NFV Service Chains at the Speed of the Underlying Commodity Hardware

GEORGIOS P. KATSIKAS

Doctoral Thesis in Information and Communication Technology
School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology
Stockholm, Sweden 2018

KTH
School of Electrical Engineering
and Computer Science
SE-164 40 Kista
SWEDEN

TRITA-EECS-AVL-2018:50
ISBN 978-91-7729-863-2

Akademisk avhandling som med tillstånd av Kungliga Tekniska högskolan framlägges till offentlig granskning för avläggande av doktorsexamen i informations och kommunikationsteknik Fredag den 14 September 2018 klockan 15:00 i Sal C, Electrum, Kungliga Tekniska högskolan, Kistagången 16, Kista.

Tryck: Universitetsservice US AB

Abstract

Link speeds in networks will in the near-future reach and exceed 100 Gbps. While available specialized hardware can accommodate these speeds, modern networks have adopted a new networking paradigm, also known as Network Functions Virtualization (NFV), that replaces expensive specialized hardware with open-source software running on commodity hardware. However, achieving high performance using commodity hardware is a hard problem mainly because of the processor-memory gap. This gap suggests that only the fastest memories of today's commodity servers can achieve the desirable access latencies for high speed networks. Existing NFV systems realize chained network functions (also known as service chains) mostly using slower memories; this implies a need for multiple additional CPU cores or even multiple servers to achieve high speed packet processing. In contrast, this thesis combines four contributions to realize NFV service chains with dramatically higher performance and better efficiency than the state of the art.

The first contribution is a framework that profiles NFV service chains to uncover reasons for performance degradation, while the second contribution leverages the profiler's data to accelerate these service chains by combining multiplexing of system calls with scheduling strategies. The third contribution synthesizes input/output and processing service chain operations to increase the spatial locality of network traffic with respect to a system's caches. The fourth contribution combines the profiler's insights from the first contribution and the synthesis approach of the third contribution to realize NFV service chains at the speed of the underlying commodity hardware. To do so, stateless traffic classification operations are offloaded into available hardware (i.e., programmable switches and/or network cards) and a tag is associated with each traffic class. At the server side, input traffic classes are classified by the hardware based upon the values of these tags, which indicate the CPU core that should undertake their stateful processing, while ensuring zero inter-core communication.

With commodity hardware, this thesis realizes Internet Service Provider-level service chains and deep packet inspection at a line-rate 40 Gbps and stateful service chains at the speed of a 100 GbE network card on a 16 core single server. This results in up to (*i*) 4.7x lower latency, (*ii*) 8.5x higher throughput, and (*iii*) 6.5x better efficiency than the state of the art. The techniques described in this thesis are crucial for realizing future high speed NFV deployments.

Keywords: NFV, service chains, synthesis, offloading, tagging, zero inter-core communication, line-rate, 100 GbE

Sammanfattning

Länkhastigheter i nätverk kommer inom en snar framtid att nå och överstiga 100 Gbps. Medan existerande specialiserad hårdvara numera kan tillgodose dessa hastigheter, tillämpas i moderna nätverk även ett nytt nätverksparadigm känt som funktionsvirtualisering av nätverk (NFV), som ersätter dyr specialiserad hårdvara med öppen källkodsprogramvara som körs på kostnadseffektiv, icke-specialiserad, vanlig dator (s.k. råvaruenheter). Att uppnå hög prestanda med hjälp av standardmaskinvara är ett svårt problem, huvudsakligen på grund av prestandaskillnader mellan processor och minne. Denna skillnad medför att endast de snabbaste cache-minnena av idag måste användas för att uppnå högsta prestanda med minsta möjliga fördröjningar i höghastighetsnätverk. Sammankopplade nätverksfunktioner (s.k. tjänstekedjor) i existerande NFV-system använder mestadels långsammare minne, vilket innebär att ytterligare CPU-kärnor eller servrar behövs för att uppnå motsvarande höghastighetsprestanda vid hanteringen av datapaket. I denna avhandling kombineras fyra bidrag som möjliggör NFV-tjänstekedjor med betydligt högre prestanda och effektivitet jämfört med den senaste tekniken.

Det första bidraget är ett ramverk som profilerar NFV-tjänstekedjor för att identifiera av orsaken till prestandaförsämringar, medan det andra bidraget utnyttjar profildata för att snabba upp tjänstekedjorna genom att kombinera multiplexering av systemanrop med olika schemaläggningsstrategier. Det tredje bidraget syntetiserar indata/utdata och tjänstekedjeoperationer för att öka den spatiala lokaliteten av nätverkstrafiken i förhållande till systemets cacher. Det fjärde bidraget kombinerar profileringsresultat från det första bidraget och syntetiseringsmetoden från det tredje bidraget för att möjliggöra NFV-tjänstekedjor kapabla att hantera datatrafik med samma höga överföringshastighet som den underliggande maskinvaran. För att göra detta överförs tillståndslösa trafikklassificeringsoperationer till tillgänglig maskinvara (d.v.s. programmerbara switchar och/eller nätverkskort) med en indikativ märkning kopplad till varje trafikklass. På serverns sida klassificeras inkomna trafikklasser baserad på märkningen, följt av tillståndsstyrd bearbetning av paketen i tillgängliga CPU-kärnor utan inbördes kommunikation mellan kärnorna.

Med användning av endast vanlig maskinvara uppnås i den här avhandlingen tjänstekedjor på Internet-leverantörsnivå och djupa paketinspektioner vid en hastighet av 40 Gbps, motsvarande den underliggande linjehastighet bearbetning, samt tillståndsstyrda tjänstekedjor med hastigheten motsvarande ett 100 GbE-nätverkskort på en server. Detta resulterar i upp till (*i*) 4,7x lägre latens, (*ii*) 8,5x högre dataöverföring och (*iii*) 6,5x ökad effektivitet jämfört med den senaste tekniken. Denna avhandling är avgörande för att förverkliga framtida höghastighetsnätverk.

Nyckelord: NFV, tjänstekedjor, syntetisering, processavlastning, paketmärkning, ingen inter-CPU-kommunikation, länkkapacitet, 100 GbE

Acknowledgements

I am first and foremost thankful to my advisors and mentors, Professors Dejan Kostić and Gerald Q. Maguire Jr as well as Dr. Rebecca Steinert. 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 implement the first working prototype of SNF. Maciej Kuzniar also played 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 its limit. Tom Barbette was my mate while implementing and evaluating Metron. Thanks to his devotion and great coding skills we managed to exploit every single bit of the available hardware. With this work a dream came true for me: A place at NSDI, the "pantheon" of networked systems.

Associate Professor Markus Hidell was the internal reviewer of both my licentiate and doctoral proposals and the advance reviewer of both theses. I would like to thank him very much for his patience and excellent & timely feedback.

Kirill Bogdanov is my colleague and the person who shared the same office, whiteboard, ambitions, Endomondo challenges, and chocolates with me. Thanks to Kirill Bogdanov, Voravit Tanyingyong, Waleed Reda, Amir Roozbeh, and Alireza Farshin I tried out lots of restaurants and pubs in Stockholm.

Last but not least, my family and friends deserve a special mention. Their unconditional support over these years has been crucial. My uncle Christos has been an exemplar of lifelong learning and his role has been central to my personal development. Zoe has always been by my side, especially during the toughest period of this journey. My parents, my brother Labros, and my best friend Panayotis have also been a source of optimism and joy. Without all these people, this thesis would not have ever been written.

'Γι΄ αυτό αξίζει να ορμάς,
στις μάχες να σφαδάζεις,
ώσπου να 'ρθεί η γλυκιά στιγμή,
τ΄ άστρο που άναψες εσύ,
γαλήνιος να κοιτάζεις.'

— Γιάννης Αγγελάκας

Georgios P. Katsikas,
Stockholm, May 28, 2018

The research leading to the licentiate part of this thesis was co-funded by the European Union (EU) in the context of the European Research Council (ERC) under EU's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement 259110 and the BEhavioural BAsed forwarding (BEBA) project with grant agreement number 644122. The doctoral phase of this thesis has been funded by the Swedish Foundation for Strategic Research with grant agreement number RIT15-0075 and partially by the Wallenberg AI, Autonomous Systems, and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

Contents

List of Figures

List of Tables

List of Acronyms

5G	Fifth Generation. 1, 56, 186
ACL	Access Control List. 112, 127, 131, 134, 135, 137, 139, 141, 143, 148, 165, 174
API	Application Programming Interface. 4, 18, 21, 23, 35, 37–39, 53, 148, 161, 180
ASIC	Application-Specific Integrated Circuit. 36
B2B	Back-to-Back. 45, 46, 72, 73, 104, 106, 107, 184
CFS	Completely Fair Scheduler. 75, 96, 97, 102–106, 109
CPU	Central Processing Unit. 8, 11, 17, 18, 20, 21, 24, 26–29, 35–37, 43, 45, 47, 50, 53, 56, 57, 62, 63, 71, 74, 76, 77, 79, 81, 84, 85, 88–90, 92–94, 96–99, 103, 104, 106–108, 111, 112, 123, 125, 126, 128, 130–132, 143, 149, 152–154, 156, 157, 159, 160, 163, 165–169, 171, 172, 174, 175, 181, 184–186, 194, 197
CRC	Cyclic Redundant Check. 37, 45, 81, 128
DAG	Directed Acyclic Graph. 65, 66, 114, 116–119, 122, 125, 149, 189
DDIO	Direct Data I/O. 27, 76, 80, 84
DMA	Direct Memory Access. 17–20, 26, 27, 84, 192

DPDK Data Plane Development Kit. 10, 22, 23, 29, 35–37, 46–48, 53, 55, 63, 73, 81–85, 87, 101, 126, 134, 139, 146, 161, 164, 169, 176, 180, 181, 183, 184, 190

DPI Deep Packet Inspection. 6, 35, 37, 38, 52, 58, 66, 147, 161, 165, 167, 172, 186

DRAM Dynamic Random Access Memory. 53, 54, 57, 62, 79, 80, 178

DTLB Data Translation Lookaside Buffer. 76, 79–81, 83, 84

ETSI European Telecommunications Standards Institute. 39

FIFO First In, First Out. 75, 97, 98

FPGA Field-Programmable Gate Array. 36, 39, 180

GbE Gigabit Ethernet. 11, 43, 44, 50, 58, 62, 63, 81, 128, 152, 169, 186, 192, 197

GPU Graphics Processing Unit. 36, 37, 147, 149, 180

HSA Header Space Analysis. 38, 127, 145

HTTP Hypertext Transfer Protocol. 6, 160, 162, 163

I/O Input/Output. 9, 10, 21–23, 27–29, 34–37, 46–48, 53, 55–57, 65, 66, 70–75, 81, 83, 84, 86, 89–93, 97–100, 102–107, 109, 114, 118, 122, 123, 125, 126, 132, 134, 146, 149, 183–185, 190, 194, 195, 197

IDS Intrusion Detection System. 3, 163

IETF Internet Engineering Task Force. 2, 3

IP Internet Protocol. 2, 3, 6, 21, 32, 37, 66, 114, 116, 121, 123–127, 129, 131, 141, 148, 169, 180

IPv4 Internet Protocol version 4. 3, 115, 120, 129, 154, 178

IPv6 Internet Protocol version 6. 129, 154, 178

ISP Internet Service Provider. 2, 6, 10, 57, 112, 127, 129, 131, 132, 135, 138, 141, 143, 145, 148, 174, 175, 184–186, 194, 195, 197

JSON JavaScript Object Notation. 91, 92

LB Load Balancer. 3, 6, 34, 50, 66, 122, 169, 172, 190

LLC Last Level Cache. 26–28, 53, 54, 57, 76, 78, 80, 84, 85, 87, 108

MAC Medium Access Control. 17, 37, 66, 116, 164, 191

Mpps Millions of Packets Per Second. 28, 45, 81, 99, 134

NAPT Network Address and Port Translator. 3, 6, 50, 66, 112, 116, 121, 122, 127, 129–131, 135, 137, 141, 169, 172, 174, 184, 190

NAT Network Address Translator. 129, 145, 160

NF Network Function. 2–6, 8, 10, 28, 31, 32, 35, 36, 38–40, 45, 46, 48, 50, 52, 53, 55–58, 61, 65, 66, 69, 70, 72, 74, 79–82, 84, 85, 88, 89, 91–94, 99, 102–107, 109, 111, 112, 114–123, 125, 128, 130, 131, 134, 137–139, 141, 145–149, 152–156, 159–161, 163, 165, 167, 169, 172, 175, 179, 181, 184, 185, 189–191, 195, 197

NFV Network Functions Virtualization. 4–12, 15, 17, 28, 31, 32, 34, 36, 38–41, 43, 44, 46–52, 54–58, 61–63, 65, 69–72, 74–77, 79–81, 86, 87, 90–95, 97–99, 101, 102, 104, 108, 109, 111, 120, 125–128, 139, 141, 145–147, 149, 151, 154–156, 158, 159, 161–164, 178–181, 183–185, 189–195, 197

NIC Network Interface Card. 11, 17–20, 22, 23, 26–29, 34–37, 39, 44–47, 49, 50, 53, 54, 57, 61–63, 66, 69–71, 74, 80, 81, 84, 87, 109, 112, 113, 126, 128–132, 138, 139, 151–154, 157, 160, 163, 164, 169, 171, 172, 179–181, 186, 192, 197

NPU Network Processing Unit. 36

NUMA Non-Uniform Memory Access. 26, 28, 126

ONOS Open Network Operating System. 152, 164, 180, 181

OS Operating System. 9, 17–22, 28, 32, 36, 40, 41, 43, 55, 62, 65, 70–72, 74, 75, 80, 84, 89, 90, 92, 94, 102, 108, 109, 146

OVS Open vSwitch. 35, 87, 164, 178, 179

OVSK Kernel-based Open vSwitch. 45, 46, 72, 73, 81, 82, 86–88, 91, 103, 104, 106, 107, 184

PCI Peripheral Component Interconnect. 17

PCM Performance Counter Monitoring. 79, 189

PID Process Identifier. 74, 92, 94

PMU Performance Monitoring Unit. 41, 108

pps Packets Per Second. 32, 93, 134

PU Processing Unit. 116, 119, 120

RFC Request For Comments. 2, 3

RMT Reconfigurable Match Tables. 33, 34, 180

RR Round-Robin. 75, 97, 98, 102

RSS Receive-Side Scaling. 35, 47, 50, 53, 113, 126, 128, 132, 165, 169, 171, 185

Rx Reception. 20, 23, 28, 34, 36, 73, 84

SCC Service Chain Coordinator. 70, 72, 74, 75, 77, 79–81, 83–95, 98, 99, 101–108, 156, 181, 183, 185, 187, 189, 190, 194–196

SDN Software Defined Networking. 4–8, 15, 31–34, 37–40, 145, 147, 152, 181

skbuff Socket Buffer. 20, 23, 74, 77, 85, 86

SNF Synthesized Network Functions. 10–12, 53, 111–114, 116–123, 125–132, 134, 135, 137–139, 141, 143, 145–149, 152, 155, 156, 164, 179, 183, 185–187, 189–191, 194–196

SNMP Simple Network Management Protocol. 164

TCP Transmission Control Protocol. 21, 37, 39, 41, 63, 115, 122, 123, 126, 139

TCU Traffic Class Unit. 112, 113, 116, 119, 120, 126, 127, 132, 135, 145, 149

TLB Translation Lookaside Buffer. 76, 108

TTL Time To Live. 32, 114, 116, 123–126, 148, 169

Tx Transmission. 20, 23, 28, 34, 36, 73, 85, 86

UDP User Datagram Protocol. 37, 41, 63, 122–126, 139, 148

VLAN Virtual Local Area Network. 37, 164

VM Virtual Machine. 36, 46, 53, 70, 79, 149, 180, 181, 195

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. Over the 20$^{\text{th}}$ century, communications were substantially facilitated by numerous profound technological advancements, such as telecommunications.

Initially, 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$^{\text{th}}$ century, telecommunications together with the digital revolution and the concomitant emergence of computers, allowed people to organize remote computer systems into telecommunications networks.

1.1 Today's Telecommunications Landscape

The 20$^{\text{th}}$ century established a global telecommunications network widely known as the Internet. Over the first two decades of the 21$^{\text{st}}$ 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 networks, 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, such as the Fifth Generation (5G) networks, have substantially evolved the Internet and contribute to an ever-increasing load upon telecommunications networks.

A forecast from Cisco's 2016 Visual Networking Index [2] offers interesting findings regarding the evolution of the telecommunications as a result of near future traffic expectations:

1. Annual global Internet Protocol (IP) traffic would 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 [3] investigated the effects of this phenomenon on industry. Their key conclusion is that the historically common connection between traffic and revenue has been blurred because of the increase in data volumes due to multimedia content and other sources of traffic. This increase in data volume requires investments in infrastructure, but at the same time revenues from voice calls are rapidly declining, thus affecting the economic equilibrium of telecommunications' stakeholders.

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 network and service providers; for example Amazon [4] 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 [5]. Moreover, market studies [6] showed that security is of utmost importance for industrial information technology stakeholders, hence information technology security services have been showing significant growth.

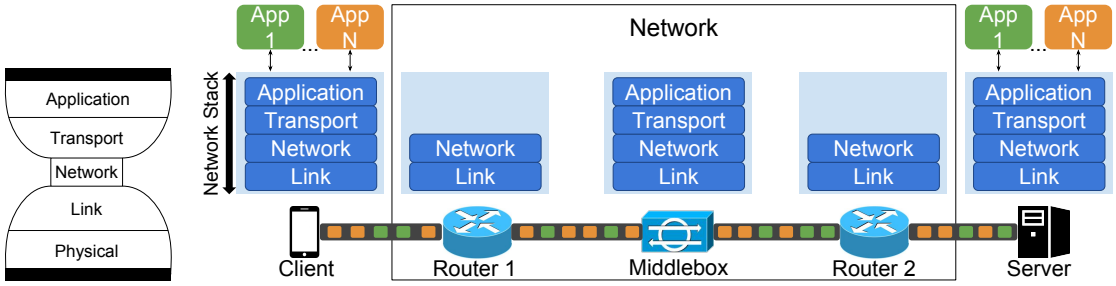
A means for telecommunications' stakeholders to increase the quality of their services is through carefully placing Network Functions (NFs) in the network. An NF takes packets at a network element's input port(s), and outputs (potentially modified) packets from the same network element's output port(s). Based on Internet Engineering Task Force (IETF)'s Request For Comments (RFC) 3234 [7], 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 routing/forwarding 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 a path (e.g., the shortest path) between a given source and destination. Another advantage is that these functions are simple and keep the network layer thin. These attributes are desirable 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 link layers. This is the well-known hourglass [7] 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 IETF RFC 3234 [7], these advanced functions (deeply and often statefully) inspect and modify the packets' structure, rewriting fields across the entire header, and in some cases even examining and modifying the packets' payload. For example, 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 Load Balancers (LBs) or wide area network optimizers; and
3. increasing security using Intrusion Detection System (IDS) and firewall 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 [8], 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 were 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 [9]. Hence, continuous replacements of specialized hardware are required by the network operators to scale their 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 [10], production networks still rely on middleboxes from different vendors that expose proprietary Application Programming Interfaces (APIs), thus posing complex management requirements [11];
3. In 2012, J. Sherry et al. stated that over the past 5 years, large enterprises had to budget an additional $1 million dollars in order to maintain their middleboxes [8]; 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 [8].

1.3 The Role of Software in Modern Networks

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) [12].

SDN decouples 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 make 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 [13] protocol in 2008, with campus [14], wide area network [15, 16, 17], and datacenter [18, 19] deployments gradually replacing traditional network designs.

NFV is important because 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 on the packets' payload. As discussed in §1.2 and depicted in Figure 1.2a, these functions have been traditionally offered by hardware-based middleboxes. In contrast, Figure 1.2b shows how NFV migrates middlebox functionality from hardware to software, running on commodity off-the-shelf servers, potentially eliminating most of the challenges posed by traditional middleboxes (see §1.2.2). Although NFV deployments have begun [20, 21], as we will see later on, NFV comes with its own problems and challenges.

(a) Traditionally, network elements were based on specialized (thus expensive) hardware.

(b) Today, network elements are based on commodity (thus inexpensive) hardware.

Figure 1.2: Packet processing in traditional vs. modern networks.

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 1.3, when a user at his/her home network requests a web page via the Hypertext Transfer Protocol (HTTP), 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 firewall, a Deep Packet Inspection (DPI), and an LB might all reside at the target datacenter before the HTTP server [23].

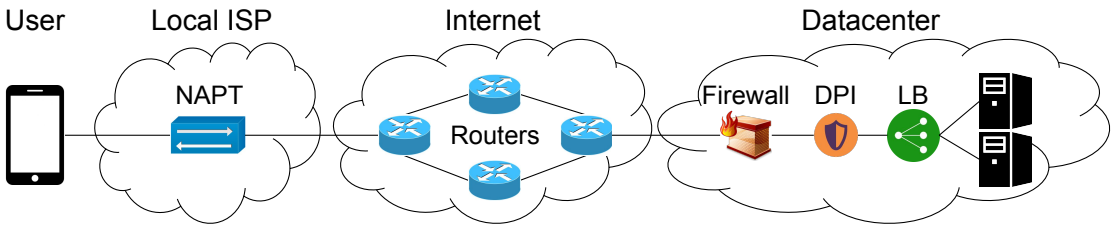


Figure 1.3: An example service chain between a user connected to a local ISP and a service hosted by a server farm in a datacenter. The service chain consists of a NAPT, several core routers, a firewall, a DPI function, and an LB.

The service chain illustrated in Figure 1.3 offers the following benefits: First, the NAPT helps the ISP of the user's access network conserve public IP addresses by multiplexing multiple users' traffic into traffic that uses a single public IP address, but different ports. Second, the core routers route traffic along a desirable path (e.g., a path with the shortest number of hops or the least congested path) between the local ISP's network and the closest datacenter that hosts the web page. Third, the firewall drops illegitimate traffic attempting to enter the target datacenter. The DPI adds another layer of security by flagging malicious content in traffic. Finally, the LB ensures that legitimate users' requests will be served as soon as possible, by selecting the least loaded server.

