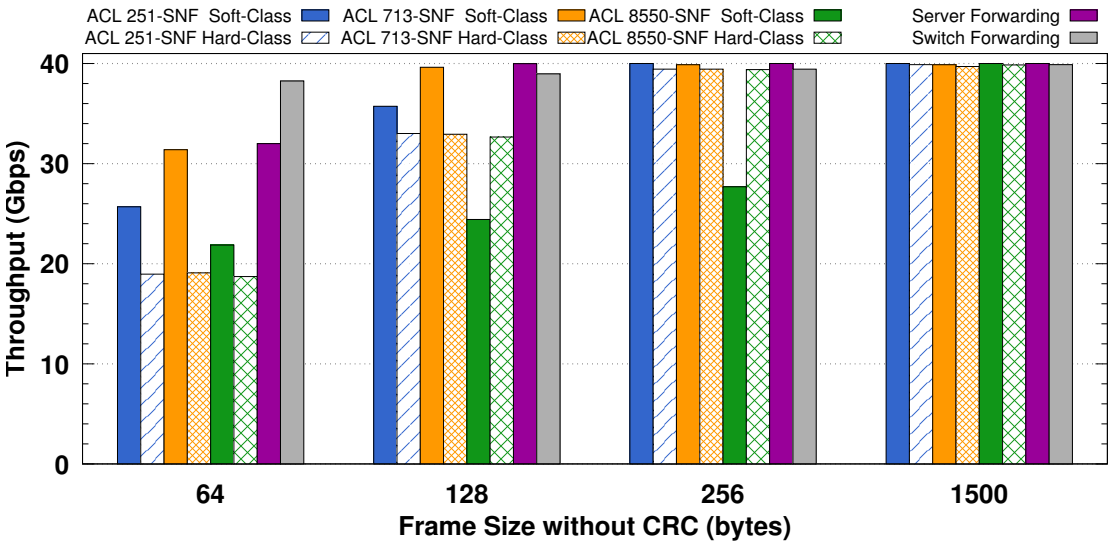
Here the focus is on the performance of the two major parts of the testbed depicted in Figure 7.13, the traffic classification and modification. These performances are measured both on the hardware-assisted and purely software-based SNFs to highlight the benefits of scaling out packet processing.

Classification

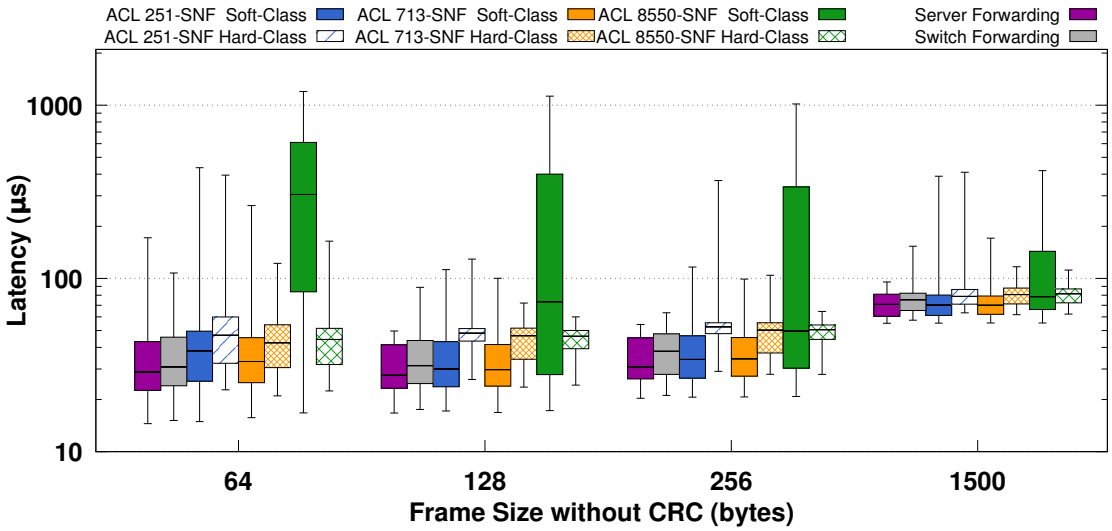
We detached the modification parts of both testbeds to isolate their classifiers. In the testbed of Figure 7.10, we directly encapsulate and output each traffic class derived by the SNF classifiers, while in the testbed of Figure 7.13, the four output ports of the switch are connected back to the origin machines.

Figure 7.14 depicts the throughput and latency of both software-based and hardware-assisted SNF classifiers. To highlight the classification cost in each case, throughput and latency are measured in two extra cases: the NFV server (that runs the software-based classifier) and the switch (that runs the hardware-assisted classifier) act as a forwarders, marked as “Server Forwarding” and “Switch Forwarding” respectively.

The first observation in this experiment is that the software-based classifier outperforms its hardware-based version for the smallest frame size (i.e., 64 bytes). Correlating this result with the experiment where the switch is simply forwarding traffic (marked as “Switch Forwarding”) shows that when the switch performs actual processing of small frames, it achieves poor throughput (see Figure 7.14a). This might be an artifact of the specific switch, but we also acknowledge that a DPDK-enabled NIC with an optimized NFV framework (such as SNF) can compete hardware-based approaches offering cost-effective solutions.



(a) Throughput (Gbps).



(b) Latency (μs) on a logarithmic scale.

Figure 7.14: The performance of the software and hardware-based SNF classification 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. Input rate is 40 Gbps for the throughput and 5 Gbps for the latency experiments across all the input links.

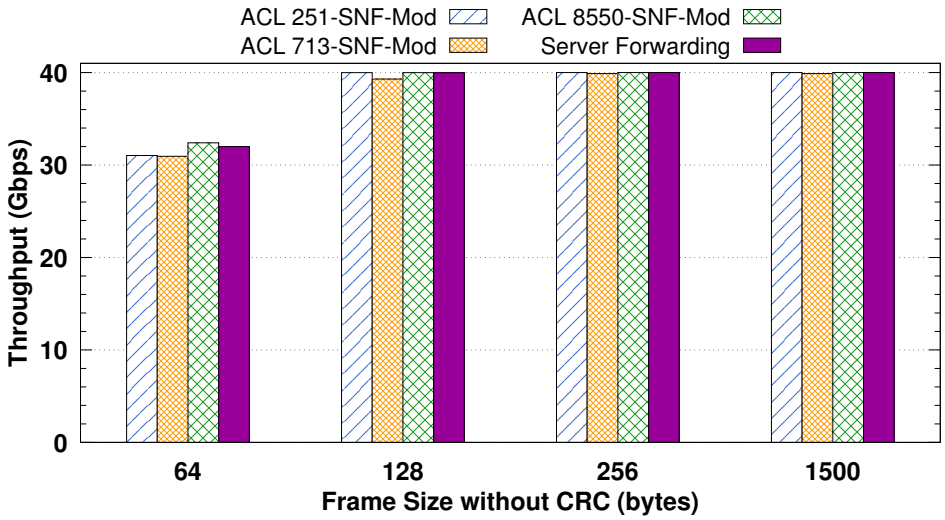
Second, for the ACLs with 251 and 713 rules, the two classifiers achieve comparable throughput (see Figure 7.14a) for all frame sizes and the latency (see Figure 7.14b) of the software-based classifier is lower than the latency of the hardware classifier (excluding some outliers).

However, in the most complex setup (i.e., the largest ACL), an ISP can benefit from the hardware classifier as it achieves 22-48% higher throughput than the software-based classifier. Regarding latency, the hardware classifier seems unaffected by the complexity of the ACLs. More interestingly, although the median latencies of the software-based classifier are comparable (for 3 frame sizes) to its' hardware version, the 75th and 99th percentiles are 10x greater. The reason for this variance was explained in § 7.5.4.1, when latency microbenchmarks were conducted for the software-based SNF and FastClick classifiers.

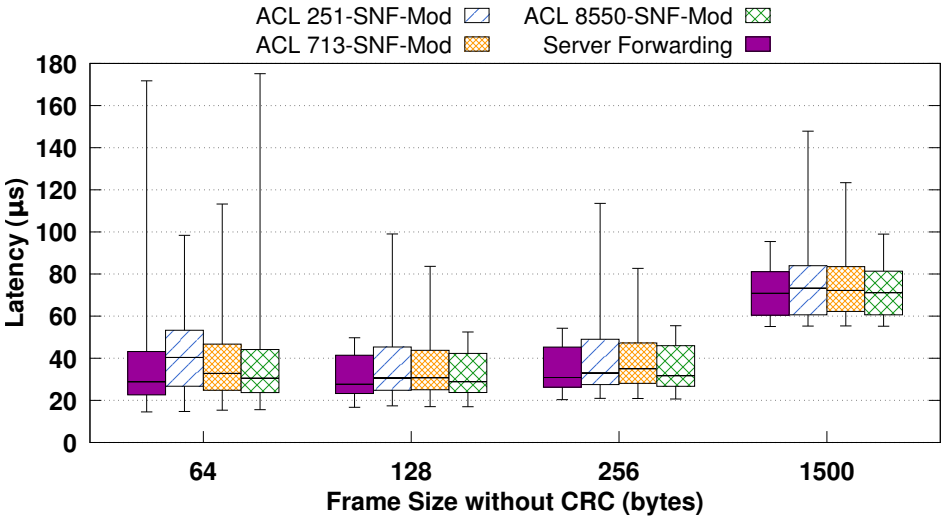
Modification

The traffic modification stage of SNF operates completely in software as it is stateful, hence both software-based and hardware-assisted SNFs pass through the same stage. To isolate this stage we bypass the classifier, by decapsulating the input frames and sending the IP traffic directly to SNF. Then, SNF applies the synthesized write operations, encapsulates the IP packets to Ethernet frames and outputs the traffic back to the origin servers. The write operations are related to the Router and NAPT NFs of these chains, hence they are *independent* of the ACLs (and their complexity). Figure 7.15 shows the throughput and latency of the modification stage.