By offloading middlebox functionality to the cloud, telecommunications' stakeholders can realize the example service chain shown in Figure 1.3 in software, using SDN and NFV. As depicted in Figure 1.4, a system administrator can specify such a service chain as a high-level policy that targets the purple 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

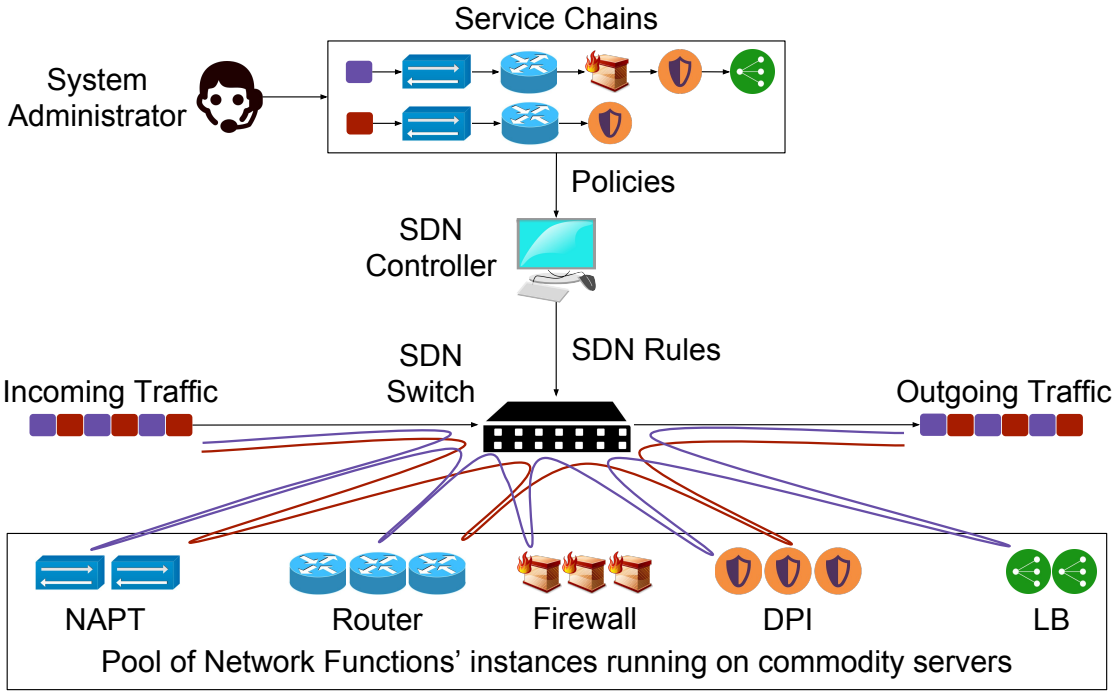


Figure 1.4: An example set of service chains translated by an SDN controller into SDN rules that steer the traffic through multiple NF instances.

policy into low-level SDN rules (e.g., using the OpenFlow protocol), that need to be installed in the underlying SDN switch, to forward the purple 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 red 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 red packets through that service chain. The software-driven paradigm depicted in Figure 1.4 also ensures on demand resource provisioning, by tearing down the NFV instances of the service chains, once the traffic has been served.

1.5 High-level Research Challenges

Link speeds in networks will in the near-future reach and exceed 100 Gbps [24, 25]. While available specialized hardware can accommodate these speeds, as introduced in §1.3, modern networks have adopted NFV; a new networking paradigm that replaces expensive specialized hardware with open-source software running on general purpose commodity hardware.

Despite the advances in SDN and NFV (see Chapter 3) as well as the dramatic evolution of high-speed computing during the last decades, achieving high performance using commodity hardware is extremely challenging. This is because commodity systems, based on general purpose Central Processing Units (CPUs), suffer from the well-known processor-memory gap. According to Patterson et al., this gap grows by 50% per year because processor speeds increase much faster than memory speeds [26].

Given the above trends related to the evolution of link, processor, and memory speeds, the chain of facts shown in Figure 1.5 is the motivation for this thesis:

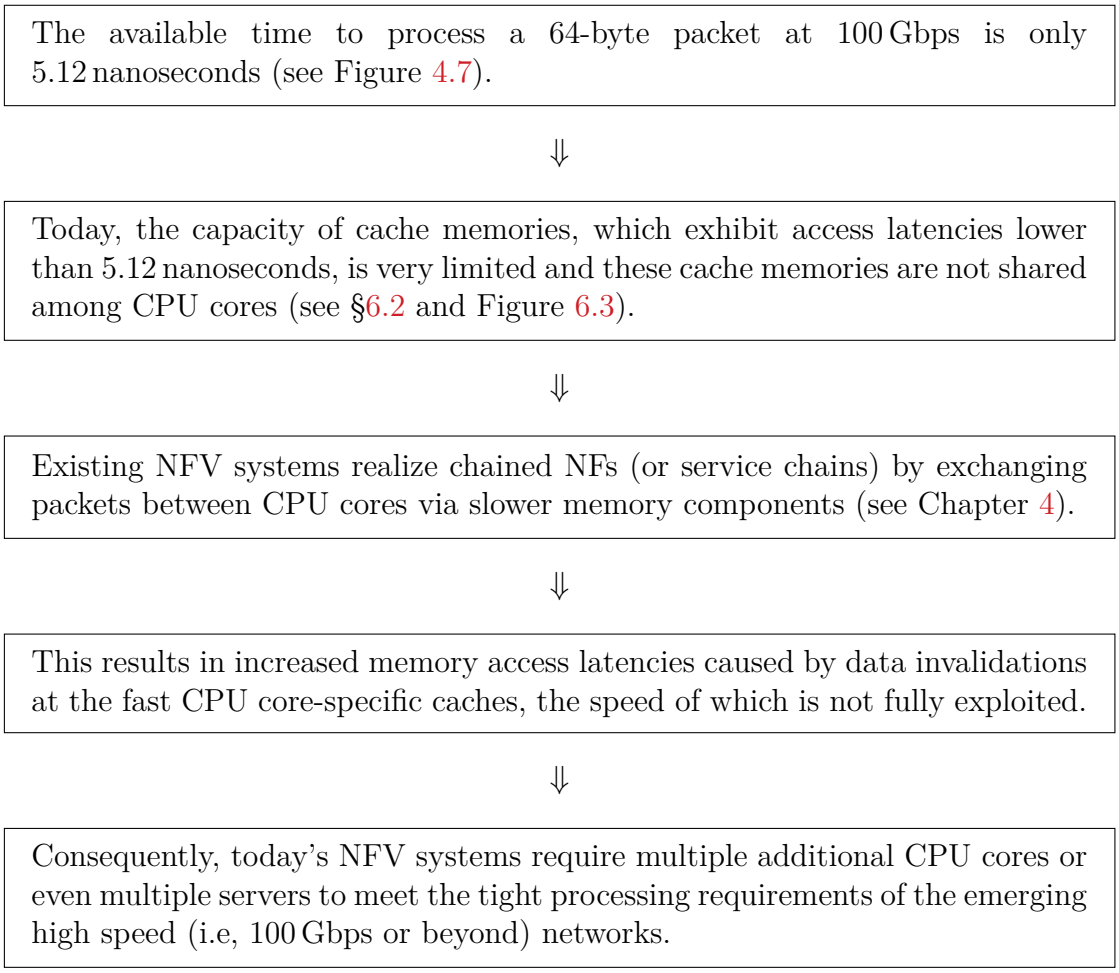


Figure 1.5: Motivation for this doctoral thesis.

This thesis redefines the state of the art in NFV packet processing, as it effectively addresses all the above issues. A more detailed description of the problems and challenges tackled by this thesis is provided in Chapter 4. Next, a summary of contributions is presented, paving the way to address these challenges.

1.6 Summary of Contributions

The challenges discussed in §1.5 and summarized in Figure 1.5 motivated the work for this thesis. This section summarizes the key contributions of this thesis to the research community and industry. 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 comprised 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 a standard Linux network driver. In this study, I designed and implemented an NFV profiler (see §6.2 starting on page 74) that can thoroughly monitor low-level performance counters, during the execution of NFV service chains [27]. The goal of this profiler was to uncover the reasons that cause service chains to exhibit high latency with an increasing service chain length.

To the best of my knowledge, this is the first NFV profiler in the literature. My tool could be leveraged by network operators to reveal the performance bottlenecks of their service chains. The novelty of this work is described in §6.6 starting on page 107.

1.6.2 Contribution 2

The results of the NFV profiler uncovered Input/Output (I/O) and scheduling overheads that increased per packet latency linearly with an increasing service chain length (see §6.3 starting on page 81). To mitigate these overheads, I built a tool [27] that accelerates user-space NFV service chains by combining custom scheduling policies with I/O multiplexing techniques (see §6.4 starting on page 91).

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 starting on page 99). 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 these results is given in Table 6.1.

To the best of my knowledge, this is the first time that I/O multiplexing and scheduling techniques were 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 starting on page 107.

1.6.3 Contribution 3

The standard Linux-based network driver used in the first two contributions 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) [28] (described in §2.3). Despite using this latest advancements in packet I/O, performance bottlenecks remained, mainly due to redundancy in the internal operations applied by the service chains. The next logical step towards faster and lower-latency NFV service chains was to revise the way service chains were deployed. For this reason, we* proposed Synthesized Network Functions (SNF) [29]; a framework that drastically consolidates chained NFs by synthesizing their internal operations into a new equivalent state machine. This contribution is described in detail in Chapter 7, where we propose a practical NFV system (see §7.1 starting on page 112) and its synthesis techniques (see §7.2 starting on page 114). 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 the underlying experimental testbed (see §7.5.2, §7.5.3, and §7.5.4.1). By enhancing this testbed with a hardware-based OpenFlow switch, we realized line-rate ISP-level service chains despite increased complexity at 40 Gbps (see §7.5.4.2 starting on page 138). The originality of this work with respect to earlier efforts is described in §7.7 starting on page 146.

*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 40 Gbps line-rate performance. Finally, the hardware-assisted experiments of this work were jointly conducted by Georgios P. Katsikas and Maciej Kuźniar.

1.6.4 Contribution 4

Although the proposed synthesis techniques provide an efficient software stack for service chains, they do not address how to efficiently map the instructions of the (synthesized) code onto the available hardware resources. According to the results of experiments described in Chapter 4, this mapping greatly affects the performance of NFV service chains, as severe performance penalties occur when packets are not directly processed by the correct CPU core.

To address this challenge, in Chapter 8 we* introduced the design (see §8.1 starting on page 152) and implementation (see §8.2 starting on page 163) of Metron; an NFV platform that automatically associates packet processing operations with programmable hardware components to realize NFV service chains at the true speed of the underlying hardware [30]. Metron eliminates the need for costly inter-core communication at the servers by delegating stateless packet processing and CPU core dispatching operations to programmable hardware devices, while realizing stateful packet processing in software running on commodity servers. Doing so offers dramatic hardware efficiency and performance increases over the state of the art. With commodity hardware assistance Metron fully exploits the processing capacity of a single server to deeply inspect traffic at 40 Gbps (see §8.2.3.1 starting on page 165) and execute stateful service chains at the speed of a 100 Gigabit Ethernet (GbE) Network Interface Card (NIC) (see §8.2.3.2 starting on page 169). Moreover, §8.2.4 shows how Metron ensures zero inter-core communication even after scaling service chains in/out, while in §8.2.3.3 we evaluate Metron's low-cost service chain placement scheme. The novelty of Metron is described in §8.3 starting on page 179.

1.7 Impact and Relevance of this Thesis

This thesis has made a major contribution in the area of modern programmable networked systems. To visualize the impact of this contribution, the results produced by the work described in this thesis are compared with state of the art approaches as follows:

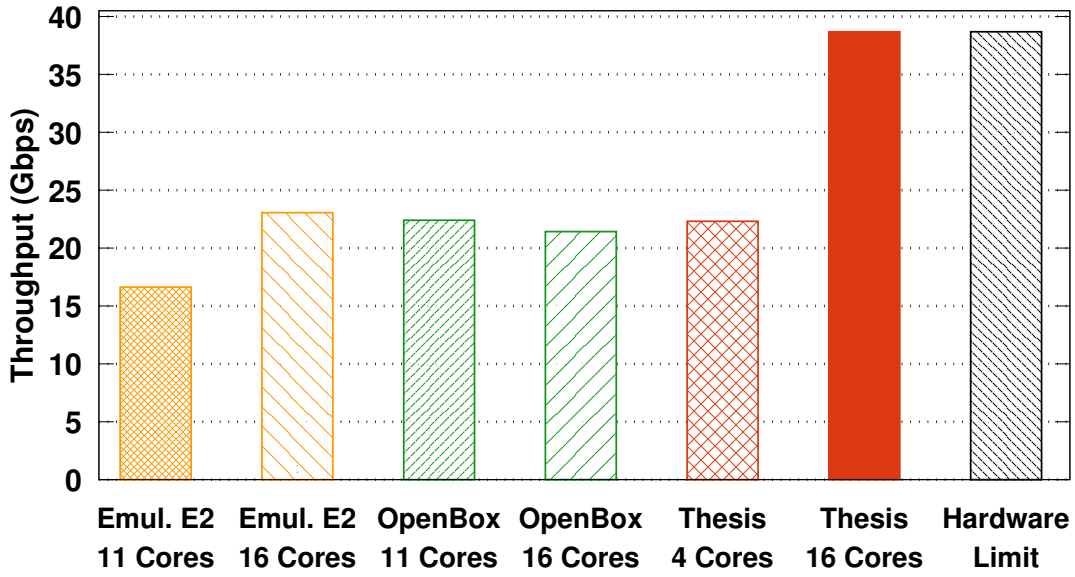
1. Metron [30] with SNF [29] (i.e., two contributions of this thesis);
2. the OpenBox state of the art framework [31]; and
3. an emulated version of the E2 state of the art framework [32].

*Metron was jointly implemented by Georgios P. Katsikas and Tom Barbette.

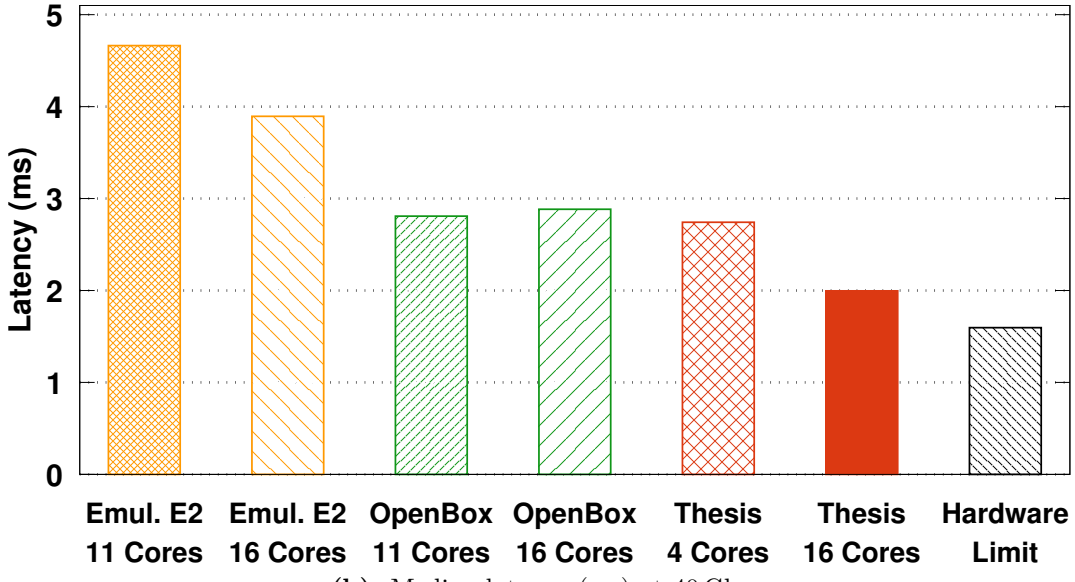
Service Chain Deployment at 40 Gbps
Figure 1.6 demonstrates how this thesis enables deep packet inspection (using a Firewall→DPI service chain) at the speed of a 40 Gbps testbed. More information about these results can be found in §8.2.3.1 starting on page 165.

Service Chain Deployment at 100 Gbps
Figure 1.7 demonstrates the first NFV platform able to provide stateful network services (using a Router→NAPT→LB service chain) at the speed of a 100 Gbps testbed. More details about these results are provided in §8.2.3.2.

Quantitative Impact and Relevance
These results are possible thanks to the joint contribution of SNF's service chain synthesis (see Chapter 7) and Metron's offloading & accurate traffic dispatching (see Chapter 8). Metron with SNF (and consequently this thesis, denoted as "Thesis" in Figures 1.6 and 1.7) realizes NFV service chains with (*i*) 2.75-6.5x better efficiency, up to (*ii*) 7.8x lower latency, and (*iii*) 4.7x higher throughput than the state of the art. More importantly, these results meet the performance levels of the underlying hardware (denoted as "Hardware Limit" in Figures 1.6 and 1.7), enabling the networking industry to keep up with the increasing link speeds (as discussed in §1.5), despite using inexpensive programmable hardware. This thesis is relevant for telecommunications' stakeholders aiming to provide high speed network services at 40, 100 Gbps, or beyond. Prime examples are large network operators, such as AT&T and Telia, as well as popular service providers, such as Google, Microsoft, Amazon, Facebook, Twitter, etc.

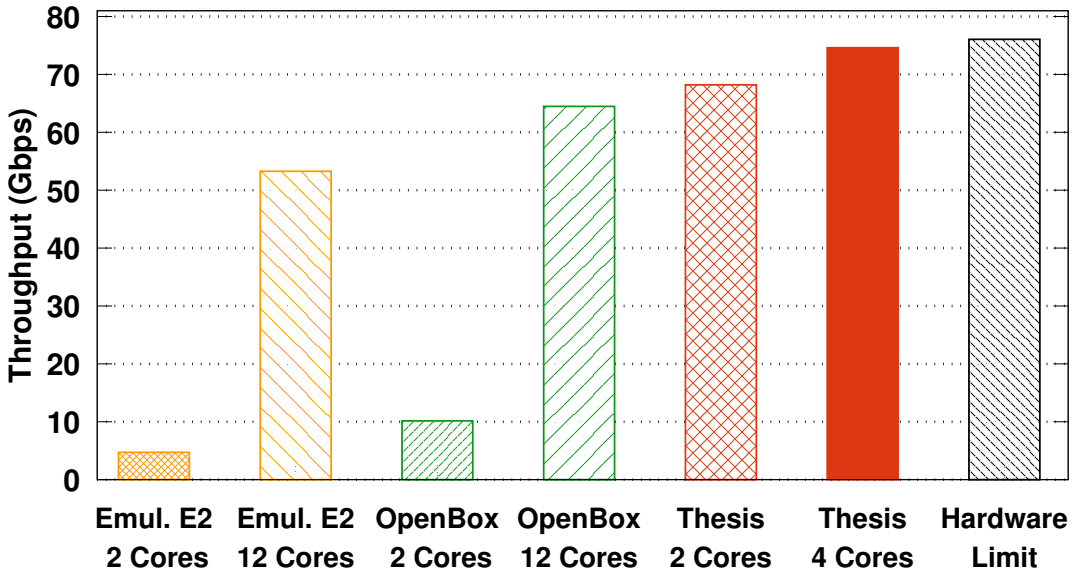


(a) Median throughput (Gbps) at 40 Gbps.

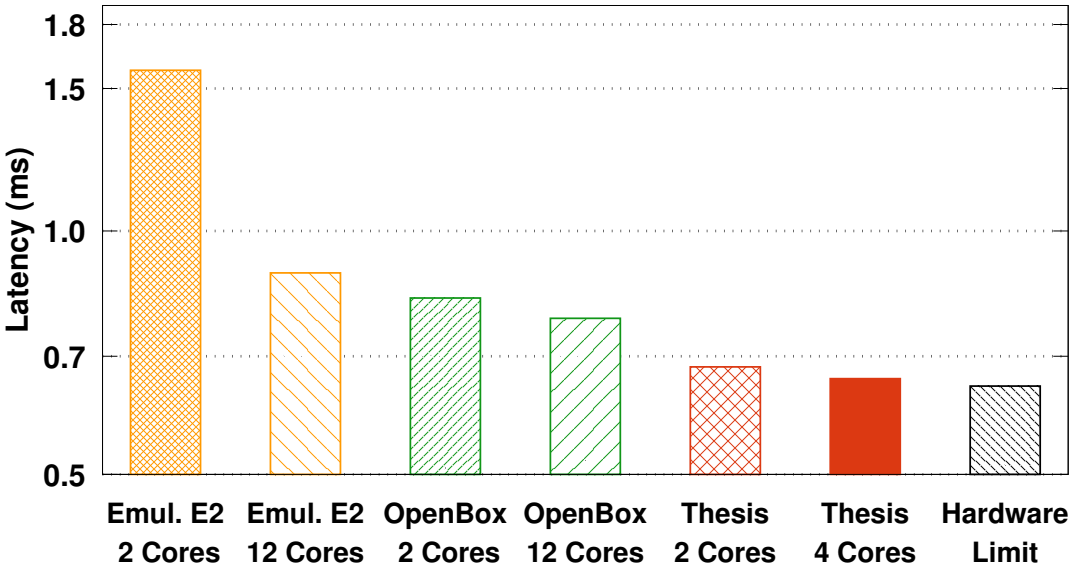


(b) Median latency (ms) at 40 Gbps.

Figure 1.6: Performance comparison of (*i*) this thesis (using SNF and Metron) and (*ii*) state of the art approaches (i.e., OpenBox and an emulated version of E2), when deeply inspecting (Firewall→DPI service chain) traffic at 40 Gbps. The performance limit of the underlying hardware is denoted as "Hardware Limit".



(a) Median throughput (Gbps) at 100 Gbps.



(b) Median latency (ms) at 100 Gbps.

Figure 1.7: Performance comparison of (*i*) this thesis (using SNF and Metron) and (*ii*) state of the art approaches (i.e., OpenBox and an emulated version of E2), when realizing a stateful service chain (Router→NAPT→LB) at 100 Gbps. The performance limit of the underlying hardware is denoted as "Hardware Limit".

1.8 Thesis Outline

The rest of this thesis is structured as follows. Chapter 2 provides background information required to understand this thesis. Chapter 3 provides an extensive literature study in the area of networked systems and NFV & SDN in particular. Given this background, Chapter 4 highlights the performance issues of state of the art NFV systems. The challenges that need to be addressed to solve these issues are also identified in §4.3 and linked with the contributions of this thesis. Having identified the problems and challenges, I then formally compose a clear research question in §4.4. Chapter 5 describes the experimental design and tools used to conduct the experiments underlying this thesis. Chapters 6, 7, and 8 introduce unique scientific contributions to approaching and solving the identified research problem. Chapter 9 challenges the research problem tackled by this thesis, by revisiting the hypotheses made in Chapter 4 and by relating the results from Chapters 6, 7, and 8 with these hypotheses. The limitations of this thesis are discussed in Chapter 10, where future work plans are also sketched. Finally, Chapter 11 positions this work in today's societal, ecological, and economical planes, while Chapter 12 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 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. §2.4 and §2.4.1 describe memory organization in computer systems. Finally, §2.4.2, §2.5, and §2.6 discuss various auxiliary technologies and tools that affect the performance of modern programmable networked systems.