For each chain four different frame sizes are tested, and as a reference point, the traffic modification performance of each chain is also compared to the "Server Forwarding" case (where the NFV servers simply forward (i.e., read and write) traffic without applying any other operations). As shown in Figure 7.15a, it is obvious that the traffic modifications operate at the speed of the hardware maintaining line-rate throughput. Moreover, these operations add very low latency, as depicted in Figure 7.15b. This finding proves that the real bottleneck of these ISP-level service chains is the classifier.



(a) Throughput (Gbps).



(b) Latency (μs) on a logarithmic scale.

Figure 7.15: The performance of the SNF modification (both software-based and hardware-assisted SNF versions use an identical setup) 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. The packet modification is independent of the complexity of the ACLs. Input rate is 40 Gbps for the throughput and 5 Gbps for the latency experiments across all the input links.

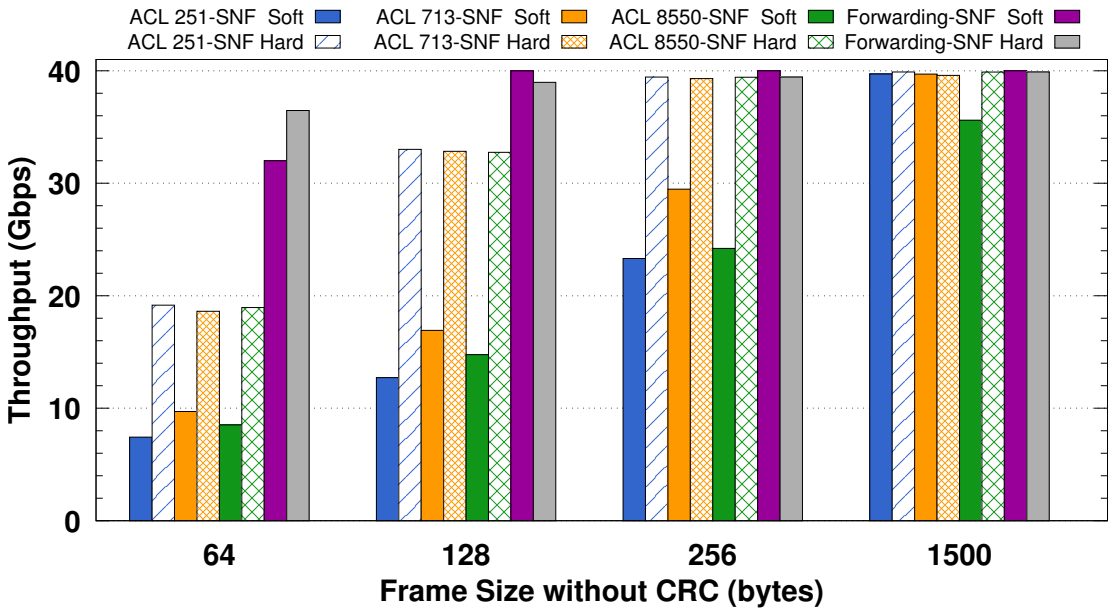
Overall Performance

After examining the performance of the individual hardware-assisted SNF stages, the focus in on the overall system performance. Figure 7.16 shows the throughput and latency of the 3 synthesized chains using a fully-fledged software-based and a hardware-assisted SNF.

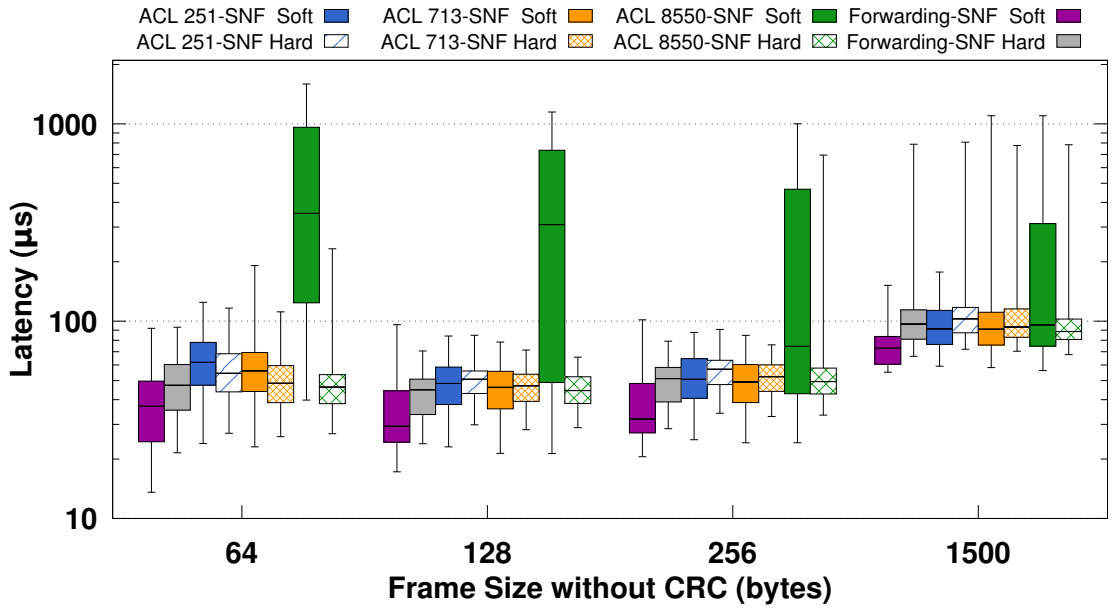
We observe that a fully-functional hardware-assisted SNF is twice as fast as the software-based version for the smallest frames. At first sight, this observation is not inline with the findings of Figures 7.14 and 7.15. Specifically, the isolated software-based SNF classifier (see Figure 7.14a) achieved 20-30 Gbps of throughput for the smallest frames, while the isolated IPSynthesizer operated at the speed of hardware, based on Figure 7.15a. Combining both, substantially decreases the throughput of the software-based SNF. We credit this behavior to the fact that 8 CPU cores might not be enough to realize such complex chains purely in software; we would like to repeat the same experiment using a CPU socket with more cores, but our testbed does not allow this. On the other hand, comparing Figure 7.14a and Figure 7.16a, the hardware-assisted SNF performs as fast as its classifier, which confirms that classification is the bottleneck.

Secondly, the hardware-assisted SNF operates at almost 20 Gbps for minimum size frames, and it reaches line-rate for 256-byte frames. Line-rate processing is also feasible for the software-based SNF, when processing the largest frames (i.e., 1500 bytes). Moreover, we confirm the observations made in § 7.5.4.1: the hardware-assisted SNF is unaffected by the complexity of the ACLs, which is not the case for the software-based SNF.

Figure 7.16b shows the latency of the hardware-assisted and software-based SNFs versus the frame size. As expected, the boxplots of the fully-functional SNF chains (see Figure 7.16b) are similar but slightly shifted upward, compared to the boxplots of the classifiers' latencies depicted in Figure 7.15b. This is due to the additional modification cost. These results also show that the median latencies of the hardware-assisted SNF are low and stable across all frame sizes and chains. Additionally, the 75^{th} percentiles are close to the median latencies and we find this result to be encouraging. Comparing the latencies of the 3 service chains with the baseline latencies of the forwarding cases, we see that SNF imposes very low overhead when processing ISP-level chains. Finally, we confirm the observation made in § 7.5.4.1 about the increased latency of the largest frames.



(a) Throughput (Gbps).



(b) Latency (μs) on a logarithmic scale.

Figure 7.16: The performance of the software-based and hardware-assisted SNF 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 processing occurs in 4 serves connected to the switch. Input rate is 40 Gbps for the throughput and 5 Gbps for the latency experiments across all the input links.

7.6 Verification

This section discusses tools that could potentially be utilized to *systematically* verify the correctness of the synthesis proposed by SNF.

Recent efforts have employed model checking [117, 118] techniques to explore the (voluminous) state space of modern networked systems in an attempt to find state inconsistencies due to bugs, misconfigurations, or other sources. Symbolic execution has also been utilized either alone [119, 120] or combined with model checking [117], to systematically identify representative input events (i.e., packets) that can adequately exercise code paths without requiring exhaustive exploration of the input space (hence bounding the verification time).

Specifically, Software Dataplane Verification [120] might be suitable for verifying NFV service chains. In [120], Dobrescu and Argyraki 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, specifically the traffic classification. However, this tool might not be able to provide sound proofs regarding all the stateful modifications of SNF, since Dobrescu and Argyraki verified only two simple stateful cases (i.e., a NAT and a traffic monitor) and did not generalize their ideas to a broader list of NFV flow modification elements.

SOFT [119] could 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 § 7.2.1.3 outputs packets that are identical with the packets delivered by the original set of NFs. Similarly, HSA [61] 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. SOFT is a special-purpose verification engine for SDN agent implementations. Therefore, both tools would require significant additional effort to verify stateful NFV pipelines.

Finally, translating an SNF processing graph into a Kinetic [118] finite state machine would potentially allow Kinetic 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, it is unclear (*i*) whether a Kinetic model can sufficiently cover complex service chains such as the ISP-level chains presented in § 7.5.4 and (*ii*) whether Kinetic's located packet equivalence classes can handle the complex TCUs of SNF without causing state space explosion.

To summarize, although the works above provide remarkable advancements in software verification, a substantial amount of additional research is required

to provide strong guarantees about the correctness of SNF. As the focus of this thesis is to deliver high speed pipelines for complex and stateful service chains, the verification of SNF is left as future work (see § 9.2).

7.7 Originality and Open Source Contributions

In § 3.2 we studied packet processing architectures, while § 3.3 discussed earlier efforts with respect to network I/O. In § 3.5 middlebox consolidation platforms were discussed. Here the originality of SNF is highlighted with respect to these works.

7.7.1 Modular NFV

The modular nature of NFV frameworks such as Click and the DPDK packet framework facilitates the development process and allows programmers to compose, and extend NFs easily. However, there exists unnecessary redundancy across modules since the modules were developed to be independent.

To the best of our knowledge, our work is the first to depart from Click's current NF-oriented form to enable the same modularity in the service stratum, thus enabling the synthesis of chained NFs as a single Synthesized-Click instance. To achieve the best possible performance, we adopted the FastClick extensions. The SNF extensions (atop FastClick) are available at [111].

7.7.2 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 ClickOS [49] together with fast network I/O [25, 24] and processing [48, 57, 58] mechanisms, service chaining is costly as shown in § 4.1.2. The main reason as shown in our experiments, for this poor performance is the I/O overhead due to forwarding packets along physically separate and virtualized NFs.

We envision NFV deployments that no longer rely on monolithic NFs, but rather permit NF composition with zero redundancy. SNF is a contribution that meets this requirement.

7.7.3 Consolidation at the Machine Level

Concentrating network processing into a single machine is a logical way to overcome the limitations stated above. CoMb [63] consolidates middlebox-oriented flow processing into one machine, mainly at the session layer. Similarly, OpenNF [62] provides a programming interface to migrate NFs, which can in turn be collocated in a physical server. DPIaaS [64] reuses the costly DPI logic across multiple instances. RouteBricks [55] exploits parallelism to scale software routers across multiple servers and cores within a single server, while PacketShader [56] and NBA [57] take advantage of cheap and powerful auxiliary hardware components such as GPUs to provide fast packet processing.

> The works above only partially exploit the benefits of sharing common middlebox functionality, thus they are far from supporting optimized service chains. SNF demonstrated that sharing and synthesizing common functionality can take NFV service chains to the next level of performance.

7.7.4 Consolidation at the Individual Function Level

This consolidation is the next level of composition of scalable and efficient NF deployments. In this context, Open Middleboxes (xOMB) [65] 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 [34] 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 in order to effectively dispatch processing across cores in the same socket.

> Slick does not allow all of 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 processing of the chain. Our synthesized SNF realizes such consolidated chains with zero context switching and zero redundancy of individual packet operations.

Very recently, Bremler-Barr, Harchol, and Hay [68] applied the SDN control and data plane 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 API. The controller communicates the NF specifications to the OpenBox Instances (OBIs) that constitute the actual data plane, 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 as we did 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 7.5. 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 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 rewrite operations of Figure 7.5 into a *single* element (see Figure 7.6) 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 § 7.5.4, the complexity of the chain-level classifier dramatically increases with an increasing number of ACL rules. Therefore, SNF implements the lazy subtraction technique proposed by Kazemian, Varghese, and McKeown [61]. The benefits of this technique were stated in § 7.4.1.

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 in their paper of how OpenBox exploits the multi-core capacities of modern NFV infrastructures.

In contrast, in § 7.5.2, § 7.5.3, and § 7.5.4 we demonstrated how SNF realizes complex, purely software-based service chains at a 40 Gbps line-rate. This is possible by exploiting multiple CPU cores and by fitting most of the data needed by an entire service chain into those cores' L1 caches.

7.7.5 Scheduling NFs for High Throughput

The E2 framework [66] 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.

7.7.6 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 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.

Chapter 8

Contributions

This chapter challenges the hypotheses of this thesis in § 8.1, while § 8.2 sets the publication status for parts of this thesis.

8.1 Challenging the Hypotheses

Table 8.1 summarizes the contributions of this thesis, with an emphasis on the results of SCC and SNF as presented in Chapters 6 and 7 respectively. Using this table, I challenged the hypotheses stated in § 4.4 regarding the research problem. In § 4.1, I showed that the performance of state of the art NFV frameworks supports the null hypothesis H_0: "Service chains inherently exhibit performance degradation that depends upon the length and complexity of the chain", as defined in § 4.4. Then, in the same section I aimed to disprove H_0, by conjecturing the hypothesis H_1: "Some service chains can be realized without their performance deteriorating despite the length and complexity of the chain.".

The two main pillars of this thesis first sought to provide evidence that supports H_1. Chronologically, I first attempted to support H_1 by challenging commodity service chains that rely on unmodified Linux network drivers. To this end, in Chapter 6 I profiled several service chains to uncover their performance problems and applied I/O and scheduling accelerations to rectify those problems. Although my results are not sufficient to support H_1, I found several encouraging outcomes:

Outcome 1: The latency and latency variance of the service chains under test were substantially lower (see Table 8.1) when using SCC. This result has direct impact on popular service chains (such as those described in [86]) that rely on commodity network drivers.

Outcome 2: The SCC Profiler helped me to understand that, currently, only service chains that rely on fast network drivers (such as DPDK) can achieve the desired outcomes, if I find solutions that will overcome their own performance problems.

Table 8.1: A summary of the contributions of this licentiate thesis.

Contribution	Achievements
SCC Profiler	• Profiled a popular NFV framework using Linux and DPDK network drivers in user-space and kernel-space. • Quantified user-to-kernel and kernel-to-user space overheads.
SCC I/O multiplexing	• 3x lower latency and 4x lower jitter for a single user-space NF using the ixgbe network driver. • 10-40% lower latency and 2x lower jitter for user-space NFV service chains.
SCC Scheduling	• 30-300% lower latency and up to 40x lower jitter for chains interconnected with OVSK. • 10-25% lower latency and 2x lower jitter for chains interconnected B2B.
SNF Synthesis of chained NFs	• Multi-core NFV at the speed of L1 caches. • 10 chained routers or NAPTs at the cost of one, achieving line-rate 40 Gbps throughput. • ISP-level chains in software with bounded median latency between 100-500 μs. • ISP-level chains with hardware assistance with bounded median latency below 100 μs and line-rate 40 Gbps throughput.

The SCC I/O multiplexing results shown in Table 8.1 indicate that the latency reduction of a single router (three-fold) is much larger than the latency reduction that SCC achieves for an entire chain of NFs. This reveals that multiplexing is a useful solution, but not drastic enough. The SCC scheduling results also indicate that the solution is not simply better scheduling - even when scheduling is combined with I/O multiplexing. These observations led me to follow more drastic approaches.

Using my first contribution as a base, we then attempted to radically revise the way NFV service chains are realized, by proposing SNF in Chapter 7. The synthesis approach designed in § 7.2 realized service chains at the speed of our hardware, as shown in § 7.5. We elaborate on this in the following paragraphs.

The H_1 hypothesis has two distinct parts. The first relates the performance of a service chain with its length. In § 7.5.2 we demonstrated how SNF can realize service chains of *increasing length* without performance degradation. As also shown in Table 8.1, our results support the first part of H_1 for chain lengths in the range $[1, 10]$. The second part of H_1 relates the performance of a service chain with its complexity. We took three example ISP-level service chains of increasing complexity (see § 7.5.4) and demonstrated that a hardware-assisted SNF can realize these chains without performance degradation (see § 7.5.4.2). These results support the second part of H_1 for service chains that contain a number of rules, in the range of $[1, 8550]$ rules in their classifier(s).

As a result, we believe that SNF provides sufficient evidence to support H_1. To put this thesis into perspective, in § 4.2.3 we showed that the upcoming generation of networks will pose strict latency requirements for applications. In this context, service chains need to deliver traffic with a sub-millisecond latency. Based on Table 8.1, SNF fulfills this requirement as:

Outcome 3: ISP-level service chains, realized in software, impose a median latency between 100-500 μs, while the 99$^{\text{th}}$ percentiles of the latency rarely (and slightly) exceed 1 ms (see § 7.5.4.1).

Outcome 4: ISP-level service chains, realized with hardware assistance of an OpenFlow switch (acting as a classifier), impose a median latency less than 100 μs. This latency is an order of magnitude lower than the target 1 ms latency, while the 99$^{\text{th}}$ percentiles of the latency never exceed 800 μs for the example chains (see § 7.5.4.2).

8.2 Publication Targets and Status

SCC and SNF, have been submitted as articles to international journals. The submission details and the status of each article are shown in Table 8.2.

Table 8.2: Journal articles and their status.

Contribution	Target Journal	Submission Date	Status
SCC	Elsevier Journal of Systems and Software	September 03, 2016	Accepted (Available at [95])
SNF	PeerJ Computer Science	September 23, 2016	Accepted (Available at [101])

Chapter 9

Limitations and Future Work

SCC and SNF are the two main pillars of this dissertation (see Chapters 6 and 7 respectively). In § 9.1 we discuss the limitations of these two works. Then, in § 9.2 we sketch a plan for future work that builds upon this thesis.

9.1 Limitations

SCC limitations: Some of the tools that SCC exploits can provide the desired functionality regardless of the underlying hardware. For example, lmbench can measure the cache and main memory latencies for a broad set of hardware architectures. However, to build an accurate system profiler, one relies on data that the underlying hardware provides. SCC achieves accuracy by making use of Intel PCM and Perf, which are particularly designed to collect data using Intel-related performance counters. We adopted this hardware-specific approach trading generality for high-precision profiling results. We believe that these results can assist cloud providers to automatically uncover performance problems of NFV services running on Intel platforms, however the emergence of ARM and other platforms in datacenters cannot be analyzed using the current implementation of SCC.

SNF limitations: We do not attempt to synthesize arbitrary software components, but rather target a broad but finite set of middlebox-specific NFs that operate on 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., using the NF DAG defined in § 7.2.2.1). We believe that this is a reasonable requirement that was also adopted by other state of the art approaches (such as Click, Slick, and OpenBox). However, if an NF provider does not want to share this information, even under

non-disclosure or via a licensing agreement, then SNF can synthesize the NFs before and after this provider's NF. This is possible by omitting the processing graph of this NF from the inputs given to the Service Chain Configurator (see § 7.2.2.1).

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 be synthesizable. In such a case, SNF can exclude the LB from the synthesis.

Moreover, SNF does not support network-wide placement of NFV chains, but we envision SNF being integrated in controllers, such as E2 or Slick.