2.1 Network Interface Cards

A NIC is a subsystem that transmits and receives data to/from a physical medium, such as a coaxial cable or an optical fiber. Note that while the acronym comes from a name that has the word "card" in it, the physical realization of a NIC need not be on a separate card, there can even be many of them on the same die as the system's CPU. Assuming that this communication is done via Ethernet NIC connected via a bus interface (in this case a Peripheral Component Interconnect (PCI) bus), a block diagram of a NIC's architecture is depicted in Figure 2.1.

As shown in 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 memory) out to the network and to receive frames into the NIC's local buffers. The memory provides temporary storage for frames (i.e., 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

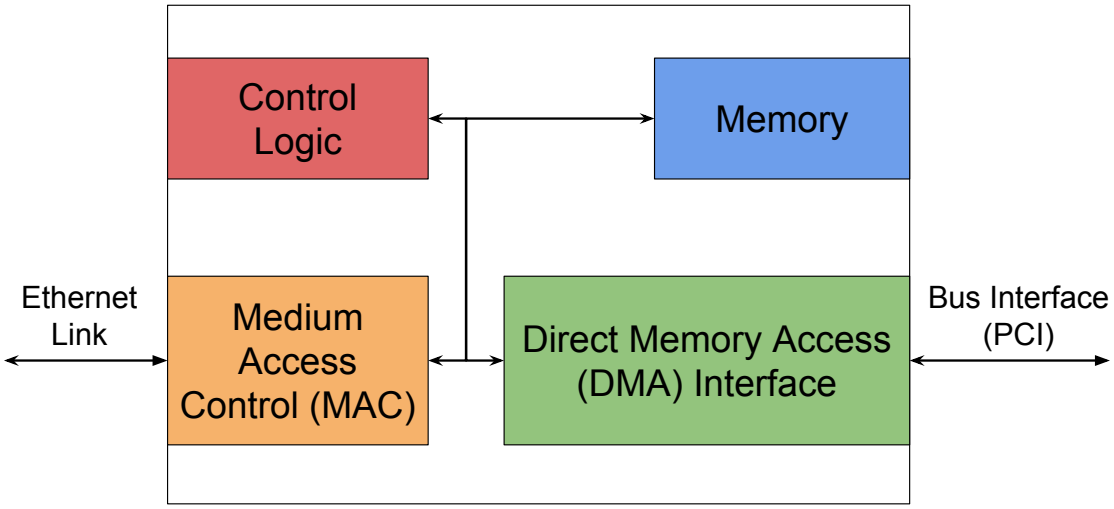


Figure 2.1: An example architecture of a NIC.

memory and the main memory or the 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 take place in parallel with the execution of instructions by the system's CPU.

2.2 Traditional Networking I/O Paradigm

Remotely located applications communicate with each other via a network. As explained in Chapter 1, such a network is comprised of interconnected devices, some of which are responsible for finding a path between a local and a remote application. In a computer system, a typical way for applications to interact with the underlying network is via an OS. An OS provides APIs that allow applications to access and share the underlying hardware. In particular, network applications interact with three key parts of the hardware: (*i*) the CPU, (*ii*) the memory system, and (*iii*) one or more network interfaces, commonly referred to 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's size. A buffer descriptor is the transaction unit between the OS and the NIC.

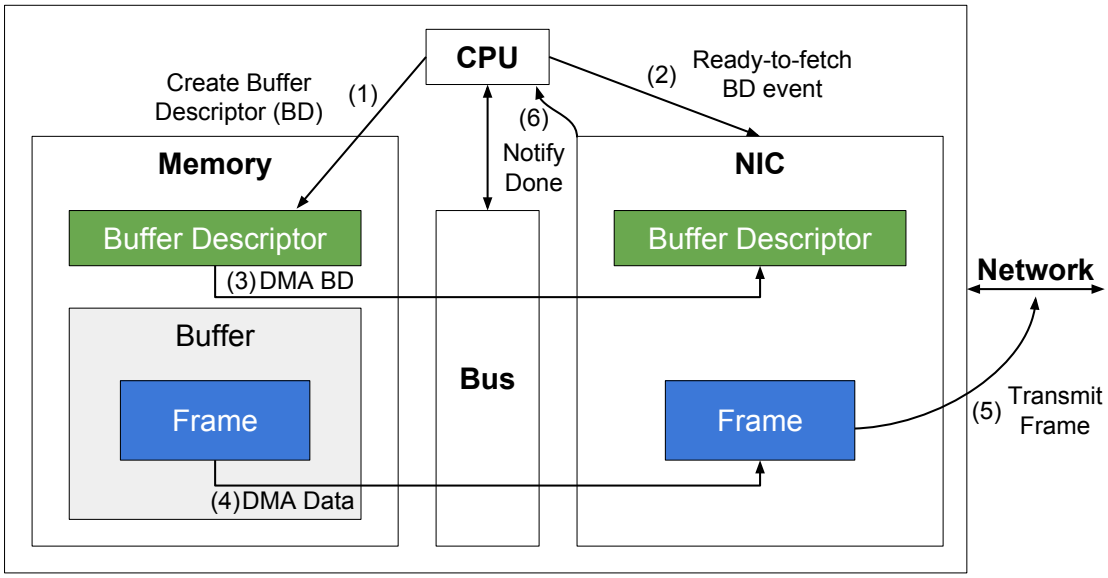


Figure 2.2: Steps for sending an Ethernet frame. Buffer descriptor is abbreviated as BD.

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 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 the frame in its local receive buffer (Step 1). By examining the next free buffer descriptor (in Figure 2.3 we assume that this descriptor is already created), the NIC determines the memory address where the frame will be stored. Then, the NIC initiates a write operation of the frame's data via DMA (Step 2). Once the frame is written into memory, the NIC updates the buffer descriptor with the size occupied by the new

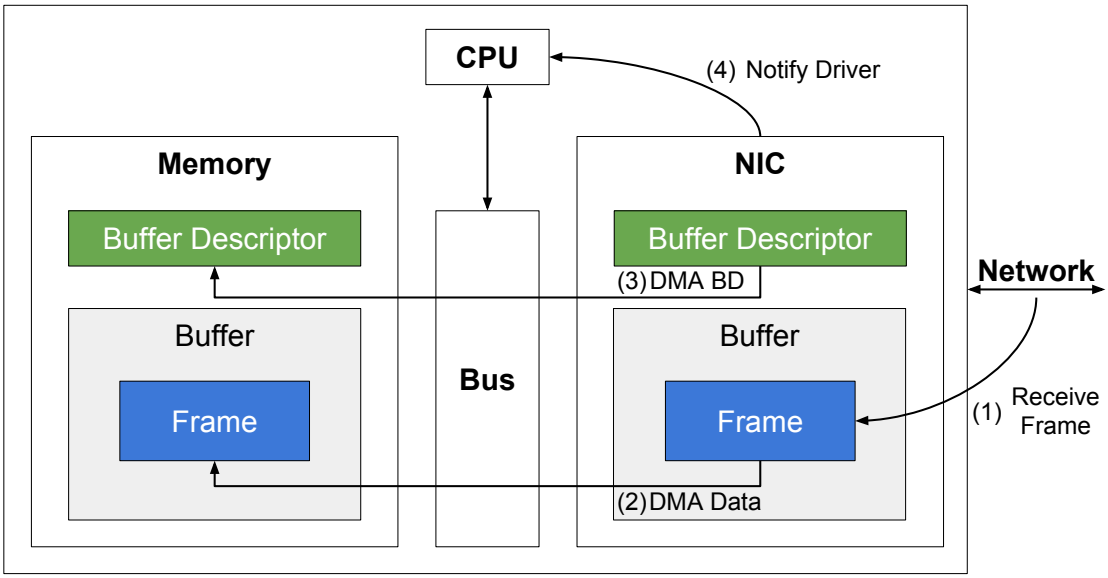


Figure 2.3: Steps for receiving an Ethernet frame. We assume that the OS has already created a buffer descriptor (BD) that points to a free memory region and the NIC has read this descriptor into its local memory via DMA.

frame and possibly some checksum information. The updated buffer descriptor is now ready to be written to 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).

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 how 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, such as 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 new frame in the kernel's queue, a software interrupt is triggered by the kernel as shown in Figure 2.4. Next, the

*Interrupts are discussed in §2.6.

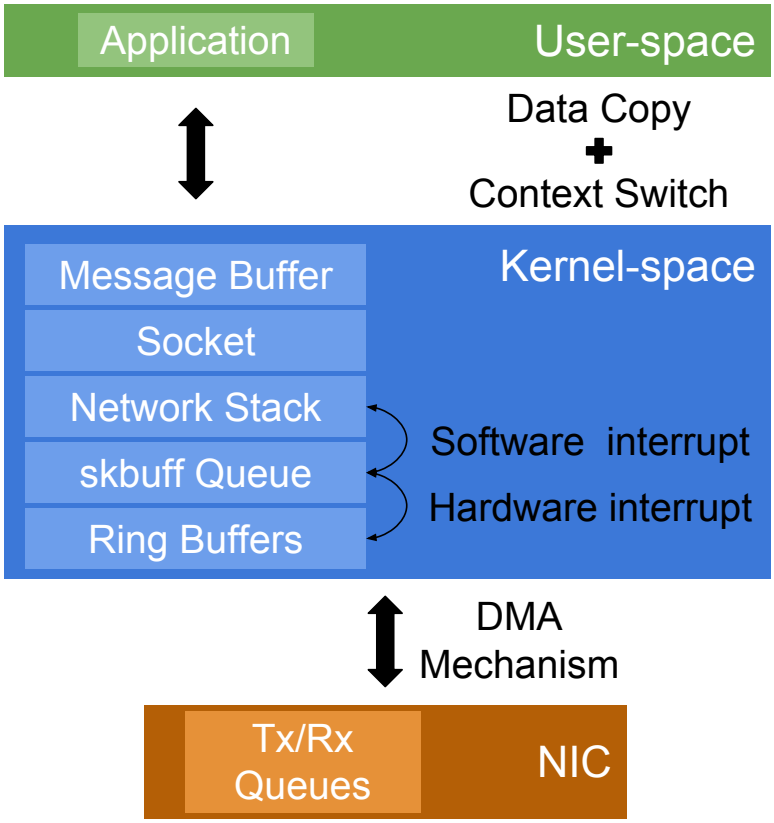


Figure 2.4: Traditional Linux network I/O paradigm.

payload that is encapsulated in the frame will dictate 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.

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 on a uni-processor system, 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 I/O Paradigm

The interactions between applications and NICs through the Linux OS described in the previous section indicate the presence of a substantial amount 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 §2.6.

This section discusses recent approaches [28, 33, 34] 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 in §3.2.2. The goal of this section is to highlight 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.

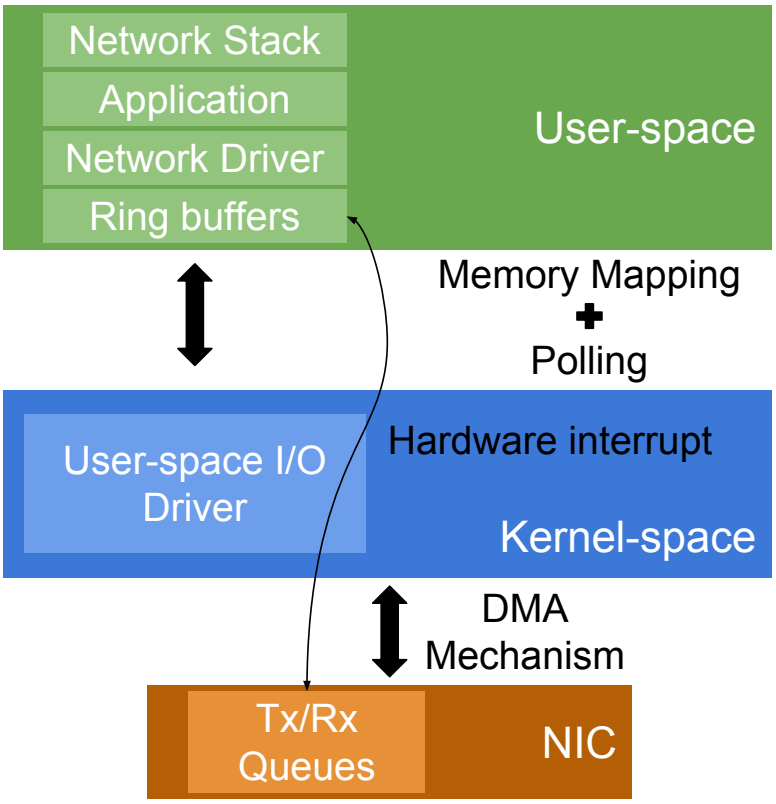


Figure 2.5: A revised Linux network I/O paradigm.

The main principle of the DPDK network I/O paradigm is that the Linux kernel does not include 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 applications access the ring buffers directly. This paradigm implies that the majority of the network driver's functionality, traditionally residing in the kernel (as was 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 to the memory shared between the application and the device;
2. data copies from kernel to user-space and the reverse are eliminated because user-space applications directly access the device's packet pool;
3. context switches are almost entirely eliminated because most of the network driver's functionality is now moved to user-space (although some functionality still resides in the kernel's user-space I/O driver as shown in Figure 2.5); and
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 performance at the cost of adding complexity to the application development 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". In contrast, in the traditional Linux network I/O scheme, this packet representation model is maintained in the kernel using skbuffs; hence applications can only access the skbuffs' contents (indirectly) using the socket system calls' API.

*Polling is discussed in §2.6.

2.4 Memory Hierarchy

In 1945, John Von Neumann proposed a design architecture for an electronic digital computer [35]. This computer architecture design consisted of (*i*) a processing unit containing a control unit, arithmetic and logic unit, and registers, (*ii*) a memory unit, and (*iii*) input and output devices, as shown in Figure 2.6.

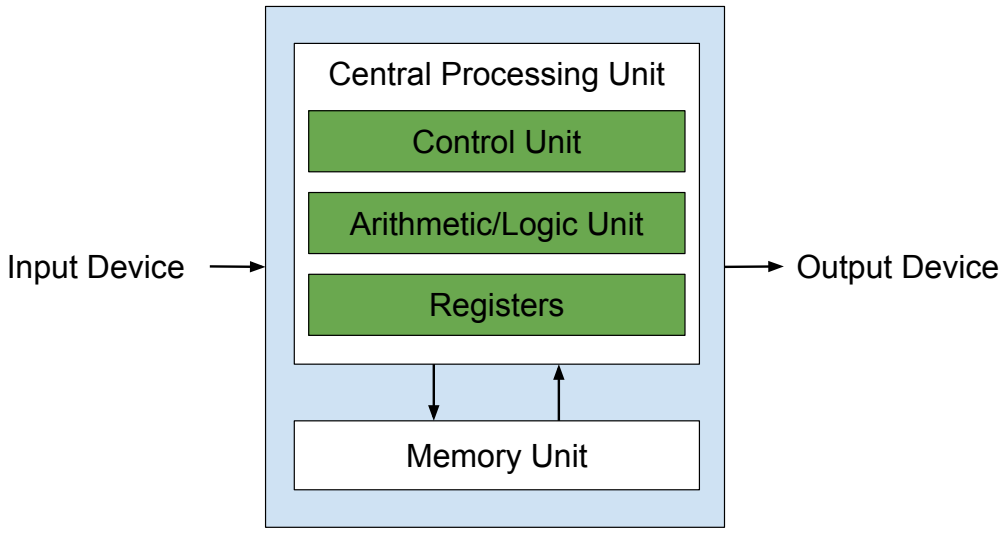


Figure 2.6: Von Neumann computer architecture, introduced in 1945.

Processor-Memory Performance Gap
Since the foundations of computer systems, memories have always been storing (and supplying) data and instructions to processors. However, processor speeds have been increasing faster than memory speeds [26]. The processors' clock speeds of the first computers were measured in Hertz or kilo Hertz, but the clock speeds of modern CPUs are commonly advertised as several giga Hertz. Transistor counts have also been doubling every almost 3 years, approximately following Moore's law [9]. In contrast, memory speeds have been following a much slower increase, as discussed earlier in §1.5. This is the well-known processor-memory performance gap, which grows 50% per year according to Patterson et al. [26].

Cache Memories Attempt to Bridge the Processor-Memory Gap
To decrease the latency to access a piece of memory, thereby bridging the processor-memory performance gap, micro-architectural techniques place faster memory components physically closer to the CPU, by building different hierarchies of caches. Figure 2.7 shows the memory hierarchy of an Intel Xeon E5-2667 v3 processor. According to this model, some memory components (e.g., registers, core-specific caches) are directly integrated into the processor, while other

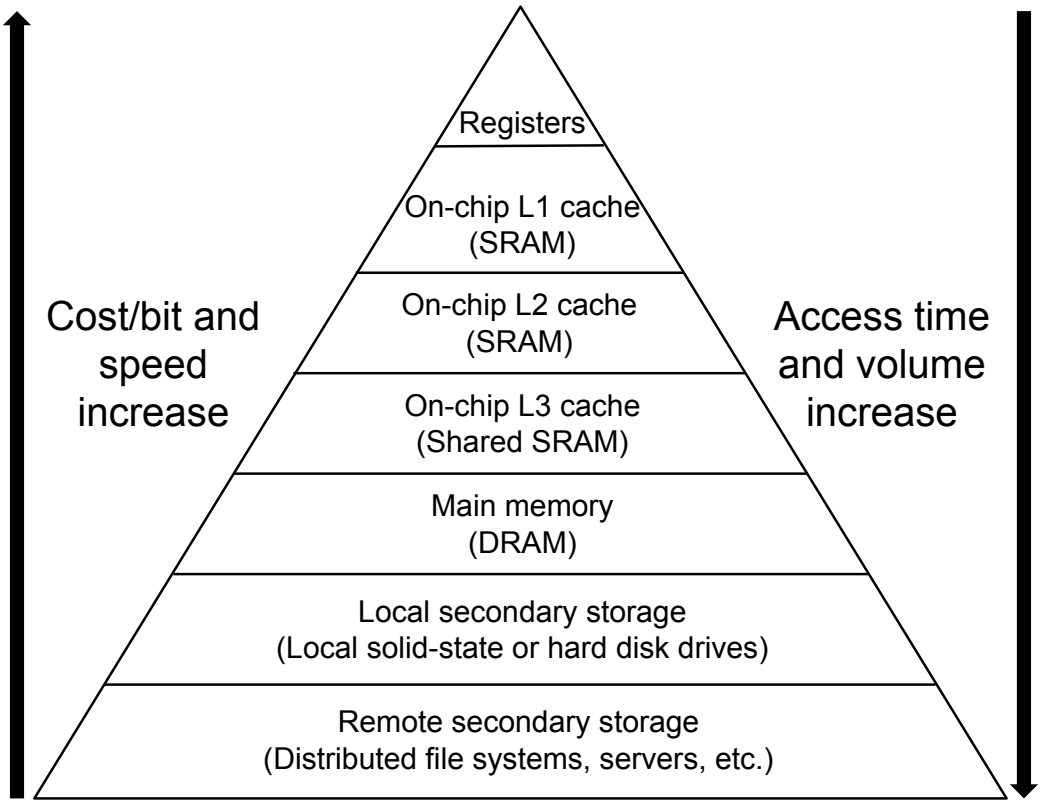


Figure 2.7: Memory hierarchy of an Intel Xeon E5-2667 v3 hardware architecture.

components (e.g., shared cache, main memory, and disks) are spatially dispersed around the processor, or even remote e.g., distributed file systems across multiple computers. Typically, the closer a memory component is to the processor, the faster, smaller, and more expensive it is according to Figure 2.7. Consequently, the key to achieving high performance is to achieve high hit rates on the smallest memories (e.g., L1/L2 cache), which translates to fewer costly accesses to/from farther memory components (e.g., main memory).

2.4.1 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 2.8. 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.

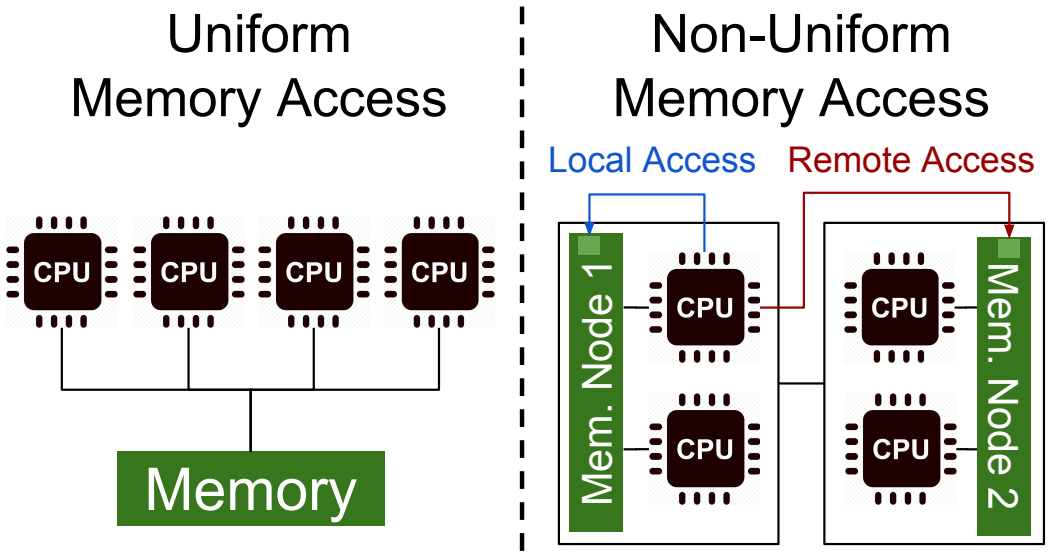


Figure 2.8: Uniform (left) vs. non-uniform (right) memory access models.

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. 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 [36]. As shown in Figure 2.8, 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.

2.4.2 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 2.9 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 is the core able to fetch the data by issuing a transaction between LLC and its local cache.