9.2 Future Work

SCC extensions: We aim to further improve the I/O performance of SCC by integrating the asynchronous, zero-copy network I/O solution proposed by Drepper [42] into FastClick. Based on our measurements reported in § 6.5.1, this integration is expected to reduce the end-to-end latency imposed by a user-space FastClick router by ~10x, as currently the best median latency of our fully-featured user-space router is 80 μs and a DPDK router achieves a median latency of 8 μs (as per § 6.3.1). Another attractive research direction would be to enforce the correct execution order of the chain in a *new, service chain-oriented* scheduler, thus completely eliminating the scheduling overheads identified by the SCC Profiler (see § 6.3.2.1).

SNF extensions: SNF would benefit from an auxiliary tool that systematically verifies the output service chain, ensuring that it exhibits the identical functionality of the original service chain. As stated in § 7.6, this is a very challenging task that requires substantial improvements in current state of the art frameworks to handle synthesized operations.

Moreover, we would like to study the possibility to offload the classification part of SNF into modern commodity NICs. This would allow a direct comparison of the classification performance between our OpenFlow-based NoviFlow switch (see § 7.5.4.2) and the NIC.

Other future plans: We are excited about the challenges of the 100 GbE era [82, 83]. In this context, networked systems will face new performance issues, leading to several interesting questions:

Future Question 1: Can existing general purpose hardware architectures realize packet exchanges between a 100 GbE NIC and a CPU at a line-rate of 100 Gbps?

Future Question 2: How can we overcome the performance issues of the Linux kernel, to allow packet generation and reception at a line-rate 100 Gbps?

Future Question 3: Is it possible to extend the firmware of modern NICs so that packets can be generated in hardware (by a NIC), thus eliminating the need to customize general purpose OSs (such as Linux)?

Chapter 10

Sustainability, Ethical, and Security Issues

Before concluding this thesis, it is important to position our work in today's societal, ecological, and economical planes. To this end, we discuss sustainability, ethica, and security issues regarding this thesis, in § 10.1 and § 10.2 respectively.

10.1 Sustainability

On a daily basis, people take decisions and actions that have an impact on the environment. In order for current and future generations to live in prosperity, we need to protect the ability of the environment to support human life. Based on the Brundtland Report, from the United Nations World Commission on Environment and Development [121], we define the term "sustainable development" as follows:

> Sustainable development is development that meets the needs of the present without compromising the ability of future generations to meet their own needs.

There are three different components to sustainability: environmental, societal, and economical. Note that the order of these components is important. Academic research, as part of science, ought to contribute to a sustainable global ecosystem, hence contributing to each of these components. Therefore, academic research must tackle the emerging research problems with solutions that can both (*i*) effectively solve the problem and (*ii*) do so, in a sustainable way.

The contributions of this thesis have shown that NFV service chains can be realized with improved performance - which offers the following societal, and economical sustainability benefits:

SCC contribution to sustainability: Our first contribution (called SCC in Chapter 6) improves the cache utilization of NFV systems by combining I/O and scheduling accelerations. We practically demonstrated this in § 6.5. Consequently, the more data SCC fits into a core's cache(s), the fewer transactions are required with main memory, hence the fewer CPU cycles are spent by the system to process this data.

SNF contribution to sustainability: Similarly, the second contribution (called SNF in Chapter 7) dramatically increases the capacity of NFV systems by consolidating an entire service chain into a few, synthesized processing elements. As explained in § 7.1, SNF maintains highly-correlated data with respect to the system's caches. We demonstrated this capacity by showing how SNF realizes long and stateful chains (see § 7.5.2 and § 7.5.3), as well as complex ISP-level chains (see § 7.5.4) at the speed of hardware.

To understand the implications of using SCC and SNF in a real NFV environment, consider the following example. A cloud provider that currently uses 10 machines to accommodate a given number of NFV service chains, might be able to either (*i*) use fewer of these machines or (*ii*) increase the number of service chains that run in these same machines by replacing the current NFV technology with SCC or SNF. Processing power in computer systems is highly-correlated with energy consumption, hence both of our contributions have a direct impact on reducing the energy consumption of NFV systems. This means that SCC and SNF contributes to reducing the power consumption of datacenters, leading to environmentally sustainable contribution to information and communication technology.

With regard to societal aspects of sustainability, throughput and latency are two important aspects of quality in communication systems. We demonstrated that SCC and SNF reduce the end-to-end latency and that SNF maintains line-rate throughput despite increasing the length or complexity of the service chains. Therefore, we believe that this thesis also contributes to societal sustainability both by increasing end-user satisfaction and because lower latency translates into higher productivity and more time for other activities.

Accommodating more services within the same infrastructure and having the customers' satisfied brings economic benefits to NFV stakeholders. Datacenter providers, network operators, ISPs, etc. can both save and make money by utilizing the contributions described in this thesis. Saving are possible by postponing investments, provided that an NFV stakeholder adopts efficient solutions that better exploit their available resources. Gains are possible by using the resources saved by the efficient solutions described above to increase the capacity of the system (i.e., serve more users or services). By using SCC, we showed that

it is possible to eliminate some overheads stemming from suboptimal I/O and scheduling. SNF can also eliminate the redundant operations in a pipeline of chained NFs. Therefore, both contributions of this thesis can save resources, hence bring increased profits to NFV stakeholders.

Finally, although the two contributions of this thesis approach the same problem, they employ orthogonal solutions. This means that an NFV stakeholder might benefit from both SCC and SNF at the same time. However, comparing the two contributions, from a sustainability point of view, we conclude that SNF is a more sustainable solution than SCC. Here is our reasoning. As the demands for ultra low latency services continue to increase [88, 89, 90], it is more and more necessary to utilize highly-optimized NFV solutions that consolidate traffic processing. This need served as an inspiration for SNF, as stated in § 4.2.3. We believe that in the near future, service chains will no longer be realized as multiple processes chained together, thus the need to apply scheduling (as per SCC) will be less important than it is nowadays, when most cloud providers run NFs in individual VMs or containers. For this reason, SNF is likely to have long-term viability and to have a greater impact on the sustainability of an NFV ecosystem. We also believe that scheduling will remain still important, hence one might use some of SCC's principles to coordinate the execution of different service chains running on top of the same hardware.

10.2 Ethical and Security Issues

Ethical and security issues arise in many areas of research. This section reports on the stance that the thesis takes with respect to ethical and security requirements.

No ethical issues have been raised in this thesis, but there are some security issues raised by both SCC and SNF. Both SCC and SNF increase the impact of a cyberattack that can exploit the increased concentration of the NFV functionality. First, this increased concentration occurs because these contributions increase the NFV consolidation, leading to an NFV stakeholder using fewer machines to accommodate the same traffic demand. As a result of this, the machines are more highly utilized (hence there is less unused capacity). Moreover, if the SCC or SNF mechanisms themselves were targeted by an attack, the impact would be large - for example, purposely reducing the performance or adding malicious processing of the packets.

Chapter 11

Conclusions

This thesis has addressed the performance problems of NFV service chains, when using (*i*) commodity and (*ii*) state of the art kernel-bypassing network drivers.

First, we studied chained NFs that use unmodified network drivers on top of a Linux OS. By automatically mining the hardware and OS-level performance counters of NFV service chains using our SCC Profiler, cloud providers can quickly and reliably identify performance bottlenecks. To prove this, we built the SCC run-time: a platform that employs I/O multiplexing and modified scheduling techniques to address the performance issues pinpointed by the SCC Profiler. Our systematic performance evaluation of standalone and chained service chains showed that SCC achieves (*i*) 3x lower end-to-end latency and (*ii*) 2x (up to 40x for certain percentiles) lower latency variance compared to a baseline solution. To achieve this performance, SCC reduces both: (*i*) cache misses and (*ii*) scheduling overheads.

Secondly, to further improve the performance of chained NFs, we have designed and implemented SNF, a framework that synthesizes NFV service chains. SNF requires minimal I/O interactions with the NFV platform and applies single-read-single-write operations on the packets, early discards irrelevant traffic classes, while maintaining state across NFs. To realize the above properties, we parse the chained NFs and build a classification graph whose leaves represent unique traffic classes. Each leaf is associated with a set of packet header modifications for which we generate equivalent code that implements the same chain using a minimal set of elements. SNF synthesized long and stateful chains have been shown to operate at a line-rate of 40 Gbps. Using an OpenFlow switch as a classifier, SNF operated at 40 Gbps for three example ISP-level deployments.

The results of this thesis have shown how to fulfill the promise of next generation networks to deliver sub-millisecond latency for service chains that are expected to be typical of those deployed in the near future.