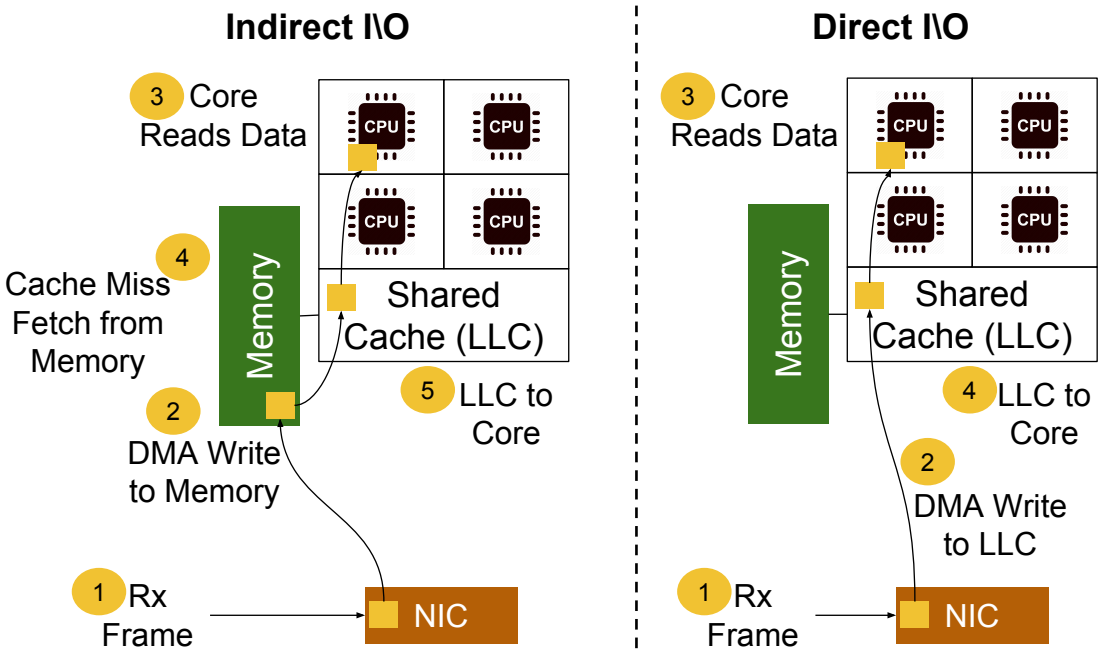


Figure 2.9: Indirect (left) vs. direct (right) network I/O models during a frame reception operation.

This network I/O model is limited by the available memory bandwidth that interconnects main memory and the LLC as well as 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 their Direct Data I/O (DDIO) technology [37]. As shown in the right-most part of Figure 2.9, 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 portion of LLC used for DDIO depends on the hardware architecture. CPU cores fetch 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 can keep most of their data in the caches.

2.5 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 a task is explicitly assigned 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.

2.6 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 (Mpps), 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 interrupts across all the available cores. This means that no single core will be loaded by serving 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 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 on 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. However, the use of polling implies that a CPU core must listen 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 NFV. The goal is to draw inspiration from these efforts and identify those gaps that motivated this thesis.

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

This chapter provides a high-level overview of the literature. Direct comparisons of the contributions reported in this thesis with the literature are given in §6.6, §7.7, and §8.3.

3.1 Early Days of Networking

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 software-based packet processing began.

The UNIX System V STREAMS [38] was a modular packet processing system that implemented implicit queuing and packet scheduling mechanisms spread throughout a STREAMS configuration. Decasper et al. [39] 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 [40] which provided 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 [41]. 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 [42], a platform to build NFs using simple and modular packet manipulation elements. Combining 16 Click elements, such as IP lookup and decrement IP Time To Live (TTL) field, one can build a software router. To extend this router into a more sophisticated NF (e.g., a middlebox that implements drop policies or differentiated services), a few elements can simply be added 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 Packets Per Second (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 [42]. Also, as compared to router plugins, Click allows more dynamic traffic classification, to be embedded into Click's pipeline.

3.2 Modern 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.

Software-based packet processing architectures achieved higher performance with the advent of multi-processor computing around 2007. In conjunction with the introduction of OpenFlow [13] in 2008, software has become a first class component creating the trend that is today called SDN. Because of these changes, the logical successors of software-based packet processing are SDN and NFV.

3.2.1 Switch Programmability using SDN

The separation between control and data plane was first proposed by Casado et al. in Ethane [43]. This idea was the genesis of the OpenFlow protocol in 2008 [13] and the beginning of a more software-oriented networking era.

OpenFlow quickly gained popularity and researchers implemented network operating systems and platforms [44, 45, 46, 47, 48, 49] to foster SDN experimentation. To facilitate application development, research efforts focused on providing programming abstractions on top of popular SDN controllers [50, 51, 52]. The software-based nature of this new networking paradigm, quickly revealed some of its weaknesses, such as the introduction of bugs in the network. To

prevent these bugs, researchers proposed network-level verification techniques that (*i*) explore the dynamic interactions between controllers and switches [53, 54] or (*ii*) verify a snapshot of the network policy [55, 56, 57, 58, 59, 60]. The goal of these works was to systematically uncover violations of key network properties, such as loops, policy violations, black holes, interference among traffic slices, etc. Verifying the control plane was insufficient, as bugs might appear in the OpenFlow agent implementations of the switches. These bugs were identified and fixed by switch-level verification schemes [61, 62, 63]. In both cases, researchers either used custom verification tools, such as NICE [53] and Kinetic [54], or exploited prior research, such as KLEE [64] and Cloud9 [65].

Network monitoring [66, 67, 68, 69] and debugging [70, 71, 72, 67] were also important research directions that emerged after the introduction of SDN. The former provides useful information about the state of switches, flows, and the network load, while the latter proposes how to detect and mitigate network failures.

Due to the fact that a network can undergo many changes in states, updating the state of the network elements appeared to be an important problem in SDN. To this end, researchers proposed mechanisms to consistently [73] and sometimes incrementally [74, 75] update the data plane. Additionally, ESPRES [76] and Dionysus [77] can schedule a given update plan so that the time required for this plan to be installed is potentially reduced. Similarly, RUM [78] is a software layer between the controller and the switches that masks and fixes incorrect rule update notifications coming from faulty switches.

To foster commercial SDN implementations, researchers published their experiences from deploying SDN solutions on real networks [14, 15, 16, 17, 18, 19, 59]. In this context, Onix [79] models the control plane as a scalable distributed system, while DevoFlow [80] appropriately modifies the OpenFlow paradigm to achieve the performance-levels required by datacenters. FlowVisor [81] proposes an approach to collocate experiments in a real network by slicing its resources using SDN.

Stateful and Protocol-Independent SDN

After several years of research and several updates in the specification, OpenFlow has improved substantially. However, some researchers challenged this popular SDN protocol. Bosshart et al. in [82] questioned the flexibility of OpenFlow, particularly the static nature of the match-action model and the limited set of packet processing actions. To overcome these limitations, they proposed a Reconfigurable Match Tables (RMT) model. RMT allows programmers to (*i*) define new headers, (*ii*) express a desired task as a parse graph, and (*iii*) use a table flow graph to express the match table topology. All of these are possible without modifying the hardware.

OpenState [83] argued that the stateless nature of the OpenFlow data plane prevents innovation in crucial stateful packet processing operations. Therefore, OpenState allowed SDN switches to perform stateful packet processing without involving an 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 [84], leading to even more powerful SDN switches; for example, to generate packets [85].

Contemporaneously with OpenState, P4 [86] introduced 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 the offspring of RMT and a logical continuation of the OpenFlow match-action protocol, with a simpler and more flexible specification of the same functionality that is target and protocol-agnostic. Several works built upon P4 to introduce programmable switches [87, 88], LBs [89], and network emulators [90], while other researchers were inspired by P4 to build programmable switches using commodity hardware [91, 92].

3.2.2 Commodity Server Programmability using NFV

Switches with increased programmability, as per OpenFlow, do not address the broad needs of network operators for stateful and deep packet processing. For this reason, NFV focused on turning cheap commodity servers into fast packet processing engines that could potentially replace specialized hardware-based middleboxes.

High Performance Network I/O

One of the first problems was to improve the I/O performance of commodity servers. 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 [93]; a modular component of the Linux kernel that permits 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 [94], 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 [95].

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, DPDK [28], netmap [33], and PFQ [34] 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. The DPDK Packet Framework [96] is a production-level software development kit that provides reusable and extensible tools to build complex packet processing pipelines. A major difference between the DPDK Packet Framework and earlier works is that DPDK inherently supports hardware acceleration. Software switch implementations such as Open vSwitch (OVS) [97, 98], VALE [99], CuckooSwitch [100], and mSwitch [101] take advantage of either DPDK or netmap to realize fast switching across (virtualized) NF instances.

Advancements in NICs

Increasingly, NICs are equipped with hardware components, such as Receive-Side Scaling (RSS) [102] and Flow Director [103], for traffic classification and dispatching to CPU cores. RSS uses a static function to dispatch traffic to a set of CPU cores by hashing the values of specific header fields. Intel's Flow Director [103] offers a vendor specific "match-action" API to classify (match) and then drop or dispatch (action) traffic to specific NIC hardware queues that can be accessed by designated CPU cores. DPDK abstracts Flow Director to offer a unique flow API for all DPDK-based NICs. Despite their limited programmability, standard NICs are based on inexpensive hardware, which has been manufactured in large volumes, thus enabling low cost network deployments.

The Rise of Smart NICs

As a result of the increased network programmability, FlexNIC [104] proposed a model for additional programmability in future NICs. To evaluate FlexNIC, Kaufmann et al. implemented a DPDK-based software prototype, which emulates NICs with advanced features. This prototype demonstrates how FlexNIC can be used to accelerate key-value stores by offloading key classification into advanced NICs and by dispatching input keys across multiple CPU cores.

NIC vendors, such as Netronome and Netcope, offer NICs with advanced processing capabilities, also called Smart NICs [105, 106, 107]. Prime examples of Smart NICs' advanced features are: (*i*) OVS offload and acceleration [108, 107], (*ii*) bulk cryptography for various cipher suites and key sizes, (*iii*) *stateful* load balancing, (*iv*) DPI [106], and (*v*) virtual evolved packet core functions. These features are not supported by today's commodity NICs.

Unlike standard NICs, Smart NICs are programmable multi-purpose devices, but with a higher manufacturing cost and a lower manufacturing volume. Field-Programmable Gate Arrays (FPGAs) and Network Processing Units (NPUs) are two prominent technologies used to manufacture Smart NICs. FPGAs contain an array of programmable logic blocks that can be reconfigured using a hardware description language, while NPUs can be re-programmed using software. Both of these technologies are in stark contrast with traditional Application-Specific Integrated Circuits (ASICs), which has been the main technology for implementing traditional middleboxes. Traditional ASICs provide higher performance but lower flexibility and reconfigurability than FPGAs or NPUs. This is the reason that new generations of *programmable* ASICs have recently emerged [109].

Overlay NFV Approaches

With all these accelerations in place, Kim et al. in [110] introduced an extension of the Click modular router that reached 28 Gbps of throughput using aggressive computation and I/O batching techniques. ClickOS [111] and NetVM [112] platforms deployed dedicated Virtual Machines (VMs), running on top of Xen [113] or KVM [114] hypervisors respectively, to realize virtual NFs. Both frameworks achieved packet switching between VMs at 10 Gbps line-rate for any packet size. OpenNetVM [115] showed that VM-based NFV deployments do *not* scale with increasing number of chained instances, hence opted for NFs running natively in lightweight Docker [116] containers. By interconnecting these containers with DPDK Tx and Rx ring buffers, OpenNetVM realized NFV service chains with higher throughput. Flurries [117] builds atop OpenNetVM [115] to provide software-based service chains on a per-flow basis. Flurries also enables multiple DPDK-based NFs to share the same CPU core, thus multiplexing thousands of service chains into a single server. NFP [118] extends OpenNetVM to allow NFs in a service chain to be executed in parallel. Dysco [119] proposes a distributed protocol for steering traffic across the NFs of a service chain. NetBricks [120] introduces new programming abstractions that allow chained NFs to operate on isolated memory areas without the (expensive) need to exchange copies of the same packet. This is possible by exploiting type checking and unique types provided by safe programming languages (e.g., Rust) and compilers (e.g., LLVM [121]).

An alternative direction towards achieving higher performance was to scale software middleboxes for modern, multi-core and sometimes heterogeneous hardware architectures. RouteBricks [122] 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 [123] 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 [124] 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 [125] 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.

3.2.3 Hardware Offloading

Several research efforts have proposed ways to offload specific functions into commodity hardware. DPDK [28] offers a large suite of hardware offloading features, such as Cyclic Redundant Check (CRC), (inner) L3-L4 checksum, large receive, Virtual Local Area Network (VLAN), queue in queue, timestamp, MAC & IP security, and TCP segmentation offloads [126]. Raumer et al. [127] offloaded the cryptographic function of a virtual private network gateway into commodity NICs, for increased performance. MICA [128] encodes database keys into the source port field of User Datagram Protocol (UDP) packets' header, which are classified by Intel NICs [103] in hardware to provide high performance in-memory key-value storage. Similarly, SwitchKV [129] offloads key-value stores into OpenFlow switches, which typically provide greater classification capabilities and more rule capacity than NICs. As discussed above, PacketShader [123], Kargus [130], NBA [124], and APUNet [131] take advantage of inexpensive but powerful GPUs to offload and accelerate packet processing. Smart NICs offload and accelerate software-based OpenFlow switches [108], CPU intensive functions, such as DPI [106], along with stateful operations, such as load balancing [105, 106].

3.3 Middlebox Management and Consolidation

This section discusses middlebox management and consolidation approaches.

Middlebox-aware Policy Enforcement
In [11], 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. FlowTags [132] 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. Following the same principles, Tracebox [133] inspected packets crossing the network to locate middleboxes and identify middlebox interference.

Middlebox Verification and State Management
Header Space Analysis (HSA) [56] provided a network model to compose transfer functions of network elements, including middleboxes. These transfer functions are used by HSA to perform static checking of network policies. NetPlumber [57] is the real-time counterpart of HSA applied to Google's SDN network, the Stanford backbone network, and Internet 2. OpenNF [134] proposed a programming API to safely and quickly migrate the state of operational middleboxes in virtualized environments. Integrating state migration, e.g., using OpenNF, into existing middleboxes requires substantial man power. StateAlyzr [135] eliminates the need for manual middlebox modifications by proposing an automated tool for efficient middlebox state management. An alternative approach to state management was proposed by Kablan, et al. [136], delegating this task to a remote distributed store.

Middlebox Consolidation and Management Architectures
CoMb [137] 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 [138]. 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) [139] 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. Slick [140] introduced a consolidated control plane for middleboxes by sharing common elements among multiple middleboxes. Apart from consolidating packet processing operations, Slick [140] also introduced a programming language based on Python that allows a programmer to compose distributed, network-wide service chains driven by a controller, while being able to address placement requirements of these chains.

In 2015, an advanced NFV framework called E2 [32] was proposed by Palkar et al. Inspired by data analytics frameworks, such as the Apache Hadoop [141], 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.

Bremler-Barr et al. applied the SDN control and data plane separation paradigm to the OpenBox [31] 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 that constitute the 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 service 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 service chain.

In 2018, Barbette et al. took this idea to the next level by proposing MiddleClick [142, 143]; a set of low-level solutions for building NFV data planes with unified abstractions for TCP session management and flow classification. Barbette et al. propose a per-session, per-NF "scratchpad", which acts as a piece of memory to store and quickly lookup NF state information. Tom Barbette's doctoral thesis [144] provides more information about MiddleClick along with other relevant optimizations for NFV service chains.

3.4 Industrial Efforts in SDN and NFV

Since the introduction of the OpenFlow [13] protocol in 2008, SDN adoption has gained momentum, as traditional network designs have gradually been replaced by SDN-enabled ones in campus networks [14], wide area networks [15, 16, 17], and datacenters [18, 19]. P4 has also been commercialized. Barefoot networks released the Tofino switch [145], a high performance P4-programmable Ethernet switch which operates at several (up to 6.5) Tb/s. Netcope P4 [107] is an FPGA-based Smart NIC, the chip of which can be re-programmed from a web service that manages the transformation of a P4 description into a firmware bit stream.

European Telecommunications Standards Institute (ETSI) has been driving NFV standardization during the last 5 years [146]. ETSI's specialized open source management and orchestration group (OSM) [147] uses an open implementation of the current NFV standards, based on a generic framework for managing compute, storage, and network resources called OpenStack [148]. CORD [149] and OPNFV [150] also use OpenStack. The former re-architects the central office as a datacenter, while the latter facilitates the interoperability of NFV components across various open source ecosystems.

3.5 Scheduling Techniques in SDN and NFV

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

SDN

Inspired by the programmable flow tables of the SDN paradigm, Sivaraman et al. [151] 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. [152] 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.

NFV

Along the same lines, a technical report from Rizzo et al. [153] 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 per second) scheduling decisions, while (*ii*) packet transmissions can run in parallel exploiting multi-core capacities.

The latest work in this area is NFVnice [154]; a scheduler that coordinates the execution of multiple NFs in an NFV service chain. NFVnice was published after our contribution in this area (see Chapter §6), focusing on increasing the throughput of NFV service chains by adjusting their scheduling. In contrast, the scheduler proposed in the earlier stage (i.e., licentiate) of this thesis [155] realizes NFV service chains with lower predictable latency. A network operator could use our approach for latency and jitter-sensitive service chains and NFVnice for throughput-sensitive service chains.

3.6 Performance Monitoring Tools

The NFV vision is to turn the hardware-based network processing into software running on 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 [156] 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, UDP, TCP), file system, process, signal handling, and context switching measurements.

Similarly, Intel's memory latency checker [157] 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 [158] and Perf [159], 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 [160], callgrind's KCachegrind tool [161] (based on valgrind), and Intel's Performance Monitoring Unit (PMU) [162] are popular tools that track applications' cache utilization. Likewise, likwid [163] allows programmers to measure the performance of either an entire application or specific parts of an application. Finally, DProf [164] 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, 12, 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. §4.3 describes the challenges associated with the problem. §4.4 formulates a research question, which is boiled down to a set of hypotheses in §4.5, and a research methodology used to tackle the research problem in §4.6.

4.1 The Problem

Several studies have shown that 10 and 40 GbE are widely used in today's datacenter networks, while a 100 GbE era will be established by 2020 [24, 25]. To this end, industry leaders, such as Cisco [165], have already replaced infrastructure for 10 Gbps networks with 40 GbE, providing 4x more capacity. QLogic and Broadcom demonstrated end-to-end interoperability between 25 and 100 GbE [166]. Link speeds in Microsoft's datacenters have exhibited a 50-fold increase between 2009 (1 Gbps) and 2017 (50 Gbps) [167], as shown in Figure 4.1. According to Microsoft Azure, these datacenters will soon migrate to 100 GbE solutions. Optical transceivers at 100 Gbps are also planned for upgrading Facebook's datacenters fabric [168], where 100 Gbps switches are already in-place [169].

Figure 4.1 shows that the relative increase in transistor counts has not followed the same rapid increase since 2009. However, the study in §3.2 shows that NFV service chains heavily rely on CPU performance. Despite the efforts to eliminate the overhead of traversing the network stack of commodity OSs (see §3.2) and to consolidate packet processing operations (see §3.3), it is still hard for NFV service chains to keep up with these emerging links speeds. Moreover, the processor-memory gap explained in §1.5 poses additional challenges. Consequently, this section aims to achieve the following goal:

Goal: Study the impact of the increasing links speeds on the performance of today's NFV service chains, by measuring the performance of state of the art network stacks 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).

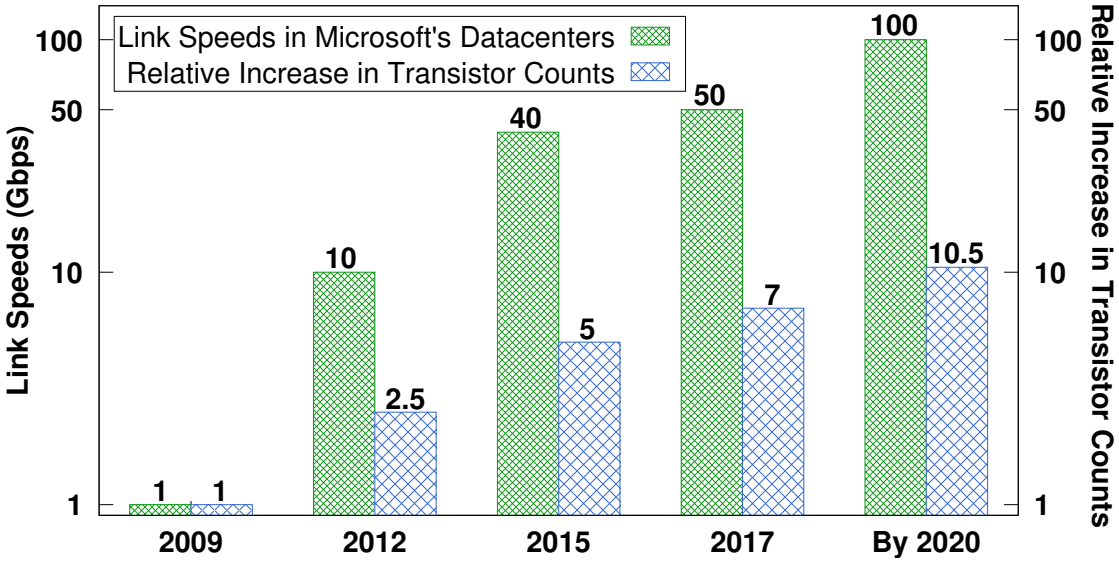


Figure 4.1: Link speeds' increase in Microsoft's Azure datacenters [167] along with the relative increase in transistor counts, following a more conservative approximation of Moore's Law, since 2009.

To achieve this goal, two identical machines are used, each with a dual socket 16-core Intel® Xeon® CPU E5-2667 v3 clocked at 3.20 GHz [170]. The input traffic rate is gradually increased from a sub-Gbps level (i.e., 0.57 Gbps) to 2.5, 40, and 100 Gbps. For experiments with input traffic up to 40 Gbps, the machines are directly interconnected using two dual-port 10 GbE Intel 82599 ES NICs. For experiments at 100 Gbps, the same machines are directly interconnected using two 100 GbE Mellanox ConnectX-4 MT27700 NICs. In both of these testbeds, 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 service chains are deployed on top of a state of the art NFV framework called Click [42]. This study is also complemented with measurements of two additional state of the art platforms, called OpenBox [31] and E2 [32], in §4.1.2.2. 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 4.2, 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) [97] 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 million frames at an input rate of 0.82 Mpps. The frame size is 64 bytes without counting the trailing frame 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 4.2 show the end-to-end latency of each service chain versus the service chain's length.

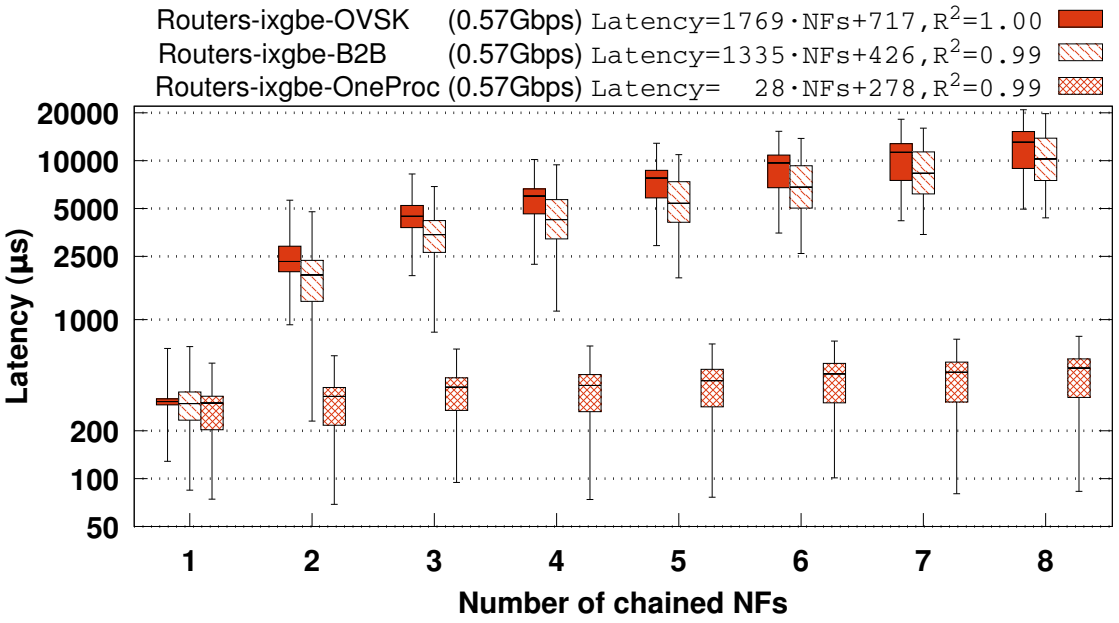


Figure 4.2: End-to-end latency (μs), plotted on a logarithmic scale, versus the service chain's length for user-space Click routers based on the native Linux network driver. The service chains run (*i*) as individual processes in containers interconnected with either OVSK, or B2B and (*ii*) in a single process. The service chains run in a single core and in the case of the OVSK service 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 service chain with 2 NFs.

To quantify the latency of each service chain, we fitted the median latencies of the boxplots, leading to the equations shown in the legend of Figure 4.2. The fitting starts from the service chains with 2 NFs. Based on these equations, each additional router in the OVSK service 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 service 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 service chain. Specifically, 28 μs of additional latency is imposed by these service chains, for each additional NF; this latency is roughly 1.5 orders of magnitude lower than the multi-process service chains. It is also worth noting that the 99$^{\text{th}}$ percentile of the latency increases to almost 1 ms for a single-process service chain with 8 NFs.

Observation 1: Service chains realized as multiple processes face severe performance degradation with increasing service 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, excessive overhead was observed when realizing user-space service chains, using unmodified Linux network drivers. For brevity, we refer to these service chains as ixgbe-based service chains (since the native Linux network driver for the underlying Intel NICs is called ixgbe). The goal of this subsection is to assess whether state of the art NFV schemes can realize these service chains without performance degradation.

Recent state of the art efforts improved the performance of software-based NFs by utilizing fast network drivers, such as DPDK [28] or netmap [33]. Despite this fact, some of these efforts, such as ClickOS [111] and NetVM [112], 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 length of the service chain. These frameworks rely on hypervisors (i.e., Xen [113] for ClickOS and KVM [114] 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 [125], was deployed with the same

service 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 service chains comprise the fastest way of service chaining; therefore, the same single-process service chains are realized using FastClick. This service chaining is much faster than ClickOS or NetVM because the service chains run *natively*, thus eliminating the hypervisor costs. In these experiments (described in the following sections), each router has four 10 Gbps NICs and the service chains are replicated across 8 CPU cores on the same socket. Input traffic is distributed across these CPU cores using the RSS functionality of the underlying Intel NICs.

4.1.2.1 Low Packet Input Rates

This section demonstrates the performance of FastClick service chains using the DPDK state of the art network driver, when input load is low.

Service Chains' Deployment at 0.57 Gbps

To compare the DPDK-based service chains with the ixgbe-based service chains presented in §4.1.1, the same 64-byte long frames are injected into FastClick at the same rate (i.e., 0.57 Gbps). The top entry in the legend of Figure 4.3 shows the end-to-end latency of these service chains with an increasing number of routers.

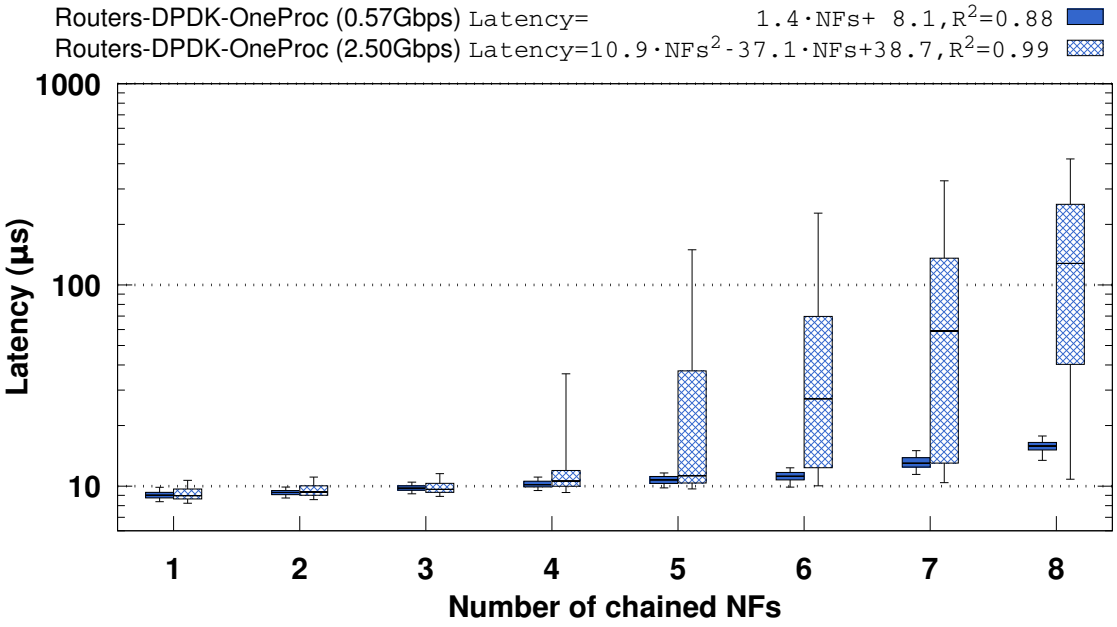


Figure 4.3: End-to-end latency (μs), plotted on a logarithmic scale, versus the service chain's length for user-space FastClick routers based on the DPDK network driver. The service chains run natively in a single process, replicated across 8 CPU cores on the same socket using RSS. The input traffic consisted of 64-byte frames at 0.57 and 2.5 Gbps.

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 4.3, an additional router in the DPDK-based service chain imposes only 1.4 μs of additional latency. Compared to the same service 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 4.2 confirms this observation.

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

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.

Service Chains' Deployment at 2.5 Gbps

Next, the input bitrate injected to the DPDK-based FastClick service chains shown in Figure 4.3 is increased by 5x (from 0.57 to 2.5 Gbps). The latency of the DPDK-based router service chains, under this increased rate, is quantified by the second legend (from top to bottom) in Figure 4.3. Despite the 5-fold increase in the input rate, the service 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 service chain of 8 routers requiring almost 500 μs to deliver packets through the service 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 service chains consist of rather simple NFs, such as routers, and the input rate is less than the maximum line-rate.

4.1.2.2 High Packet Input Rates

As 100 Gbps switches and NICs are starting to be standardized and deployed, maintaining high performance at the ever-increasing data rates is vital for the success of NFV. To examine the scalability of state of the art NFV frameworks at the current (40 Gbps) and emerging (100 Gbps) link speeds, two additional experiments are performed. The input rates in these experiments are 40 and 100 Gbps respectively, while the focus is now shifted; we study the ability of state of the art NFV service chaining frameworks to maintain high throughput at these challenging input rates.

Deploying Service Chains at 40 Gbps

The first experiment targets a 40 Gbps deployment, which is a typical link speed in datacenter networks today [167], using the same FastClick router service chains and the same frame size. Note that the maximum achievable throughput of the testbed for the 64-byte frame size is 31.5 Gbps, since this is the limit of the underlying Intel NICs (also reported by Barbette et al. [125]). Figure 4.4 shows the throughput of the router chains.

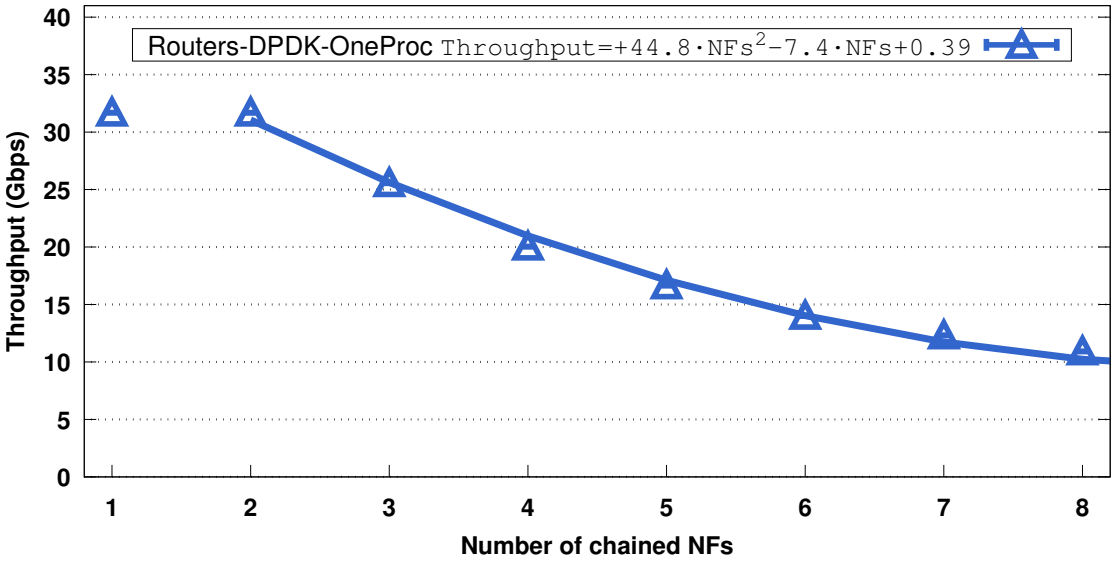


Figure 4.4: Throughput (Gbps) versus the service chain's length for user-space FastClick routers based on the DPDK network driver. The service chains run natively in a single process, replicated across 8 CPU cores on the same socket using RSS. The input traffic consisted of 64-byte frames at 40 Gbps (10 Gbps per NIC with 4 NICs in total). The fit to the median throughput, stated in the legend, begins from the service chain with 2 NFs.

In this experiment FastClick can operate at the maximum throughput only for a service chain of 1 or 2 routers. The equation fitted to the median throughput for each service chain shows that there is a quadratic throughput degradation that results in a service chain of 8 routers achieving only 10 Gbps of throughput. This degradation is nearly a third the line-rate throughput of this testbed and one would expect an even greater degradation if more complex NFs are chained together.

Deploying Service Chain at 100 Gbps
Router service chains with 3 or more NFs exhibited performance degradation when tested at 40 Gbps. The third experiment aims to test the limits of state of the art NFV frameworks at the challenging input rate of 100 Gbps, using a more realistic service chain. The first NF of the service chain under test is a router with a large routing table of 3117 entries. These entries derive from the analysis of 4 million packets from a real campus trace, obtained from University of Liège. The router is chained with two stateful NFs: a NAPT and an LB that implements a flow-based round robin policy. In this experiment, the total CPU capacity (i.e., 16 CPU cores) of the server is exploited.

The state of the art frameworks under test are OpenBox [31] and an emulated version of E2 [32]. OpenBox is based on FastClick and uses RSS to dispatch input packets to the available CPU cores. To emulate E2 a dedicated CPU core (i.e., core 1) is used to dispatch packets to the remaining CPU cores of the server (i.e., 2-16), where the NFs of the service chain are executed. Figure 4.5 shows the throughput of these systems versus the number of available CPU cores. To quantify the maximum attainable throughput of this testbed, a simple RSS-assisted forwarding NF is used, denoted as "Hardware Limit RSS" in Figure 4.5. This NF reads input frames from the Mellanox 100 GbE NIC and immediately emits these frames back without performing any processing.

The results in Figure 4.5 show that both state of the art systems exhibit a slow but linear increase in throughput with an increasing number of CPU cores. A linear regression on the medians between 1 and 12 CPU cores (the emulated E2 starts from 2 CPU cores) shows that the throughput of OpenBox increases by 5.37 Gbps with each additional core, while the throughput of the emulated E2 increases by 4.91 Gbps per core. However, adding more than 12 CPU cores does not bring further performance gains. Specifically, the throughput of OpenBox plateaus around 67 Gbps, while the performance of the emulated E2 drops from 53 to 41 Gbps. In contrast, the black crosses in Figure 4.5 show that the maximum attainable throughput of this testbed is 76 Gbps, with this result achieved using only 4 CPU cores.

From the last two experiments, which stressed state of the art NFV frameworks 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.

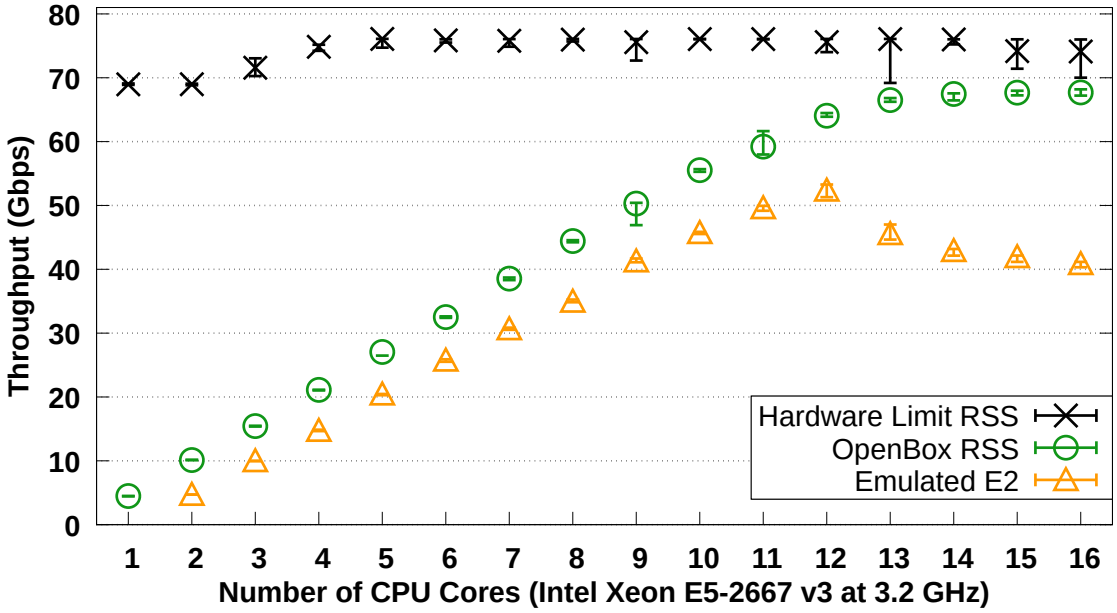


Figure 4.5: Throughput (Gbps) of a stateful service chain (Router→NAPT→LB) with increasing number of CPU cores at 100 Gbps. Comparison of: (*i*) OpenBox [31] with RSS and (*ii*) a software-based dispatcher emulating E2 [32]. "Hardware Limit RSS" shows the forwarding speed of the server (i.e., no service chain) using RSS as a traffic dispatcher.

4.1.3 Problem Definition

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

Problem Definition: State of the art NFV frameworks, exhibit *serious performance degradation*, when realizing service chains using either (*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 Causes

The experimental process in §4.1 quantified the performance degradation of state of the art NFV systems. This section highlights the reasons for the observed performance degradation.

4.2.1 The Consequences of Inaccurate Packet Dispatching

In an NFV service chain, packets move from one physical or virtual server (hereafter simply called server) to another to realize a programmable data plane. The servers themselves are predominantly multi-core machines. Different ways of structuring the NFs exist, e.g., one per physical core or using multiple threads to leverage multiple cores within each NF. Network functions range from simple stateless ones to complex functions, such as DPI, and potentially stateful (e.g., proxy) functions. Regardless of the deployment model and NF types, every time a packet enters a server, a fundamental problem occurs:

> Which core within a multi-core machine is responsible for handling this packet? This problem reoccurs every step of the service chain and can cause costly inter-core transfers.

To identify the core that will process an incoming packet, NFV frameworks typically only examine the header fields. Here, there is a big mismatch between the way modern servers are structured and the desired packet dispatching functionality. Figure 4.6 shows three widely used categories of packet processing models in NFV, as previously discussed in Chapter 3, and specifically in §3.2.

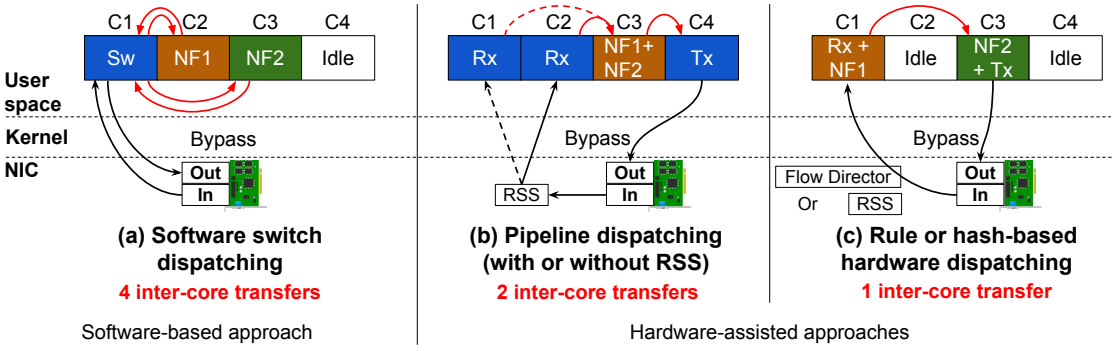


Figure 4.6: State of the art packet processing models either have too many inter-core packet transfers or load balancing problems due to load imbalance and/or idle cores.

Software-based Dispatching
The first category (see Figure 4.6a), augments the weak programmability of current NICs with a software layer that acts as a programmable traffic dispatcher between the hardware and the overlay NFs. E2 [32], with its software component called SoftNIC [171], falls into this category. SoftNIC requires at least one dedicated CPU core for traffic dispatching and steering (see Figure 4.6a), while the NFs run on other CPU cores. Earlier works, such as ClickOS [111] and NetVM [112], also used software switches on dedicated cores to dispatch packets to VMs, but without the flexibility of E2.

Pipeline Dispatching (with or without RSS)
Rather than having a shim layer between the NFs and the NICs to select the next hop in a service chain, the second category of packet processing models (see Figure 4.6b) involves a pipeline of reception, processing, and transmission threads, each on a different (set of) core(s). If more than one reception core is required, this model uses RSS [102] as described below. For example, OpenNetVM [115], Flurries [117], and NFP [118] (a parallel version of OpenNetVM) fall into this category. Similar to E2, these works introduce programmability by augmenting the reception and processing parts of the pipeline with traffic steering abilities.

Hardware-assisted Dispatching using RSS or Flow Director
The last category of packet processing models relies on two hardware features provided by a large fraction of NIC vendors today. First, RSS uses a static function to dispatch traffic to a set of CPU cores by hashing the values of specific header fields. Second, NICs can be programmed via a vendor specific "match-action" API to dispatch traffic to specific cores (e.g., Intel's Flow Director [103]). Unlike all previous models, these approaches *do not* require dedicated dispatchers, hence they offer higher performance. OpenBox* [31], FastClick [125], our earlier work SNF [29], and RouteBricks [122] use RSS, while CoMb [137] uses Flow Director.

Summary
None of these schemes guarantee that the core receiving the incoming packet will be the one processing it. Flow hashing as in RSS can introduce serious load imbalances under skewed (e.g., heavy flows with the same hashes) workloads. Flow Director permits explicit flow affinity, but suffers from the limited classification capabilities of today's commodity NICs. When there is a mismatch, the packet is handed off to the correct core. However, this requires transferring the packet via Dynamic Random Access Memory (DRAM) or LLC to the target processing core. Even when using the LLC this a slow operation, as the LLC takes several tens of cycles even for a cache hit!

*Originally, OpenBox was built on top of Click using Linux-based I/O. To fairly compare it against our work, OpenBox was accelerated using FastClick's DPDK engine and RSS [172].

The available time to process 64-byte packets at 100 Gbps is only *5.12 ns/packet* (see Figure 4.7). However, existing NFV systems require at least *one inter-core transfer per packet*, either via LLC or DRAM. This transfer alone takes several nanoseconds (up to 14.3 ns for LLC and up to 71.7 ns for DRAM on an Intel Xeon E5-2667 v3 according to Figure 6.3 on page 78).

Therefore, there is a clear mismatch between the processing requirements of high-speed networks and the way that state of the art NFV systems process packets. Our earlier work [27] demonstrated that dramatic speedups (i.e., several times lower latency and orders of magnitude lower latency variance) occur if the correct core receives the packet straight from the NIC and the packet remains in the core-specific cache(s).

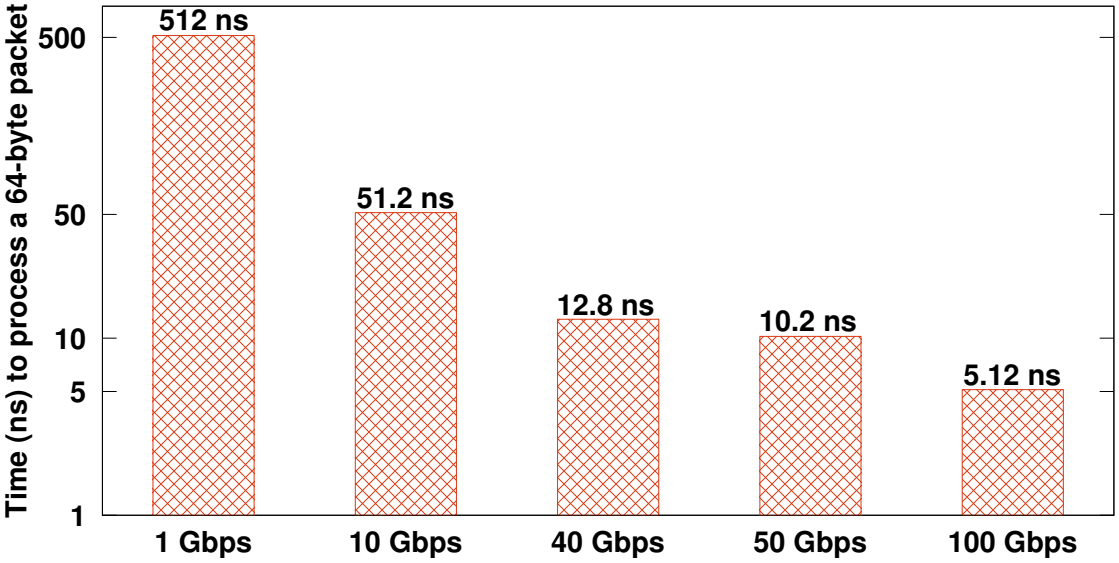


Figure 4.7: Available time in nanoseconds, plotted on a logarithmic scale, to process 64-byte long frames at increasing speeds (1-100 Gbps). It is extremely challenging to realize packet processing at 100 Gbps on a commodity server, as the available time is only 5.12 ns/64-byte packet.