Bibliography

[1] G. Anagnostopoulos, *A Companion to Aristotle*, ser. Blackwell Companions to Philosophy. Wiley, 2009. [Online]. Available: https://books.google.se/books?id=_BhqbWOnnK4C

[2] J. N. Gupta and S. K. Sharma, *Intelligent Enterprises of the 21st Century*. Idea Group Pub., 2004. [Online]. Available: https://books.google.se/books?id=GEAyWTqFnJkC

[3] Cisco, "Visual Networking Index (VNI), The Zettabyte Era-Trends and Analysis," Jul. 2016, Document ID: 1465272001663118. [Online]. Available: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/VNI_Hyperconnectivity_WP.html

[4] IBM Institute for Business Value, "Five telling years, four future scenarios," 2010. [Online]. Available: http://www-05.ibm.com/cz/gbs/study/pdf/GBE03259USEN.PDF

[5] G. Linden, "Make Data Useful," in *Data Mining (CS345) class at Stanford*, Nov. 2006. [Online]. Available: https://sites.google.com/site/glinden/Home/StanfordDataMining.2006-11-28.ppt

[6] Chauver Professional, "Information Technology for the Lighting Professional – The importance of Throughput," Sep. 2015. [Online]. Available: https://www.chauvetprofessional.com/information-technology-for-the-lighting-professional-the-importance-of-throughput/

[7] Bundeskriminalamt (BKA), "ICT Security Market: The Facts and Figures," 2010. [Online]. Available: http://www.t-systems.com.my/umn/uti/705920_1/blobBinary/Trendwatch-ps.pdf?ts_layoutId=715686

[8] B. E. Carpenter and S. W. Brim, "Middleboxes: Taxonomy and Issues," Internet Request for Comments (RFC) 3234 (Informational), Internet Engineering Task Force, Feb. 2002. [Online]. Available: http://www.ietf.org/rfc/rfc3234.txt

[9] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making Middleboxes Someone else's Problem: Network Processing As a Cloud Service," in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '12. New York, NY, USA: ACM, 2012, pp. 13–24. [Online]. Available: http://doi.acm.org/10.1145/2342356.2342359

[10] G. E. Moore, "Readings in Computer Architecture," M. D. Hill, N. P. Jouppi, and G. S. Sohi, Eds. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, ch. Cramming More Components Onto Integrated Circuits, pp. 56–59. [Online]. Available: http://dl.acm.org/citation.cfm?id=333067.333074

[11] K. Sällberg, "A Data Model Driven Approach to Managing Network Functions Virtualization: Aiding Network Operators in Provisioning and Configuring Network Functions," Master's thesis, KTH Royal Institute of Technology, School of Information and Communication Technology, Kista, Sweden, Jul. 2015, TRITA-ICT-EX-2015:159. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-171233

[12] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "SIMPLE-fying Middlebox Policy Enforcement Using SDN," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 27–38. [Online]. Available: http://doi.acm.org/10.1145/2486001.2486022

[13] European Telecommunications Standards Institute, "Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges & Call for Action," 2012. [Online]. Available: https://portal.etsi.org/NFV/NFV_White_Paper.pdf

[14] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1355734.1355746

[15] M. Kobayashi, S. Seetharaman, G. Parulkar, G. Appenzeller, J. Little, J. Van Reijendam, P. Weissmann, and N. Mckeown, "Maturing of OpenFlow and Software-defined Networking Through Deployments," *Comput. Netw.*, vol. 61, pp. 151–175, Mar. 2014. [Online]. Available: http://dx.doi.org/10.1016/j.bjp.2013.10.011

[16] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving High Utilization with Software-driven WAN," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 15–26. [Online]. Available: http://doi.acm.org/10.1145/2486001.2486012

[17] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a Globally-deployed Software Defined WAN," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 3–14. [Online]. Available: http://doi.acm.org/10.1145/2486001.2486019

[18] A. Gupta, L. Vanbever, M. Shahbaz, S. P. Donovan, B. Schlinker, N. Feamster, J. Rexford, S. Shenker, R. Clark, and E. Katz-Bassett, "SDX: A Software Defined Internet Exchange," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, pp. 551–562. [Online]. Available: http://doi.acm.org/10.1145/2619239.2626300

[19] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat, "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015, pp. 183–197. [Online]. Available: http://doi.acm.org/10.1145/2785956.2787508

[20] D. Meyer, "RCR Wireless News: Many telecom operators planning NFV deployments this year," June 15, 2015. [Online]. Available: http://www.rcrwireless.com/20150615/network-function-virtualization-nfv/35-of-telecom-operators-plan-nfv-deployments-this-year-tag2

[21] L. Hardesty, "IHS Report: Most NFV Deployments Start with vCPE," Aug. 2016, Accessed: October 20, 2016. [Online]. Available: https://www.sdxcentral.com/articles/news/ihs-report-nfv-deployments-start-vcpe/2016/08/

[22] P. Quinn and T. Nadeau, "Problem Statement for Service Function Chaining," Internet Request for Comments (RFC) 7498 (Informational), Internet Engineering Task Force, Apr. 2015. [Online]. Available: http://www.ietf.org/rfc/rfc7498.txt

[23] W. Liu, H. Li, O. Huang, M. Boucadair, N. Leymann, Q. Fu, Q. Sun, C. Pham, C. Huang, J. Zhu, and P. He, "Service Function Chaining (SFC) General Use Cases," Internet Request for Comments (RFC) Working Draft, IETF Secretariat, Internet-Draft draft-liu-sfc-use-cases-08, September 2014, Expired on March 21, 2015. [Online]. Available: https://tools.ietf.org/html/draft-liu-sfc-use-cases-08

[24] DPDK, "Data Plane Development Kit (DPDK)," 2016, Accessed: October 17, 2016. [Online]. Available: http://dpdk.org

[25] L. Rizzo, "Netmap: A Novel Framework for Fast Packet I/O," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 9–9. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342821.2342830

[26] N. Bonelli, A. Di Pietro, S. Giordano, and G. Procissi, "On Multi—gigabit Packet Capturing with Multi—core Commodity Hardware," in *Proceedings of the 13th International Conference on Passive and Active Measurement*, ser. PAM'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 64–73. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-28537-0_7

[27] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali, "A Case for NUMA-aware Contention Management on Multicore Systems," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 557–558. [Online]. Available: http://doi.acm.org/10.1145/1854273.1854350

[28] Intel, "Data Direct I/O Technology," Accessed: October 17, 2016. [Online]. Available: http://www.intel.co.jp/content/dam/www/public/us/en/documents/white-papers/data-direct-i-o-technology-overview-paper.pdf

[29] D. M. Ritchie, "A Stream Input-Output System," *AT&T Bell Laboratories Technical Journal*, vol. 63, pp. 311–324, 1984.

[30] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner, "Router Plugins: A Software Architecture for Next Generation Routers," in *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '98. New York, NY, USA: ACM, 1998, pp. 229–240. [Online]. Available: http://doi.acm.org/10.1145/285237.285285

[31] D. Cinege, "The Linux Router Project: A look at one of the fastest growing Linux distributions, that you may never actually see," *Linux Journal*, no. 59, Mar. 1999. [Online]. Available: http://www.linuxjournal.com/article/3223

[32] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small Forwarding Tables for Fast Routing Lookups," in *Proceedings of the ACM SIGCOMM '97 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '97. New York, NY, USA: ACM, 1997, pp. 3–14. [Online]. Available: http://doi.acm.org/10.1145/263105.263133

[33] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click Modular Router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, Aug. 2000. [Online]. Available: http://doi.acm.org/10.1145/354871.354874

[34] B. Anwer, T. Benson, N. Feamster, and D. Levin, "Programming Slick Network Functions," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR '15. New York, NY, USA: ACM, 2015, pp. 14:1–14:13. [Online]. Available: http://doi.acm.org/10.1145/2774993.2774998

[35] DPDK, "Packet Framework," 2016, Accessed: October 17, 2016. [Online]. Available: http://dpdk.org/doc/guides/prog_guide/packet_framework.html

[36] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-independent Packet Processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2656877.2656890

[37] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "OpenState: Programming Platform-independent Stateful Openflow Applications Inside the Switch," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 44–51, Apr. 2014. [Online]. Available: http://doi.acm.org/10.1145/2602204.2602211

[38] European Union Horizon 2020 project, "BEhavioural BAsed forwarding (BEBA)," 2015-2017, Accessed: November 03, 2016. [Online]. Available: http://www.beba-project.eu/

[39] R. Bifulco, J. Boite, M. Bouet, and F. Schneider, "Improving SDN with InSPired Switches," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '16. New York, NY, USA: ACM, 2016, pp. 11:1–11:12. [Online]. Available: http://doi.acm.org/10.1145/2890955.2890962

[40] R. Olsson, "pktgen the linux packet generator," 2005, pp. 1–24. [Online]. Available: https://www.kernel.org/doc/ols/2005/ols2005v2-pages-19-32.pdf

[41] S. Bhattacharya, S. Pratt, B. Pulavarty, and J. Morgan, "Asynchronous I/O Support in Linux 2.5," *Proceedings of the Linux Symposium*, pp. 351–366, Jul. 2003. [Online]. Available: http://www.linuxinsight.com/files/ols2003/pulavarty-reprint.pdf

[42] U. Drepper, "The Need for Asynchronous, Zero-Copy Network I/O," *Proceedings of the Linux Symposium*, vol. 1, pp. 247–260, Jul. 2006. [Online]. Available: https://www.kernel.org/doc/ols/2006/ols2006v1-pages-247-260.pdf

[43] Open vSwitch, "An Open Virtual Switch," Accessed: October 17, 2016. [Online]. Available: http://openvswitch.org

[44] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The Design and Implementation of Open vSwitch," in *Proceedings of the 12th*

USENIX Conference on Networked Systems Design and Implementation, ser. NSDI'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 117–130. [Online]. Available: http://dl.acm.org/citation.cfm?id=2789770.2789779

[45] L. Rizzo and G. Lettieri, "VALE, a Switched Ethernet for Virtual Machines," in *Proceedings of the 8th ACM International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '12. New York, NY, USA: ACM, 2012, pp. 61–72. [Online]. Available: http://doi.acm.org/10.1145/2413176.2413185

[46] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen, "Scalable, High Performance Ethernet Forwarding with CuckooSwitch," in *Proceedings of the 9th ACM International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '13. New York, NY, USA: ACM, 2013, pp. 97–108. [Online]. Available: http://doi.acm.org/10.1145/2535372.2535379

[47] M. Honda, F. Huici, G. Lettieri, and L. Rizzo, "mSwitch: A Highly-scalable, Modular Software Switch," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR '15. New York, NY, USA: ACM, 2015, pp. 1:1–1:13. [Online]. Available: http://doi.acm.org/10.1145/2774993.2775065

[48] J. Kim, S. Huh, K. Jang, K. Park, and S. Moon, "The Power of Batching in the Click Modular Router," in *Proceedings of the Asia-Pacific Workshop on Systems*, ser. APSYS '12. New York, NY, USA: ACM, 2012, pp. 14:1–14:6. [Online]. Available: http://doi.acm.org/10.1145/2349896.2349910

[49] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "ClickOS and the Art of Network Function Virtualization," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 459–473. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616448.2616491

[50] J. Hwang, K. K. Ramakrishnan, and T. Wood, "NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 445–458. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616448.2616490

[51] Xen, "The Xen Project," Accessed: October 17, 2016. [Online]. Available: http://www.xenproject.org/

[52] KVM, "Kernel-based Virtual Machine (KVM)," Accessed: October 17, 2016. [Online]. Available: http://www.linux-kvm.org/page/Main_Page

[53] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. Ramakrishnan, and T. Wood, "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, August 2016. [Online]. Available: http://faculty.cs.gwu.edu/~timwood/papers/16-HotMiddlebox-onvm.pdf

[54] Docker, "Docker Containers," 2016, Accessed: October 17, 2016. [Online]. Available: https://www.docker.com/

[55] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "RouteBricks: Exploiting Parallelism to Scale Software Routers," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 15–28. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629578

[56] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU-accelerated Software Router," in *Proceedings of the ACM SIGCOMM 2010 Conference*, ser. SIGCOMM '10. New York, NY, USA: ACM, 2010, pp. 195–206. [Online]. Available: http://doi.acm.org/10.1145/1851182.1851207

[57] J. Kim, K. Jang, K. Lee, S. Ma, J. Shim, and S. Moon, "NBA (Network Balancing Act): A High-performance Packet Processing Framework for Heterogeneous Processors," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. New York, NY, USA: ACM, 2015, pp. 22:1–22:14. [Online]. Available: http://doi.acm.org/10.1145/2741948.2741969

[58] T. Barbette, C. Soldani, and L. Mathy, "Fast Userspace Packet Processing," in *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 5–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=2772722.2772727

[59] G. Detal, B. Hesmans, O. Bonaventure, Y. Vanaubel, and B. Donnet, "Revealing Middlebox Interference with Tracebox," in *Proceedings of the 2013 Conference on Internet Measurement Conference*, ser. IMC '13. New York, NY, USA: ACM, 2013, pp. 1–8. [Online]. Available: http://doi.acm.org/10.1145/2504730.2504757

[60] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing Network-wide Policies in the Presence of Dynamic Middlebox Actions Using Flowtags," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 533–546. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616448.2616497

[61] P. Kazemian, G. Varghese, and N. McKeown, "Header Space Analysis: Static Checking for Networks," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 9–9. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228311

[62] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "OpenNF: Enabling Innovation in Network Function Control," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, pp. 163–174. [Online]. Available: http://doi.acm.org/10.1145/2619239.2626313

[63] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and Implementation of a Consolidated Middlebox Architecture," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 24–24. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228331

[64] A. Bremler-Barr, Y. Harchol, D. Hay, and Y. Koral, "Deep Packet Inspection as a Service," in *Proceedings of the 10th ACM International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '14. New York, NY, USA: ACM, 2014, pp. 271–282. [Online]. Available: http://doi.acm.org/10.1145/2674005.2674984

[65] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat, "xOMB: Extensible Open Middleboxes with Commodity Servers," in *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '12. New York, NY, USA: ACM, 2012, pp. 49–60. [Online]. Available: http://doi.acm.org/10.1145/2396556.2396566

[66] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: A Framework for NFV Applications," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. New York, NY, USA: ACM, 2015, pp. 121–136. [Online]. Available: http://doi.acm.org/10.1145/2815400.2815423

[67] Apache, "Hadoop," Accessed: October 17, 2016. [Online]. Available: http://hadoop.apache.org/

[68] A. Bremler-Barr, Y. Harchol, and D. Hay, "OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions," in *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 511–524. [Online]. Available: http://doi.acm.org/10.1145/2934872.2934875

[69] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown, "Programmable Packet Scheduling at Line Rate," in *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 44–57. [Online]. Available: http://doi.acm.org/10.1145/2934872.2934899

[70] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker, "Universal Packet Scheduling," in *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, ser. NSDI'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 501–521. [Online]. Available: http://dl.acm.org/citation.cfm?id=2930611.2930644

[71] L. Rizzo, P. Valente, G. Lettieri, and V. Maffione, "A Fast and Practical Software Packet Scheduling Architecture," Tech. Rep., May 2016. [Online]. Available: http://info.iet.unipi.it/~luigi/papers/20160511-mysched-preprint.pdf

[72] L. McVoy and C. Staelin, "lmbench: Portable Tools for Performance Analysis," in *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '96. Berkeley, CA, USA: USENIX Association, 1996, pp. 23–23. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268299.1268322

[73] Intel, "Memory Latency Checker (MLC) v3.1a," 2013, Accessed: July 5, 2016. [Online]. Available: https://software.intel.com/en-us/articles/intelr-memory-latency-checker

[74] J. Levon, "OProfile: A System Profiler for Linux," July 2016, Accessed: October 17, 2016. [Online]. Available: http://oprofile.sourceforge.net/news/

[75] Perf, "Linux profiling with performance counters," Accessed: October 17, 2016. [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

[76] A. R. Lebeck and D. A. Wood, "Cache Profiling and the SPEC Benchmarks: A Case Study," *Computer*, vol. Volume 27, Number 10, no. 10, pp. 15–26, Oct. 1994. [Online]. Available: http://dx.doi.org/10.1109/2.318580

[77] KCachegrind, "KCachegrind profiler," Accessed: October 17, 2016. [Online]. Available: http://kcachegrind.sourceforge.net/html/Home.html

[78] Intel, "Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors," Accessed: October 17, 2016. [Online]. Available: https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf

[79] likwid, "Performance monitoring and benchmarking suite," July 2016, Accessed: October 17, 2016. [Online]. Available: https://github.com/RRZE-HPC/likwid

[80] A. Pesterev, N. Zeldovich, and R. T. Morris, "Locating Cache Performance Bottlenecks Using Data Profiling," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. New York, NY, USA: ACM, 2010, pp. 335–348. [Online]. Available: http://doi.acm.org/10.1145/1755913.1755947

[81] Intel, "Xeon E5 2667 v3 processor," Accessed: October 17, 2016. [Online]. Available: http://ark.intel.com/products/83361/Intel-Xeon-Processor-E5-2667-v3-20M-Cache-3_20-GHz

[82] Network Computing, "100 Gbps Headed For The Data Center," Nov. 2014. [Online]. Available: http://www.networkcomputing.com/data-centers/100-gbps-headed-data-center/407619707

[83] The VAR Guy Blog, "The Road to 25/50/100 Gigabit Ethernet," May 2015. [Online]. Available: http://thevarguy.com/blog/road-2550100-gigabit-ethernet

[84] Cisco, "Migrate to a 40-Gbps Data Center with Cisco QSFP BiDi Technology." [Online]. Available: http://www.cisco.com/c/en/us/products/collateral/switches/nexus-9000-series-switches/white-paper-c11-729493.html

[85] A. Viejo, "QLogic and Broadcom First to Demonstrate End-to-End Interoperability for 25Gb and 100Gb Ethernet," January 27, 2015. [Online]. Available: https://globenewswire.com/news-release/2015/01/27/700249/10116850/en/QLogic-and-Broadcom-First-to-Demonstrate-End-to-End-Interoperability-for-25Gb-and-100Gb-Ethernet.html

[86] Amazon, "Compute and Networking Services for Amazon Web Services," Accessed: October 17, 2016. [Online]. Available: http://docs.aws.amazon.com/gettingstarted/latest/awsgsg-intro/gsg-aws-compute-network.html

[87] ——, "EC2 Container Service (ECS)," Accessed: October 17, 2016. [Online]. Available: https://aws.amazon.com/ecs/

[88] H. Benn, "Vision and Key Features for 5th Generation (5G) Cellular," Jan. 2014. [Online]. Available: http://cambridgewireless.co.uk/Presentation/RadioTech_30.01.14_HowardBenn.Samsung.pdf

[89] Ericsson, "5G Radio Access Technology and Capabilities," Apr. 2016. [Online]. Available: http://www.ericsson.com/res/docs/whitepapers/wp-5g.pdf

[90] Nokia Networks, "5G use cases and requirements," Jul. 2014. [Online]. Available: http://resources.alcatel-lucent.com/asset/200010

[91] T. K. Brown, "Socratic Method and Critical Philosophy, Selected Essays by Leonard Nelson," *Philosophy and Phenomenological Research*, vol. 11, no. 2, pp. 283–285, 1950.

[92] L. Nelson and T. K. Brown, *Socratic Method and Critical Philosophy Selected Essays*. Yale Univ. Press, 1949.

[93] J. W. Creswell, *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. SAGE Publications, 2003. [Online]. Available: https://books.google.ie/books?id=nSVxmN2KWeYC

[94] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "MoonGen: A Scriptable High-Speed Packet Generator," in *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*, ser. IMC '15. New York, NY, USA: ACM, 2015, pp. 275–287. [Online]. Available: http://doi.acm.org/10.1145/2815675.2815692

[95] G. P. Katsikas, G. Q. Maguire Jr., and D. Kostić, "Profiling and accelerating commodity NFV service chains with SCC," *Journal of Systems and Software*, vol. 127C, pp. 12–27, Feb. 2017. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2017.01.005

[96] Amazon, "Enhanced Networking with SR-IOV," Accessed: October 17, 2016. [Online]. Available: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/enhanced-networking.html

[97] G. P. Katsikas, "FastClick user-space I/O multiplexing using standard Linux network drivers." 2016, Accessed: October 17, 2016. [Online]. Available: https://github.com/gkatsikas/fastclick/tree/mmap

[98] ——, "System benchmarks," 2016, Accessed: October 17, 2016. [Online]. Available: https://github.com/gkatsikas/system-bench

[99] C. S. Pabla, "Completely Fair Scheduler," *Linux Journal*, vol. 2009, no. Volume 2009, Issue 184, Aug. 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1594371.1594375

[100] Intel, "Hardware events for Intel Haswell processors," Accessed: October 17, 2016. [Online]. Available: https://software.intel.com/en-us/node/589935

[101] G. P. Katsikas, M. Enguehard, M. Kuźniar, G. Q. Maguire Jr., and D. Kostić, "SNF: Synthesizing high performance NFV service chains," *PeerJ Computer Science*, vol. 2, p. e98, Nov. 2016. [Online]. Available: http://dx.doi.org/10.7717/peerj-cs.98

[102] Intel, "High Precision Event Timers (HPET)," Accessed: October 17, 2016. [Online]. Available: http://www.intel.se/content/dam/www/public/us/en/documents/technical-specifications/software-developers-hpet-spec-1-0a.pdf

[103] B. Sigoure, "Custom tools for measuring the cost of context switching," July 2010, posted at 11/14/2010 08:53:00 PM, http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html.