4.3 The Challenges

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

4.3.1 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 researcher is diverted from the performance problem of the commodity network driver and (*ii*) popular services to date, offered by e.g., Amazon [173], 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: This challenge is addressed in Chapter 6, by introducing an NFV profiler [27]. 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.

4.3.2 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 work [34, 28, 110, 33]. 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 §3.5, Sivaraman et al. [151] revised the abstractions of modern switches, by proposing programmable packet scheduling techniques, while Mittal et al. [152] 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 efforts above inspired us to study scheduling schemes for NFV service chains. However, in contrast to [151, 152], a chain of NFs might require a global scheduler to make service 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 [174]. 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: This challenge is addressed in Chapter 6, by proposing an NFV service chain scheduler [27]. 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.

4.3.3 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 §3.3, several NFV consolidation frameworks have been proposed since 2012 [137, 139, 138, 140, 32, 31], mainly targeting reducing this redundancy. However, none of these efforts managed to *entirely* eliminate the redundancy of the operations within the service chain.

For example, Slick [140] 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 service 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 5G [175, 176, 177] communications. Specifically, meeting the target latency of 1 ms set by various 5G 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 service chain (i.e., ideally achieving a single hop) and must eliminate all redundancies across the internal operations of the service 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 a service 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: This challenge is addressed in Chapter 7, by proposing a highly-optimized NFV service chain synthesis framework [29]. Our approach minimizes the number of elements that apply read, write, and discard operations on packets, allowing long, stateful, and complex service chains to be realized at 40 Gbps.

4.3.4 Challenge 4

Even highly-optimized service chain implementations, using our synthesis approach, might exhibit performance degradation. This can happen for many reasons. First, when a packet processing element of a service chain is too large to fit into a server's core-specific cache(s), thus introducing costly LLC and/or DRAM transfers. A common example is a large packet classifier with several thousands of entries (e.g., a firewall or core router), which is typical in e.g., ISP networks. Second, when the packet processing operations of a service chain are not properly mapped to the underlying system resources, such as CPU cores and their caches. This can be problematic when the input load exceeds a threshold. For example, some state of the art NFV approaches use dedicated CPU core(s) to perform packet I/O, which then transfer(s) batches of packets to another (set of) CPU cores for processing. These transfers might not affect the throughput of a service chain at 10 Gbps, but greatly compromise its performance at 100 Gbps (see Figure 4.5).

Challenge 4: In these cases, network operators require means to exploit all available resources. State of the art NFV approaches do not offer such a capability, despite the fact that modern networked systems are comprised of programmable network devices, such as OpenFlow switches and NICs. We believe that it is vitally important to leverage all available hardware in order to realize NFV service chains at the emerging, and extremely challenging, high-speed networks of 100 Gbps and beyond.

Approach to Challenge 4: This challenge is addressed in Chapter 8, by proposing an NFV service chaining platform that operates at the true speed of the underlying commodity hardware [30]. The proposed approach exploits the available programmable hardware to perform early traffic classification and

tagging, which is then used by commodity servers to perform hardware-based dispatching to the correct CPU core, while ensuring zero inter-core communication among the components of a service chain. With commodity hardware, the proposed approach can perform DPI at 40 Gbps and realize stateful network functions at the speed of a 100 GbE network card on a single server. Our approach has 2.75-6.5x better efficiency than OpenBox, a state of the art NFV system, while ensuring key requirements such as elasticity, fine-grained load balancing, and flexible traffic steering.

4.4 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, causes, and challenges (see §4.1, §4.2, and §4.3), it is time to formulate the research problem that this thesis project tackles. That said, we state the 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?

4.5 Hypotheses

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, complexity, and processing model of the service chain.

In this thesis we use the Socratic method [178, 179], 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, 7, and 8), (*iii*) formulate an alternative hypothesis (shown next), and (*iv*) evaluate our solutions to provide evidence that show that H_0 leads to contradictions, thus H_1 holds (see §7.5 and §8.2).

The alternative hypothesis we want to support is:

> H_1: Some service chains can be realized without their performance deteriorating despite the length and complexity of the service chain, when using an appropriate processing model.

4.6 Research Methodology

This thesis project follows a quantitative research approach as specified by John Creswell in [180]. In short, this approach is based on the closed loop depicted in Figure 4.8. 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.

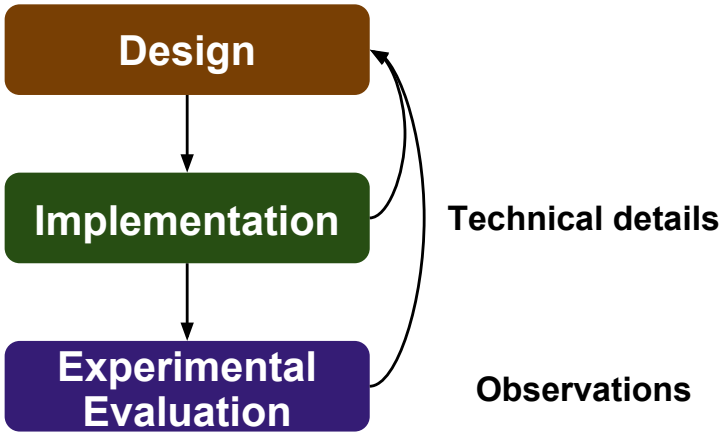


Figure 4.8: The research methodology followed by this thesis.

Chapter 5

Experimental Setup

This chapter describes the experimental setup followed to deploy, run, and measure the performance of chained NFs. At a high level, the testbed used throughout the experiments of this thesis is shown in Figure 5.1. Depending on the experiment, the total link speed of this testbed can either be 10, 40, or 100 Gbps. This capacity is provided by using either 1x10, 4x10, or 1x100 Gbps NICs in each machine as explained below. Machine 1 generates and sinks bi-directional traffic (using different cores), while machine 2 acts as the NFV host, where NFV service chains and tools are deployed.

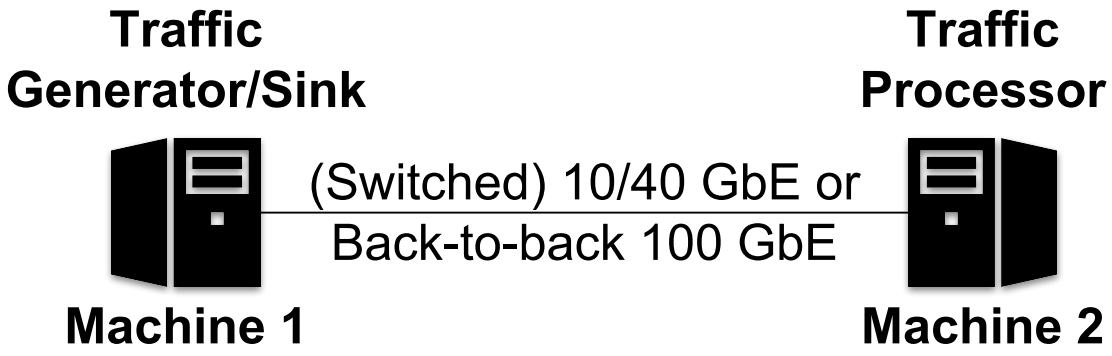


Figure 5.1: The experimental setup used throughout this thesis. Machine 1 generates and sinks bi-directional traffic, while machine 2 realizes the chained packet processing. The total link speed of the testbed is 10, 40, or 100 Gbps depending on the number and type of NICs being used. Deployments at 10 or 40 Gbps use either back-to-back interconnects or an OpenFlow switch between the servers. Deployments at 100 Gbps were only possible using back-to-back interconnects between servers.

Servers' Hardware Specifications
The testbed consists of six identical servers, each with a dual socket 16-core Intel®Xeon®CPU E5-2667 v3 [170] clocked at 3.20 GHz. The cache sizes are: 2x32 KB L1 (instruction and data caches), 256 KB L2, and 20 MB L3, with 128 GB of main memory (i.e., DRAM) clocked at 2.133 MHz and distributed across 8 slots. Hyper-threading is disabled.

Testbed at 10/40 Gbps
During the licentiate phase of this thesis [155] each machine had two dual-port 10 GbE Intel 82599 ES NICs with total capacity 40 Gbps. The OS was the Ubuntu 14.04.3 distribution with Linux kernel v.3.13. In machine 2, an entire CPU socket was 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. For the majority of the experiments in that thesis, two out of the six machines were used as shown in Figure 5.1.

Testbed at 100 Gbps
During the doctoral phase of this thesis the OS on all of the machines was upgraded to the Ubuntu 16.04.2 distribution with Linux kernel v.4.4. Also, three of the machines in this testbed were equipped with an additional 100 GbE Mellanox ConnectX-4 MT27700 NIC. These NICs allow us to further stress the performance limits of NFV service chains, thus better evaluate the relevance of our contributions with respect to such an emerging and challenging data rates.

Switching Infrastructure
Back-to-back connectivity is not the most common method of interconnecting servers. Network operators typically use a switching infrastructure to interconnect multiple servers together. This allows them to scale the network and to perform more advanced traffic steering. Moreover, modern switches are also programmable, allowing parts of the packet processing operations to be offloaded, thus reducing the processing demands at the servers. Two programmable switches are used in some of the experiments in this thesis: a NoviFlow 1132 switch and an HP 5130 EI Switch [181] with software version S5130-3106. The former switch is a powerful multi-port 10 GbE OpenFlow switch used to evaluate two of our contributions in §7.5.4.2 and §8.2. The latter switch is an inexpensive hybrid (i.e., legacy and OpenFlow-based) switch used to assess how hardware diversity affects the third contribution of this thesis (see §8.2.5.2). §5.2 describes how these switches are used in the testbed.

Next, the two main components of the testbed are described. The traffic injection and measurement module is described in §5.1, while §5.2 describes the traffic processing part of the testbed. The low-level configuration of the testbed is included in this thesis as Appendix A.2.

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 [182] 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.

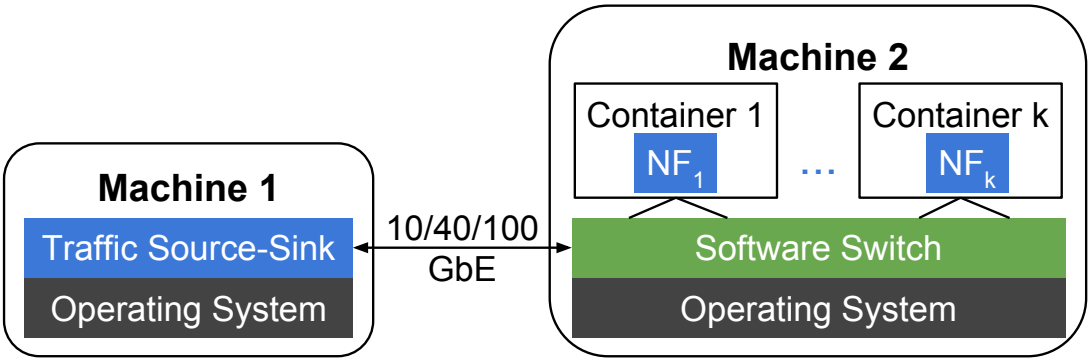
Latency mode is used to measure the end-to-end time required to send traffic from machine 1, traverse the NFV deployment under test in machine 2, and receive the traffic back to machine 1. Click is used as a traffic source and sink for this purpose. Before the packets leave the traffic generator, they 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, both timestamps are stored in memory, and end-to-end latency statistics are dumped to a file after the end of an experiment.

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 or UDP, or even replay traces from pcap files. After the traffic generator has sent all the required packets, it produces a report that gives 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 one can assess if a service chain exhibited throughput degradation or packet loss.

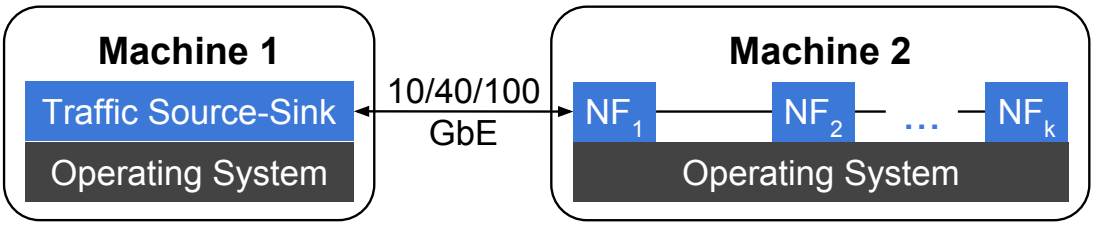
5.2 Traffic Processor

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*. As shown in Figure 5.2, three different service chains are considered for the tests of this thesis.

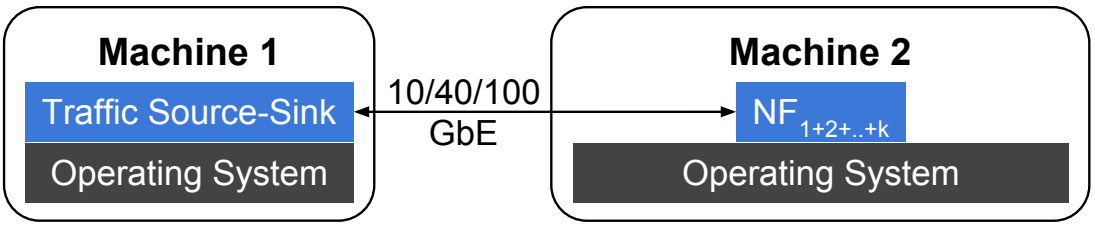
*Each experiment reports the exact version of the DPDK driver being used.



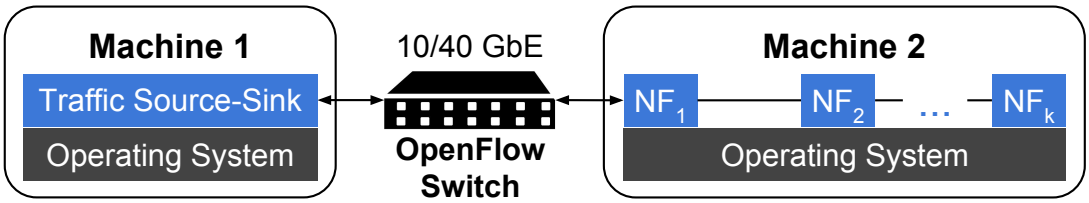
(a) Multi-Process NF chain, each running in a different container, on top of a software switch.



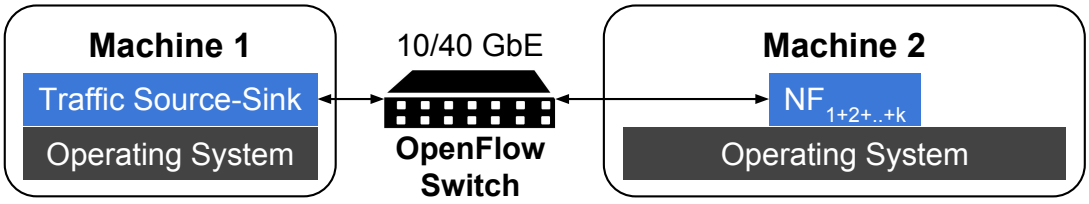
(b) Single-Process NF chain running natively.



(c) Synthesized single-process NF chain running natively.



(d) Single-Process NF chain running natively, preceded by an OpenFlow switch.



(e) Synthesized single-process NF chain running natively, preceded by an OpenFlow switch.

Figure 5.2: Five deployment types for chained NFs used throughout this thesis.

The first service chain deployment type, shown in Figure 5.2a, runs on top of a software switch. In this case, 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 service chain. Each NF is a separate process, thus multiple containers are chained as shown in Figure 5.2a. This type of service chain is called a "Multi-Process" service chain.

To avoid (*i*) the I/O communication cost among NFs in a service chain and (*ii*) the overhead of running OS-level virtualization using containers, the entire service chain is realized in a single process that runs natively. This type of service chain deployment is shown in Figures 5.2b, 5.2c, 5.2d, and 5.2e, where the virtualization ("V") from NFV is removed, aiming for better performance. Figure 5.2b depicts a "Single-Process" service chain that runs exactly the same code as the "Multi-Process" service chain, but all in one native process. Similarly, Figure 5.2c shows a "Synthesized Single-Process" service chain that runs code equivalent in functionality with the other two chains, but from which the redundant operations have been removed. A way to synthesize service chains is proposed in Chapter 7 and exploited by an ultra high performance NFV platform in Chapter 8.

The service chains shown in Figures 5.2d and 5.2e are identical with the service chains shown in Figures 5.2b and 5.2c, but in the former two deployments machines 1 and 2 are interconnected via a NoviFlow 1132 OpenFlow switch instead of being interconnected back-to-back. Because this switch contains 10 Gbps ports, the total link rate of the deployments shown in Figures 5.2d and 5.2e is 10 or 40 Gbps, depending on the number of ports being used.

At the end of each service chain, output frames are sent out of machine 2, back to the origin machine. Next, we present how 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 run in software, hence we need a framework to facilitate the realization of NFs, while maintaining high performance. Click [42] is selected as such a framework, because it is among the most popular software-based packet processing architectures in academia and it is also used by industry [183].

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.

Unless stated otherwise, 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 [42]. 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 a service chain using 4 Intel NICs, 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 firewall requires an IPFilter element with one or more traffic filtering rules between CheckIPHeader and GetIPAddress elements. A NAPT NF requires an IPRewriter element with one or more traffic modification rules between the same two elements. To implement an LB that modifies the destination IP address of input packets in a round-robin fashion, a RoundRobinIPMapper element is used in tandem with an IPRewriter. Finally, to implement a DPI NF a library with payload inspection elements was ported to FastClick [172]. A regular expression matching element called RegexMatcherMP was used to detect malicious strings in input IP packets' payload [172]. Overall, the following five NFs were implemented in the course of this thesis:

1. IP router;
2. L3-L4 firewall;
3. NAPT;
4. L3 LB; and
5. DPI;

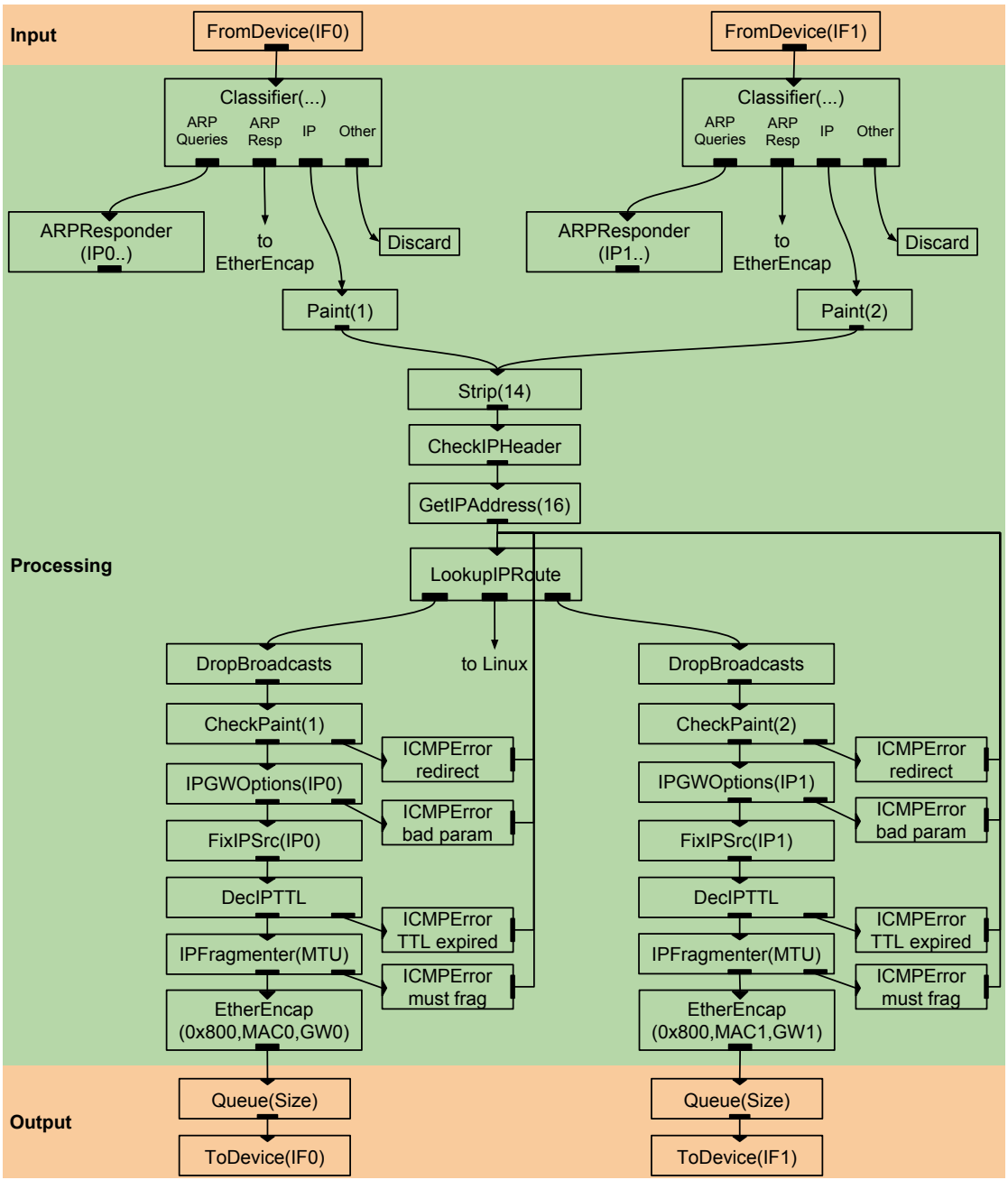


Figure 5.3: A Click implementation of an IPv4 router. I/O elements are shown with orange background, while processing elements are shown with green background.

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 §4.3.1. 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 [173]. In such an environment, even state of the art frameworks, such as ClickOS [111] and NetVM [112], 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" [27] (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 [184], 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 §4.3.2.

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 service chain. In contrast to earlier efforts [151, 152], SCC employs a global NFV scheduler to make service chain-level decisions, rather than an internal scheduler that executes local switch policies.

§6.1 gives 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.