[104] L. Deri, "Direct NIC Access with PF_RING," 2011, Accessed: October 17, 2016. [Online]. Available: http://www.ntop.org/pf_ring/

[105] M. Enguehard, "Hyper-NF: synthesizing chains of virtualized network functions," Master's thesis, KTH Royal Institute of Technology, School of Information and Communication Technology, Kista, Sweden, Jan. 2016, TRITA-ICT-EX, 2016:2. [Online]. Available: http://urn.kb.se/resolve?urn=urn%3Anbn%3Ase%3Akth%3Adiva-180397

[106] D. E. Taylor and J. S. Turner, "ClassBench: A Packet Classification Benchmark," *IEEE/ACM Trans. Netw.*, vol. 15, no. 3, pp. 499–511, Jun. 2007. [Online]. Available: http://dx.doi.org/10.1109/TNET.2007.893156

[107] Intel, "Receive-Side Scaling (RSS)," 2016, Accessed: October 17, 2016. [Online]. Available: http://www.intel.com/content/dam/support/us/en/documents/network/sb/318483001us2.pdf

[108] S. Woo and K. Park, "Scalable TCP Session Monitoring with Symmetric Receive-side Scaling. KAIST Technical Report," 2012, pp. 1–7. [Online]. Available: http://www.ndsl.kaist.edu/~kyoungsoo/papers/TR-symRSS.pdf

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

[110] W. Sun and R. Ricci, "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*, ser. ANCS '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 25–36. [Online]. Available: http://dl.acm.org/citation.cfm?id=2537857.2537861

[111] G. P. Katsikas, "SNF extensions of FastClick's stateful flow processing elements." 2016, Accessed: October 17, 2016. [Online]. Available: https://github.com/gkatsikas/fastclick/tree/snf

[112] Cisco, "Scaling NFV - The Performance Challenge," 2014. [Online]. Available: http://blogs.cisco.com/enterprise/scaling-nfv-the-performance-challenge

[113] SDX Central, "Performance - Still Fueling the NFV Discussion," 2015. [Online]. Available: https://www.sdxcentral.com/articles/contributed/vnf-performance-fueling-nfv-discussion-kelly-leblanc/2015/05/

[114] M. Bagnulo, P. Matthews, and I. van Beijnum, "Stateful NAT64: Network Address and Protocol Translation from IPv6 Clients to IPv4 Servers," Internet Request for Comments (RFC) 6146 (Proposed Standard), Internet Engineering Task Force, Apr. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6146.txt

[115] S. Perreault, I. Yamagata, S. Miyakawa, A. Nakagawa, and H. Ashida, "Common Requirements for Carrier-Grade NATs (CGNs)," Internet Request for Comments (RFC) 6888 (Best Current Practice), Internet Engineering Task Force, Apr. 2013. [Online]. Available: http://www.ietf.org/rfc/rfc6888.txt

[116] R. Penno, D. Wing, and M. Boucadair, "PCP Support for Nested NAT Environments," Internet Request for Comments (RFC) Working Draft, IETF Secretariat, Internet-Draft draft-penno-pcp-nested-nat-03, January 2013, Expired on July 25, 2013. [Online]. Available: https://tools.ietf.org/html/draft-penno-pcp-nested-nat-03

[117] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A NICE Way to Test Openflow Applications," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228312

[118] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark, "Kinetic: Verifiable Dynamic Network Control," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 59–72. [Online]. Available: http://dl.acm.org/citation.cfm?id=2789770.2789775

[119] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostić, "A SOFT Way for Openflow Switch Interoperability Testing," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '12. New York, NY, USA: ACM, 2012, pp. 265–276. [Online]. Available: http://doi.acm.org/10.1145/2413176.2413207

[120] M. Dobrescu and K. Argyraki, "Software Dataplane Verification," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 101–114. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616448.2616459

[121] World Commission on Environment and Development, *Our Common Future*. Oxford University Press, 1987. [Online]. Available: http://EconPapers.repec.org/RePEc:oxp:obooks:9780192820808

[122] Intel, "64 and IA-32 Architectures Developer's Manual," 2016, Accessed: October 17, 2016. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf

[123] Linux, "Perf events' for Intel chipsets in Linux kernel sources," 2002, Accessed: October 17, 2016. [Online]. Available: https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/arch/x86/kernel/cpu/perf_event_intel.c?id=dea4f48a0a301b23c65af8e4fe8ccf360c272fbf

[124] Intel, "Ethernet Flow Director," 2016, Accessed: October 17, 2016. [Online]. Available: http://www.intel.com/content/www/us/en/ethernet-controllers/ethernet-flow-director-video.html

[125] Linux ethtool, "Display or change Ethernet card settings," Accessed: October 17, 2016. [Online]. Available: http://www.linuxcommand.org/man_pages/ethtool8.html

[126] Linux, "Receive Offload," Accessed: October 17, 2016. [Online]. Available: https://lwn.net/Articles/358910/

Appendix A

Appendix

A.1 Collecting Performance Counters

This appendix discusses how Perf [75] can be instructed to collect useful performance counters to profile NFV systems. Perf is a profiling tool for OSs. It is capable of catching several events, such as generalized hardware, software, tracepoint, and cache events. Each of these events has a different type and their source can be either the processor's PMU or the kernel. Since the underlying hardware plays a major role in the number and functionality of these events, Perf uses symbolic names to define these events in a hardware-agnostic way.

We consulted Perf's man pages [122] for event descriptions and realized that these descriptions are usually ambiguous and these pages suggest referring to the CPU manuals of the specific processor for additional information. Therefore, in order to obtain accurate information about the exact nature of each event, one needs to map the symbolic names exposed by Perf to the events as documented by the CPU vendor's processor specification document. In our case, the "Intel® 64 and IA-32 Architectures Developer's Manual" [122] provides very specific information that in conjunction with the processor's event types in the Linux kernel sources [123] reveal useful insights about the available performance counters of our chip.

In the following subsections, we summarize this information by using tables to map Perf's symbolic names to their descriptions for each type of event.

A.1.1 Generalized Hardware Events

In a Linux-based OS, if Perf's event type is PERF_TYPE_HARDWARE, we are measuring one of the generalized hardware CPU events. Table A.1 shows the available generalized hardware counters of our machine as reported by Perf along with a description based upon Intel's documentation.

Note that in this table the term "instructions at retirement" means the actual instructions of the target program flow (not the speculative instructions fetched by the CPU). The "last level cache" for this processor is the L3 cache. A "micro-op" corresponds to a small, basic instruction that performs operations on data stored in one or more registers. A "mispredicted branch instruction" corresponds to an unsuccessfully "guessed" instruction inserted into the pipeline by the branch predictor.

Table A.1: Mapping between Perf's generalized hardware events and Intel's descriptions.

Perf's Event ID	**Description**
PERF_COUNT_HW_CPU _CYCLES	Number of core clock cycles when the clock signal on a specific core is running (i.e., this CPU is not halted).
PERF_COUNT_HW _INSTRUCTIONS	Number of instructions at retirement. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. Faults before the retirement of the last micro-op of a multi-op instruction are not counted.
PERF_COUNT_HW _CACHE_REFERENCES	Number of requests originating from the core that reference a cache line in the LLC.
PERF_COUNT_HW _CACHE_MISSES	Counts each cache miss condition for references to the LLC.
PERF_COUNT_HW _BRANCH _INSTRUCTIONS	Counts branch instructions at retirement. This includes the retirement of the last micro-op of a branch instruction.
PERF_COUNT_HW _BRANCH_MISSES	Counts mispredicted branch instructions at retirement. This includes the retirement of the last micro-op of a branch instruction in the architectural path of execution and experienced misprediction by the branch prediction hardware.

A.1.2 Hardware CPU Cache Events

Next, the event type responsible for cache-related hardware events in Perf is PERF_TYPE_HW_CACHE. As shown in Table A.2, in our machine Perf reports five cache events, each with a different cache identifier: L1 data, L1 instruction, LLC, DTLB, and instruction TLB cache events. To calculate a value for a particular cache operation (i.e., read, write, or prefetch) one needs a bitwise OR operation between the operation and cache identifier. Then, to obtain the result (i.e., hit or miss) of any operation, a similar operation is required between the previous result and the result identifier. More detailed information is available in Perf's man pages [122].

Table A.2: Description of Perf's hardware CPU and cache events.

Perf's Event ID	Operation	Result	Description
PERF_COUNT _HW_CACHE _L1D	LOAD	HIT	Successful read accesses to L1 data cache.
		MISS	Missed read accesses to L1 data cache.
	STORE	HIT	Successful write accesses to L1 data cache.
		MISS	Missed write accesses to L1 data cache.
	PREFETCH	HIT	Not available.
		MISS	Missed prefetch accesses to L1 data cache.
PERF_COUNT _HW_CACHE _L1I	LOAD	MISS	Missed read accesses to L1 instruction cache.
PERF_COUNT _HW_CACHE _LL	LOAD	HIT	Successful read accesses to LLC.
	STORE		Successful write accesses to LLC.
	PREFETCH		Successful prefetch accesses to LLC.
PERF_COUNT _HW_CACHE _DTLB	LOAD	HIT	Successful read accesses to DTLB.
		MISS	Missed read accesses to DTLB.
	STORE	HIT	Successful write accesses to DTLB.
		MISS	Missed write accesses to DTLB.
PERF_COUNT _HW_CACHE _ITLB	LOAD	HIT	Successful read accesses to instruction TLB.
		MISS	Missed read accesses to instruction TLB.
PERF_COUNT _HW_CACHE _BPU	LOAD	HIT	Successful read accesses of the branch prediction unit.
		MISS	Missed read accesses of the branch prediction unit.