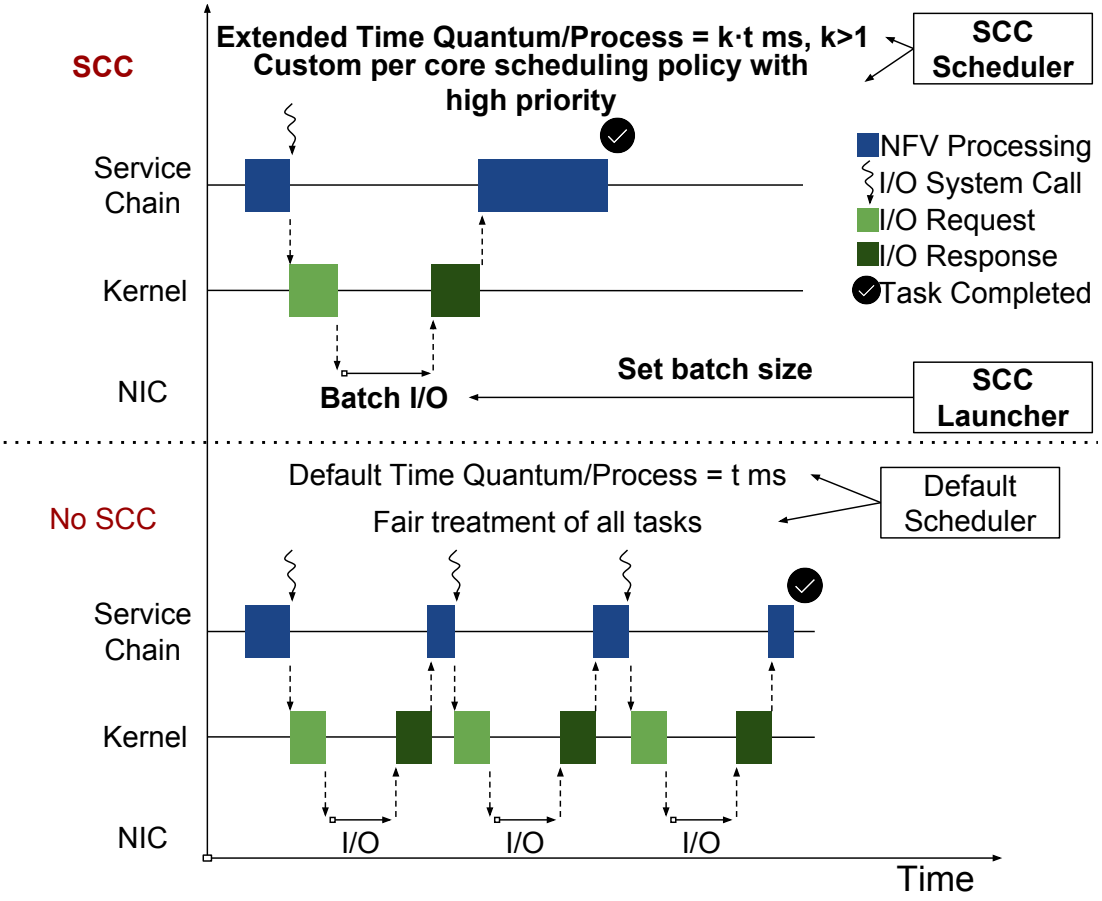


Figure 6.1: The SCC run-time combines (*i*) tailored scheduling for NFV service chains via the SCC Scheduler with (*ii*) fewer (but longer) user to/from kernel-space interactions by multiplexing I/O-related system calls via the SCC Launcher. SCC achieves faster completion time, hence lower latency, than 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 [185]) builds upon the popular FastClick NFV framework to address the first challenge of this thesis, introduced in §4.3.1.

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 §4.3.2.

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 make 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 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 service chains using the ixgbe network driver.	Not implemented for service chains interconnected with OVSK. 10-40% lower latency and 2x lower jitter for B2B interconnected service chains.
Scheduling benefits for user-space service chains using the ixgbe network driver.	30-300% lower latency and up to 40x lower jitter for service chains interconnected with OVSK. 10-25% lower latency and 2x lower jitter for B2B interconnected service 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 what parts of 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 [159, 158, 161] 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 [159] 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 when 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 [186]), 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.

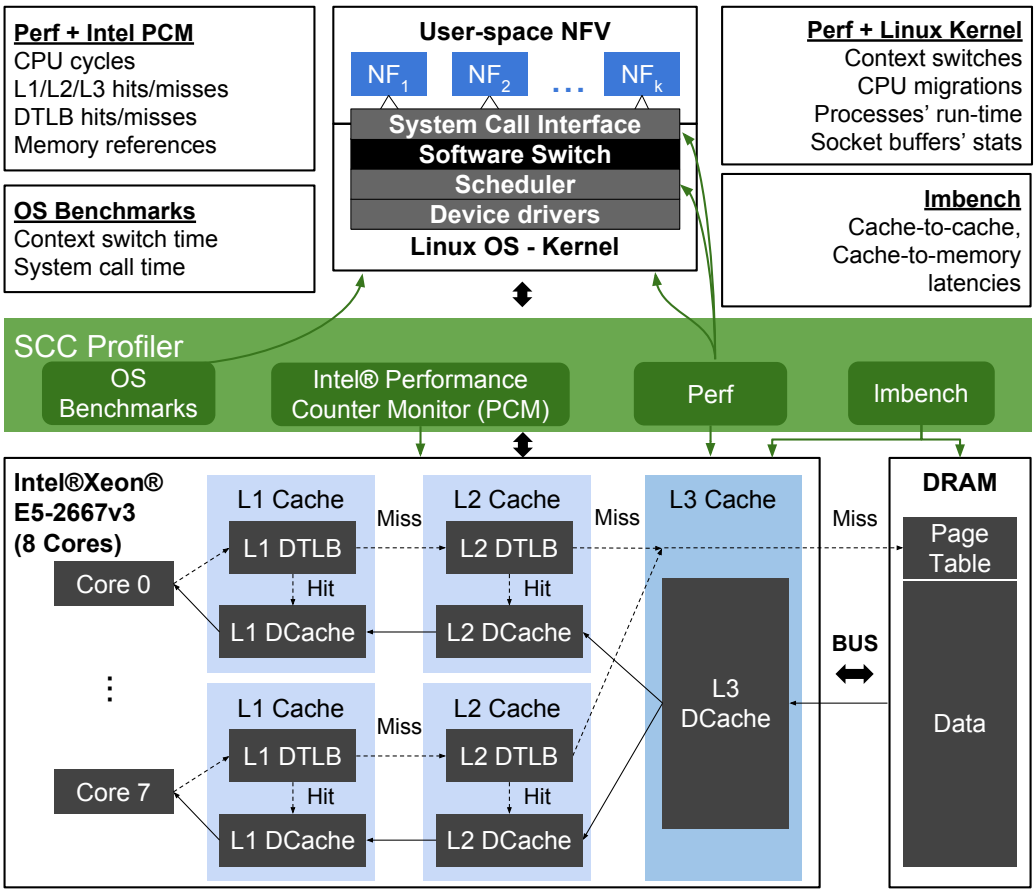


Figure 6.2: The SCC Profiler. lmbench measures the latencies to access each part of the memory hierarchy. The SCC Profiler combines the latencies from lmbench with (*i*) the hardware counters obtained by Intel's PCM and Perf and (*ii*) the software counters obtained by Perf and OS benchmarks, to measure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

default Completely Fair Scheduler (CFS) [187] (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 [37] technology as explained in §2.4.2. The LLC portion used by DDIO can be up to 10% of the LLC's capacity, which in our case is 10% of 20 MB, i.e., 2 MB [37]. However, as we will see from our measurements, fully exploiting the processor 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 higher throughput.

The SCC Profiler uses lmbench [156] 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) and will eventually fill all of the caches and part of the main memory, in order 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 latency. This way we emulate the latency to access 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 the 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 size. The reason behind this is that the hardware executes prefetch requests, in parallel with the current data processing, thus bringing cache lines from the next higher level store into the current cache before this data 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.

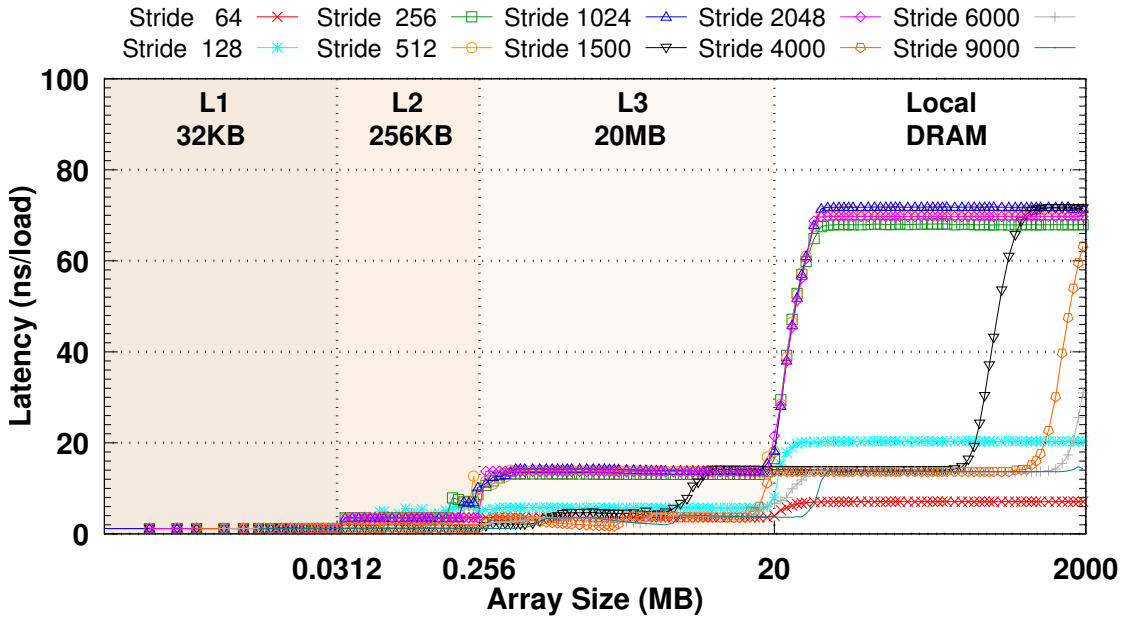


Figure 6.3: Latencies to access a progressively increasing array size (1 KB-2 GB) from different levels of the memory hierarchy versus different stride sizes in bytes for an Intel® Xeon® CPU E5-2667 v3 clocked at 3.2 GHz.

According to Larry McVoy and Carl Staelin [156], 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 [156], 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 [188]. More information is provided in Appendix A.1. Specifically, during the execution of an NFV service chain we acquire CPU core, L1, and DRAM events using Perf, while L2 and L3 events are fetched using Intel's Performance Counter Monitoring (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, a variety of latency factors are composed (as shown in Table 6.3). To calculate the total per packet latency, we sum the data access and address translation latency factors of each memory level (i.e., 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, a service chain might be comprised of multiple NFs, usually each NF is deployed as a separate process (e.g., considering a service 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 is likely to 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

*In this chapter we do not consider a single synthesized NF from a chain of NFs as we do in [29] (see also Chapter 7).

only 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, hence the counters are aggregate values collected during the execution of an experiment. The

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}

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 accessing main memory, as do the DTLB misses at the L2 cache, hence we can *infer* the number of main memory references by adding 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 [189] 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 and 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 [28] 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). As noted earlier, we chose a small frame size to impose more work on the CPU core, and hence greater 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.

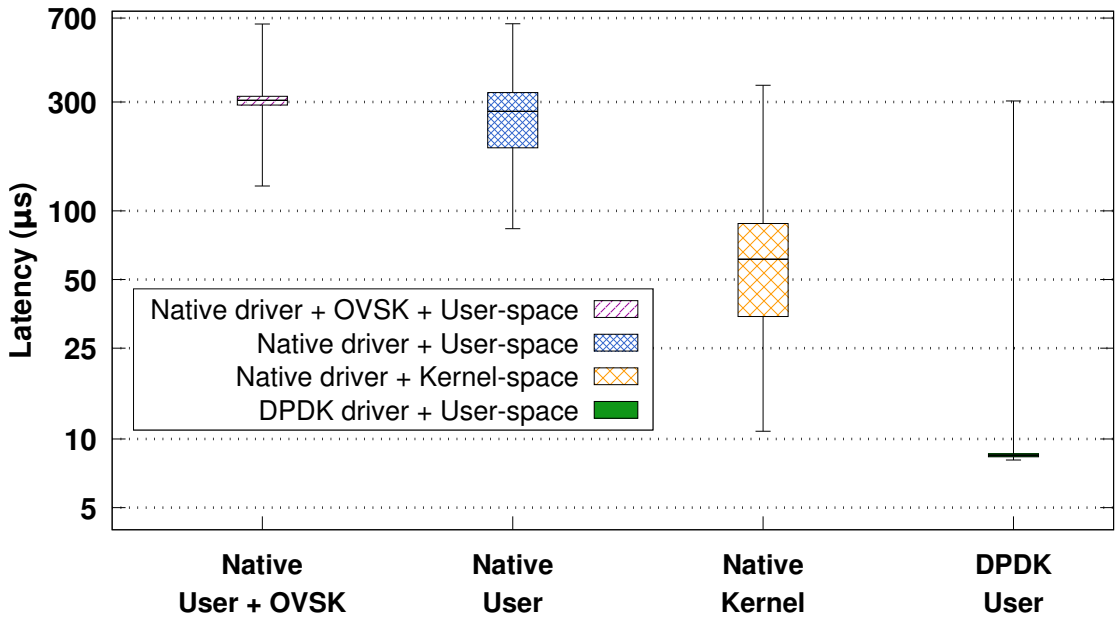


Figure 6.4: End-to-end latency (μs), plotted on a logarithmic scale, for 64-byte frames through four FastClick routers, each running in a different I/O context in a single core as stated in the legend. The input packet rate is 0.82 Mpps.

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 two 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 to which the queue points, 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 in the LLC, but eventually is evicted 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 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. In contrast, 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 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 (corresponding to 436.6 ns) for a receive and 1120 cycles per packet (corresponding to 350 ns) 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 (see also §6.3.1.2), and (*ii*) FastClick's user-space FromDevice element 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 reasons 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 in the number of the main memory references of the two routers (with 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. However, 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 [33], 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 service 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 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 service chain versus the service 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 service 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 (more details are provided in §6.3.2.2).

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

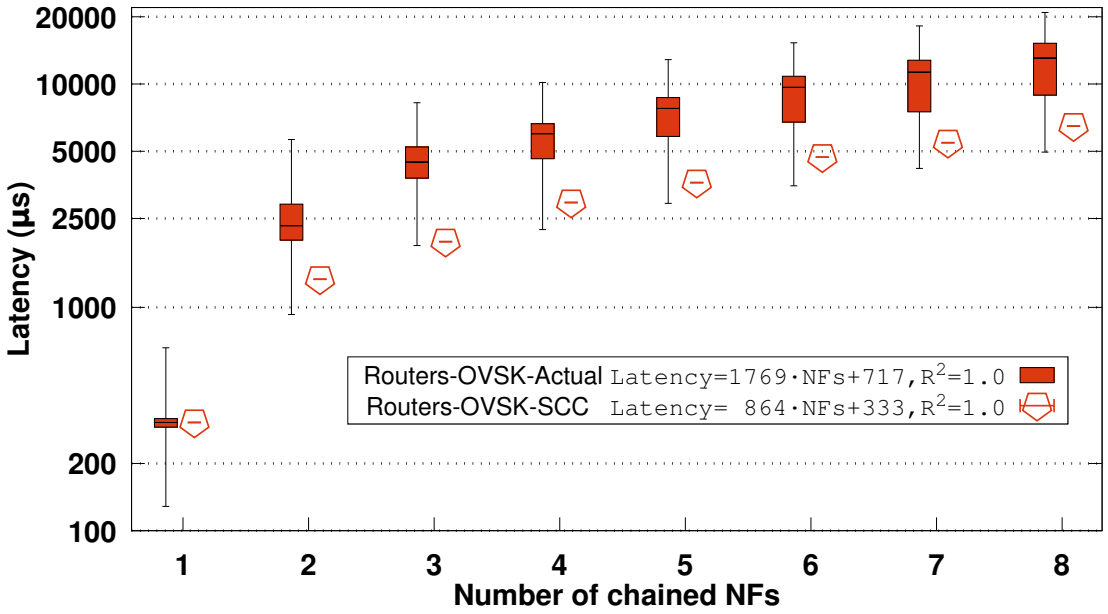


Figure 6.5: End-to-end latency (μs), plotted on a logarithmic scale, (i) measured at the traffic sink (boxplots) and (ii) calculated by the SCC Profiler (points), versus the service chain's length for user-space FastClick routers, running in containers on top of OVSK. The routers run in a single core 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 service chain with 2 NFs.

6.3.2.1 Root Cause Analysis

To visualize the latency of each service 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 service chain with 2 NFs. Based on these equations, each additional router in the service 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 service chain's latency. Looking at the number of memory references per packet, performed by each service chain, the equation that describes their dependence on the service 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 service 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 service 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 service 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 service chain's run-time. This metric reflects the overhead, added by the OS, to execute the service chain.

Table 6.5 depicts the values of these two metrics for four different service chain lengths, as obtained from the experiment shown in Figure 6.5. For a single router (i.e., with service 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 service chain's length leads to increasing waiting and scheduling overheads. The processor spends 75x more time waiting than actually running a service chain of 8 routers, while the time that this service chain is runnable but does not execute on the processor due to contention in the scheduler is almost 4x higher than the service 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.

Service 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.

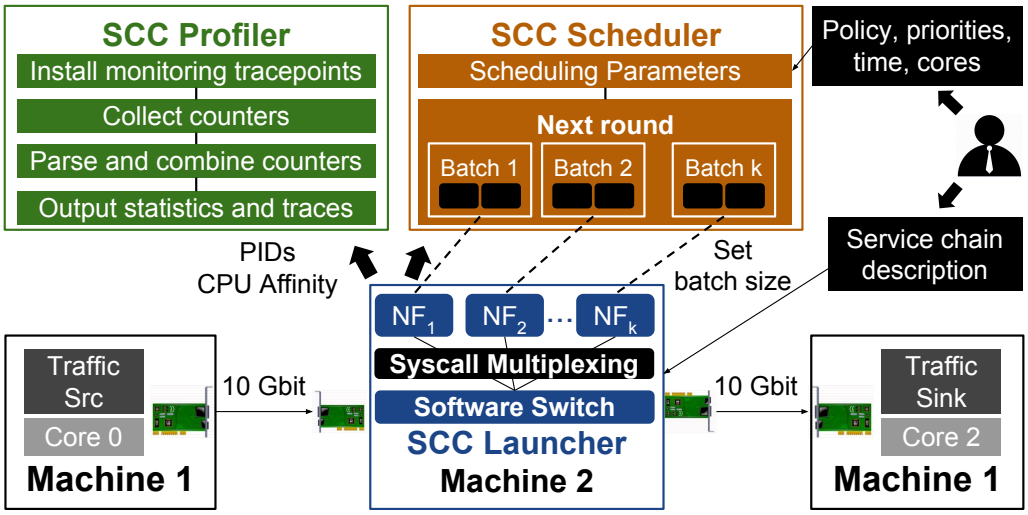


Figure 6.6: The Service Chain Coordinator in the context of our testbed. A system administrator inputs a service chain description and configuration (top right). The SCC Launcher identifies the service components, applies the requested configuration and deploys the service chain (bottom center). Using 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 service 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 service 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 service 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 service 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 service 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 service 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 uni-processor 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.).

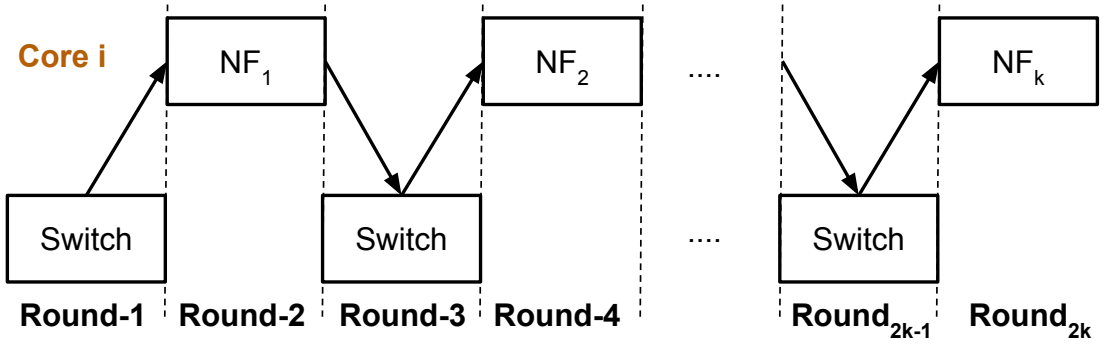
The system administrator might allocate a different core for the switch, to run the NFs in a dedicated core. In this case, the scheme depicted in Figure 6.7b can be used. Modern OSs maintain task queues per core, providing mechanisms to hand off the work from one task scheduler to another, hence better 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).

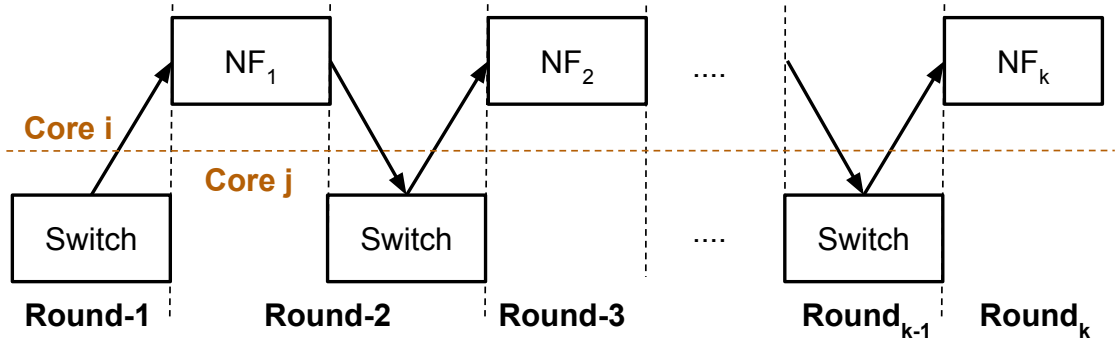
A key task of an NFV scheduler is to guarantee that the scheduling plan (e.g., as shown in Figure 6.7) requested by the system administrator will be executed meticulously. This is challenging because modern OSs employ task migration mechanisms to balance the load among all the available cores, when some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



(b) Example of a multi-processor 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

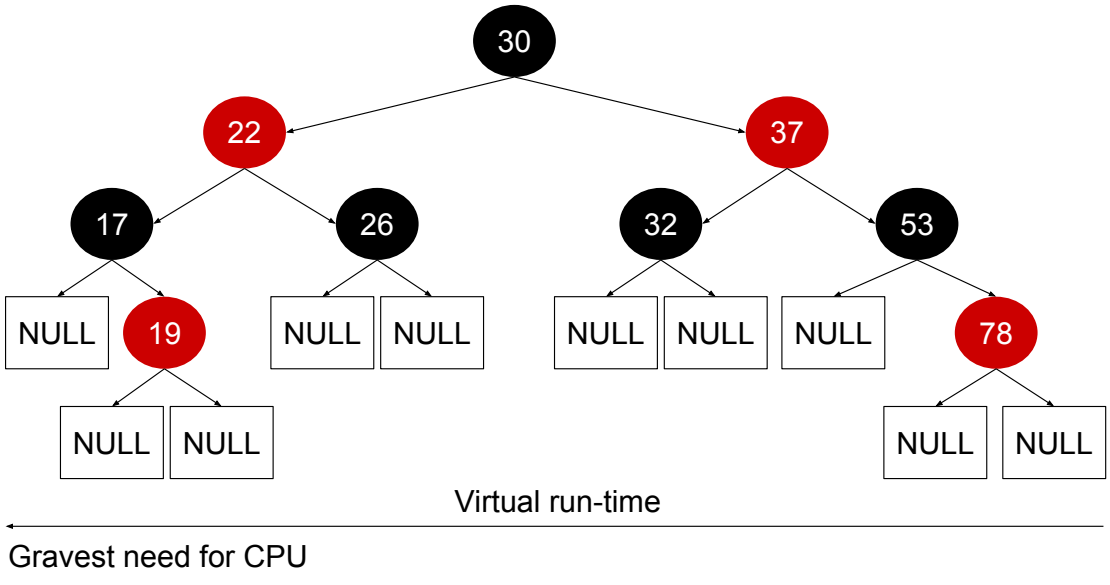


Figure 6.8: The Linux Completely Fair Scheduler's red-black tree data structure for selecting the next runnable task.