A.1.3 Software Events from the Linux Kernel

Finally, Perf exposes a kernel-level API from which we can retrieve the software events listed in Table A.3.

Table A.3: Mapping between Perf's software events and descriptions.

Perf's Event ID	Description
PERF_COUNT_SW _CPU_CLOCK	A high-resolution per CPU timer.
PERF_COUNT_SW _TASK_CLOCK	A clock count specific to the task that is running.
PERF_COUNT_SW _CONTEXT_SWITCHES	Number of context switches as happening in the kernel.
PERF_COUNT_SW _CPU_MIGRATIONS	Number of times the task has been migrated to different CPU.
PERF_COUNT_SW _PAGE_FAULTS	Number of page faults.
PERF_COUNT_SW _PAGE_FAULTS_MIN	Number of minor page faults. These did not require disk IO to handle.
PERF_COUNT_SW _PAGE_FAULTS_MAJ	Number of major page faults. These did not require disk IO to handle.
PERF_COUNT_SW _ALIGNMENT_FAULTS	Number of alignment faults. These happen in 64-bit systems when unaligned memory accesses happen.
MEM_LOADS	Number of main memory load operations (event=0xcd,umask=0x1,ldlat=3).
MEM_STORES	Number of main memory store operations (event=0xd0,umask=0x82).
CONSUME_SKB	Number of socket buffers (skbuffs) consumed.
SYS_ENTER_SENDTO	Number of *sendto* system calls.
SYS_ENTER_RECVFROM	Number of *recvfrom* system calls.
SYS_ENTER_POLL	Number of *poll* system calls.
SYS_ENTER_MMAP	Number of *mmap* system calls.
SCHED_STAT_RUNTIME	Time the task is executing on a CPU.
SCHED_STAT_SLEEPTIME	Time the task is not runnable, including IO waiting time.
SCHED_STAT_WAITTIME	Time the task is runnable but not actually running due to scheduler contention.
SCHED_STAT_IO_WAIT	Time the task is not runnable, including IO waiting time.
SCHED_STAT_BLOCKED	Time the task is in uninterruptible state.

A.2 Testbed Configuration

This appendix discusses the low-level configuration of the testbed used to conduct the experiments for this licentiate thesis. The components of the testbed are described in Chapter 5. The discussion here focuses on the underlying hardware. In particular, CPU-related issues are discussed in appendix A.2.1, while appendix A.2.2 tackles NIC-specific configuration.

A.2.1 CPU Pinning and Isolation

In our setup, we isolated an entire CPU socket (i.e., 8 cores) on both machines 1 and 2, using the kernel isolation parameter specified in § 2.6. To ensure that the pinning works correctly, during the execution of an NF we inspect a software-based performance counter reported by the Linux kernel using Perf. Specifically, the counter PERF_COUNT_SW_CPU_MIGRATIONS is monitored (see Table A.3). The value of this counter must be zero as an NF must always be pinned to one CPU core, hence no migrations should happen.

A.2.2 NIC Configuration

The following sections discuss: how one can modify the number of allocated buffer descriptors (see appendix A.2.2.1). how one can exploit the multiple hardware queues of modern NICs and the CPU cores of the testbed to parallelize packet processing (see appendix A.2.2.2), how to configure the NIC and OS to achieve better performance (see appendix A.2.2.4 and appendix A.2.2.3).

A.2.2.1 NIC Buffer Descriptors

Our NICs can accommodate up to 4096 Tx and 4096 Rx buffer descriptors in their local memory, although the default values for both Tx and Rx ring buffer sizes set by ixgbe are 512 descriptors. Having more buffer descriptors available could be beneficial in cases that the incoming packet rate is very high. For example, when we use netmap as a packet I/O mechanism, we set the number of Tx descriptors to 2048 (maximum number currently supported by netmap). However, one must be careful when manually allocating buffer descriptors because a large number of theses descriptors might not fit into the system's caches, hence the NIC will be forced to involve main memory more frequently, thus increasing the per packet latency for some packets.

We performed some experiments with NFV applications using the standard, unmodified ixgbe network driver; these experiments indicated that 256 Rx and 1024 Tx buffer descriptors provide the best results in terms of throughput and latency. With the DPDK network driver, the respective number of both Tx and Rx descriptors is 1024.

A.2.2.2 NIC Hardware Queues

Each NIC in our testbed has 128 hardware queues. By default, 16 of these queues are used as this is the number of CPU cores per machine. We can exploit these queues to dispatch incoming flows to different hardware queues and pin the available cores to these queues using either Intel's Flow Director [124] or RSS [107]. With Flow Director, we can issue *ethtool* [125] commands to the NIC, to classify incoming frames and redirect the matched frames to a particular queue. Using RSS, we can achieve similar functionality by using one of the hash functions implemented by the firmware to classify incoming frames as we wish. Both techniques facilitate parallel packet processing.

After some tests with NFV applications based on ixgbe, we found that the optimal number of Tx and Rx queues for this driver is 1 and 16 respectively. In the case of the DPDK driver, the maximum number of queues (i.e., 128) is used both for Tx and Rx.

A.2.2.3 Interrupts and Polling

For the experiments that use the unmodified ixgbe network driver, we used interrupts as this is the default setup of this driver. For the experiments that use the DPDK network driver, we achieve high performance I/O by constantly polling the NICs at the cost of underutilizing the cores involved in the polling process.

A.2.2.4 NIC Offloading Features

Most networking hardware vendors nowadays implement a portion of the network stack, traditionally done by the OS, in the NIC. Using system tools such as *ethtool*, one can retrieve the supported features for a NIC. Our 10 GbE Intel 82599 ES NICs support the offloading features shown in Table A.4, where we briefly describe the purpose of each feature. The features listed in this table might affect an NFV application either positively or negatively.

Positive effects occur when the offloaded functions save CPU cycles for applications running on the system. For example, imagine a traffic generator that creates IP packets and has to calculate the checksum field of the IP header for each packet. This operation consumes more CPU cycles in software compared to the checksum calculation function implemented in a NIC. One can use the *ethtool* feature "tx-checksumming" described in Table A.4 to offload the checksum calculation to the NIC.

Table A.4: Ethtool offloading features supported by Intel 82599 ES NICs.

Feature Name (*ethtool*)	**Short Description**
tx-checksumming	Calculate the checksum of transmitted frames for IPv4, IPv6 and Stream Control Transmission Protocol.
rx-checksumming	Calculate the checksum of received frames.
scatter-gather	Reads/Writes frames into/from multiple buffers.
tcp-segmentation-offload	TCP Segmentation. Linux kernel calculates the receive window of the client, the send window for this connection and then pushes as much data as possible to the NIC as permitted by these restrictions.
generic-segmentation-offload	Generalization of TCP Segmentation. It covers more protocols, such as UDP and Datagram Congestion Control Protocol.
large-receive-offload	Incoming frames are merged at reception time so that the OS sees far fewer of them [126].
generic-receive-offload	A stricter version of large receive offload. The criteria for which frames can be merged are greatly restricted; the MAC headers must be identical and only a few TCP or IP headers can differ. As a result of these restrictions, merged frames can be re-segmented losslessly.
udp-fragmentation-offload	IP fragmentation functionality of large UDP datagrams.
tx-vlan-offload	Virtual local area network tagging for transmitted frames.
rx-vlan-offload	Virtual local area network tagging for received frames.
ntuple-filters	Distributes frames to hardware queues by applying header space filtering as per Intel's Flow Director [124].
receive-hashing	Distributes frames to hardware queues by applying a hash function on a header space as per Intel's RSS [107].
rx-vlan-filter	Filtering of ingress virtual local area network traffic.
tx-nocache-copy	Allow no-cache copy from user on transmission.
rx-all	Do not drop received frames with incorrect frame checksum sequences.
l2-fwd-offload	L2 forwarding.

Negative effects can be caused by some features. If a large set of these functions is enabled, it might reduce the performance of some NICs, when exchanging packets at line-rate. Secondly, when the application attached to the NIC needs to perform middlebox-specific operations (as per the NFV case), some features might obscure critical information from that application and affect the way the application reacts to the packets. For instance, if the large-receive-offload feature is enabled, the NIC merges batches of frames to fewer but longer frames. This means that if there are important differences between the headers in the incoming frames, those differences will be lost. However, some NFV applications might employ decision elements (e.g., routing based on destination IP addresses) to adapt the forwarding to these header changes, hence large-receive-offload will break this forwarding.

For these reasons, it is important to co-design hardware and software in NFV such that performance and correctness are not affected. After performing measurements with and without the offloading features listed in Table A.4, we decided that it is beneficial to follow the configuration specified in Table A.5.

Table A.5: Selected states for the offloading features of an Intel 82599 ES NIC. The default values are based on the Linux-based ixgbe network driver version 3.19.1.

Feature Name (*ethtool*)	**Default State**	**Selected State**
tx-checksumming	On	Off
rx-checksumming	On	Off
scatter-gather	On	On
tcp-segmentation-offload	On	On
generic-segmentation-offload	On	On
large-receive-offload	On	On
generic-receive-offload	On	Off
udp-fragmentation-offload	Off	Off
tx-vlan-offload	On	On
rx-vlan-offload	On	Off
ntuple-filters	Off	Off
receive-hashing	On	On
rx-vlan-filter	On	On
tx-nocache-copy	Off	Off
rx-all	Off	Off
l2-fwd-offload	Off	Off