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 the interval [-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.

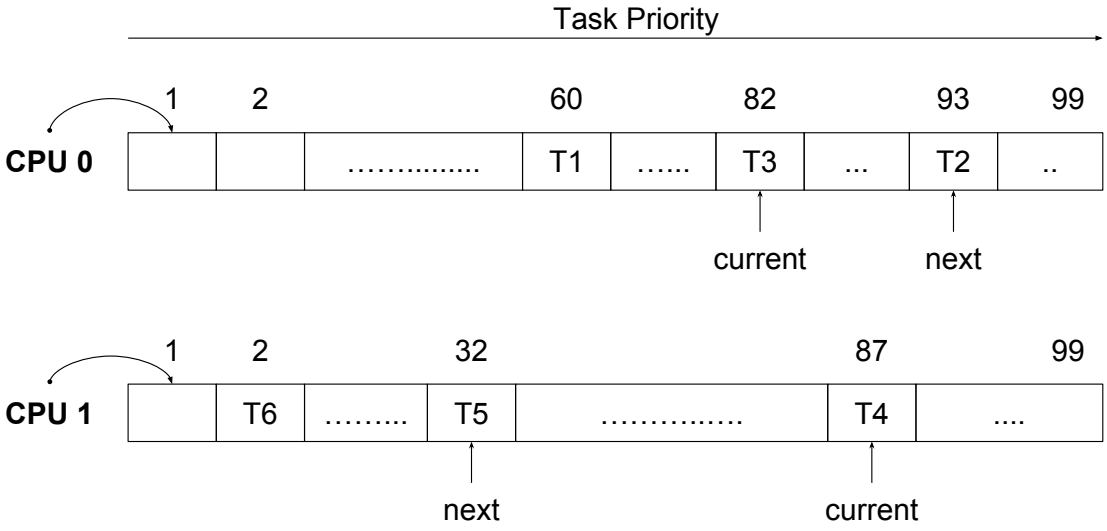


Figure 6.9: An example scenario of per processor real-time tasks queued on run queues. The current and next tasks to run are indicated by indices. The queues are static arrays and the indices denote the priority of each task.

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 scheduler 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 service 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 reference for comparing 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 are 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.*

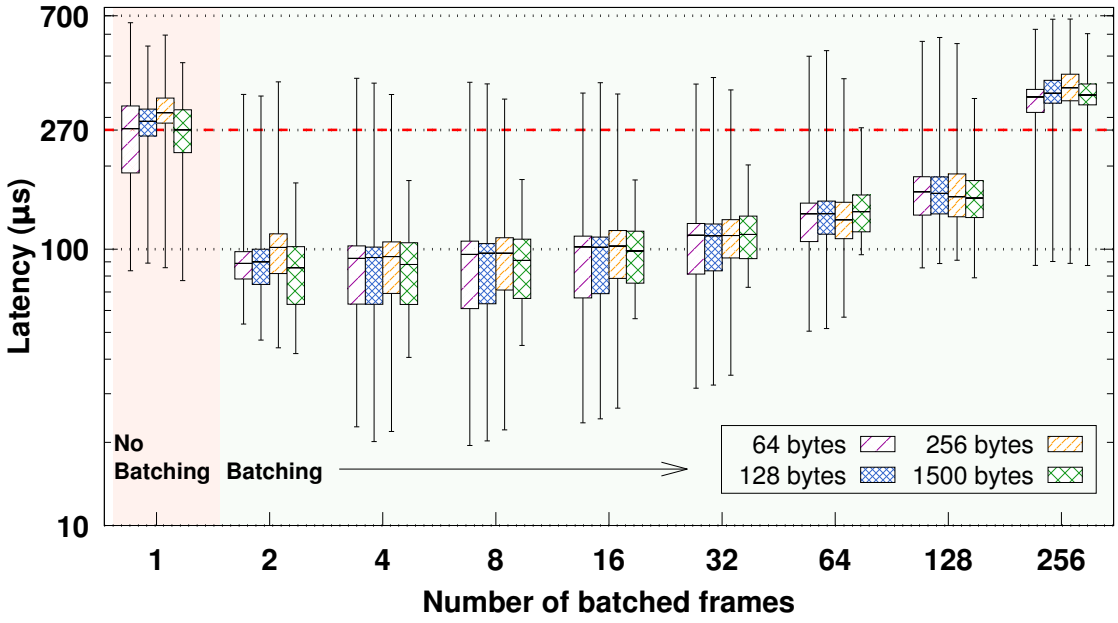


Figure 6.10: End-to-end latency (μs), plotted on a logarithmic scale, versus the number of frames multiplexed/batched into one system call for a user-space FastClick router using the native Linux network driver. The router runs in a single core and the input packet rate is 0.82 Mpps with 64, 128, 256, 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 [110].

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 transfer 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 service chain and attributed these overheads to the inability of CFS to grant large enough time quanta per NF. Here, we deploy NFV service 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 service 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 service 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 service 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 left most boxplot (the top line in the legend) is scheduled by the default CFS (with the default time quantum up to 100 ms), while the other two sets of service 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

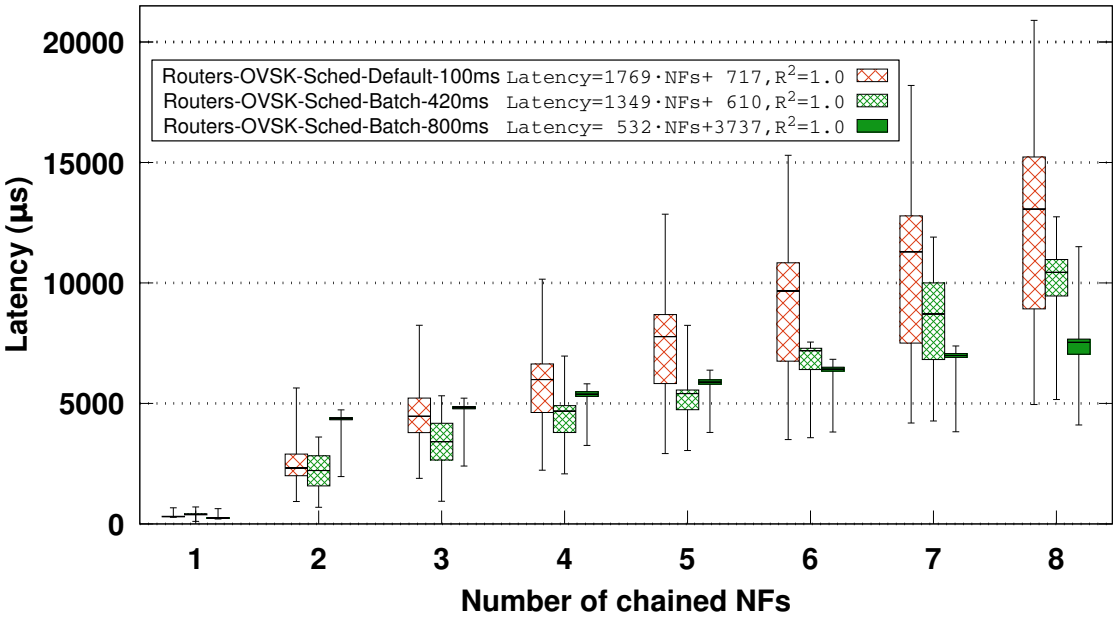


Figure 6.11: End-to-end latency (μs) versus the service chain's length for FastClick routers, running in containers on top of OVSK. The service chains are scheduled either with the default or batch CFS policies, the latter with different time quanta allocations. The routers run in a single core, OVSK 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 service 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 service chain, hence no multiplexing is used in this test.

Based on Figure 6.11, the SCC Scheduler realizes the service chains with a considerably lower latency compared to the default scheduler. Starting from the service 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 service 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 service 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 service chain's length. In contrast, using the maximum time quantum (i.e., 800 ms) appears to be an overkill for short service chains, as the latency is actually higher than the default CFS. However, if an NFV provider wants to deploy service 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 service 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 service chains exhibit a huge latency variance, that is orders of magnitude greater than the batch scheduling cases for long service chains. For example, the variance between the 25$^{\text{th}}$ and the 75$^{\text{th}}$ latency percentiles, for a service chain with 7 routers, when using the default scheduling scheme is 6327 μs. The same percentiles for the same service 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 service 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 service 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 service chains. The top set of service chains in the legend (the left most boxplot) 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 service 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 service chains use the batch CFS without I/O multiplexing, while the last three sets of service chains use I/O multiplexing with batch sizes 2, 16, and 32 respectively.

Looking first at the latencies of the service 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 service 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%

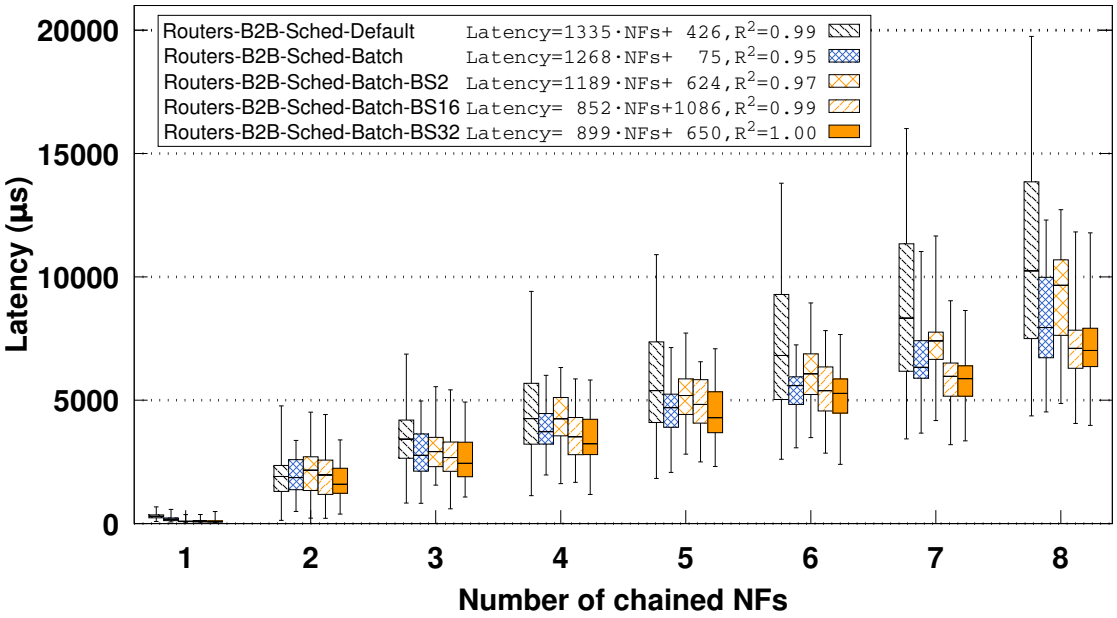


Figure 6.12: End-to-end latency (μs) versus the service chain's length for FastClick routers, running in B2B chained containers. The top set of service chains in the legend are scheduled by the default CFS. The other four service chains are scheduled by the batch CFS; the first of them does not use I/O 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 service chains with 2 NFs.

lower (1268 μs versus 1335 μs), one should pay attention to the intercepts of these functions (75 versus 426 μs for a zero length service chain). These values indicate the basic processing cost of each service 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 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 service 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 service 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 service chain's run-time. For a single router (i.e., service 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 service chain of 4-8 NFs the time that the service chain is runnable but does not execute due to contention in the scheduler is 3-4x greater than the actual run-time of the service chain for both OVSK and B2B cases. The batch CFS scheduler employed by SCC reduces this overhead by ~50%. However, for a service 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 service 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 service chain of 2 routers, since our I/O multiplexing better exploits the available CPU time.

As for the metric "Wait/RunTime", also defined in §6.3.2.1, this is mostly affected by the I/O mechanism of the service chain. We measured a ~40-60% reduction of this overhead when we used SCC's I/O multiplexing, compared to a service 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 service 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.

Service 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 §3.2), scheduling (see §3.5), and system profiling (see §3.6).

6.6.1 System Benchmarks

lmbench [156] and Intel's memory latency checker [157] measure 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, either of these tools 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 tool. In addition to memory latencies, SCC requires kernel-level benchmarks to measure scheduling and system call overhead. Although there are relevant available tools such as that described in [190], own benchmarks is needed to acquire these metrics (as we have done in [186]).

6.6.2 Code Profilers

OProfile [158] and Perf [159] 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 were 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 [160], callgrind's KCachegrind tool [161] (based on valgrind), and Intel's PMU [162]. 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 [163] 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 [164] 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. [151] 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. [152] 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 [33], DPDK [28], PFQ [34], and PF_RING [191] 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 [94] 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 [185] is built on top of FastClick. Earlier efforts have successfully applied similar techniques [33, 110, 28] 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, while maintaining the ability to run on a standard Linux kernel.

Chapter 7

Synthesizing High Performance NFV Service Chains

Several packet processing consolidation attempts were made to improve the performance of chained NFs, as discussed in §3.3. Contemporaneously with one of the most effective attempts called OpenBox [31], we implemented the mechanisms specified in [192] 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 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, potentially 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 service 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: (*i*) consolidates all the *read* operations of a traffic class into one element, (*ii*) early-discards those traffic classes that lead to packet drops, and (*iii*) associates each traffic class with a *write-once* element. Moreover, SNF shares elements among NFs to avoid unnecessary overhead and compresses

*The work described in this chapter is based on the journal article "SNF: Synthesizing high performance NFV service chains" [29] (the authors of the article retained the copyright and give their joint approval for parts of this material to appear in this thesis).

the number and length of the service 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 service 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 [13] 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 [42] 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 [193].

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 service chains. SNF service 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 an OpenFlow switch allows SNF to realize ISP-level service chains at 40 Gbps (for most frame sizes), while bounding the median 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 service 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 RSS [102] scheme by [194] 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.

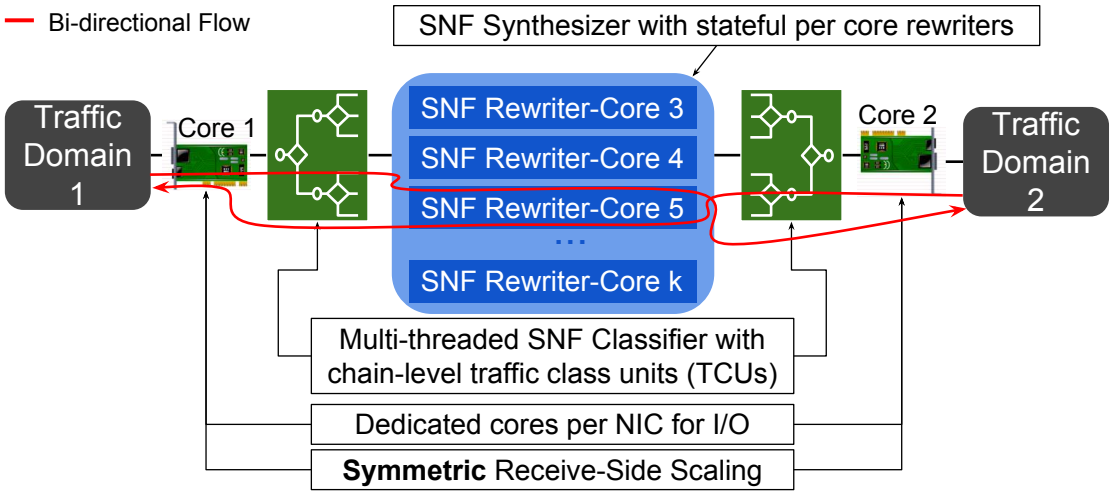


Figure 7.1: SNF running on a machine with k (k>5 in this example) CPU cores and 2 NICs. Dedicated CPU cores per NIC deliver bi-directional flows to packet processing CPU cores via symmetric RSS. Processing cores concurrently classify traffic and access individual, stateful SNF rewriters to modify the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

7.1.1 Main Objectives

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

Multiple network I/O interactions between the service 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 service 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 service 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.

These issues are addressed in order to face the third challenge of this thesis, introduced in §4.3.3. 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 [56], 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.

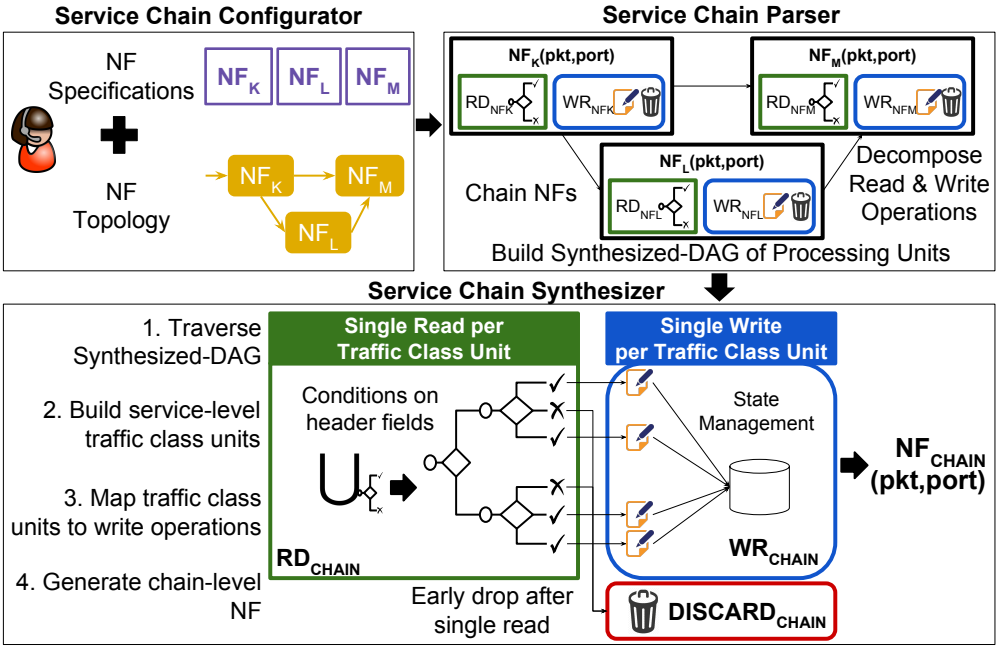


Figure 7.2: The SNF framework. The network operator inputs a service chain and its topology (top left part). SNF parses the chained NFs, decomposes their read and write parts, and composes a Synthesized-DAG (top right part). While traversing the Synthesized-DAG, SNF builds the traffic class units of the service chain, associates them with write/discard operations, leading to a synthesized service 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 service chain is described as a DAG of interconnected NFs (i.e., service 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 [31, 140, 42]. 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 service 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 service chain-level DAG, hence two NFs are interconnected as follows:

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

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

Entry points: In addition to the internal connections within a service chain (i.e., connections between NFs), the Service Chain Configurator also requires the entry points of the service chain. These points are the interfaces of the service 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 service chain-level DAG that describes the service 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 service 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($t = (\phi, \omega)$,$U = \{(\phi_i, \omega_i)_{i \leq m}\}$)
2: **for** $i \in (1, m)$ **do**0
3: $\phi' \leftarrow$ INTERSECT(t, ϕ_i)
4: $\omega' \leftarrow \omega_i \circ \omega$
5: $t' = (\phi', \omega')$
6: TRAVERSE(t',$U.successors[i]$)

Algorithm 2 Intersecting a TCU with a filter

```
1:  function INTERSECT(t = (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:     return φ'
```

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

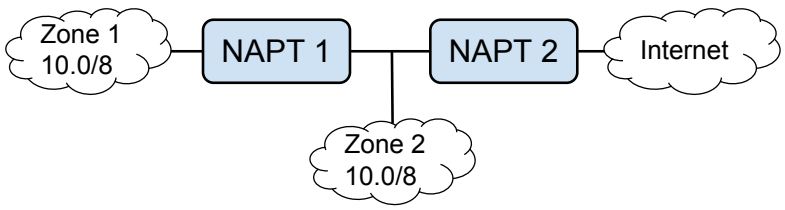


Figure 7.3: Example NAPT chains, where two zones share the same IPv4 prefix.

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 [195].

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

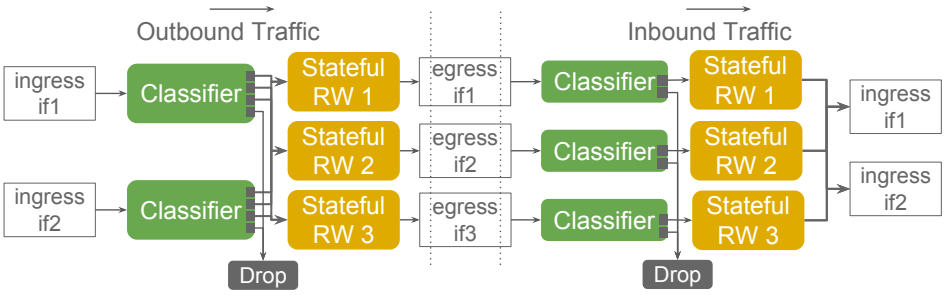


Figure 7.4: State management in SNF.

7.3 A Motivating Use Case

To understand how SNF works and what benefits it offers, 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 service chain consists of a NAPT, a layer 4 firewall, 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 service chain, while UDP is filtered at the firewall (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

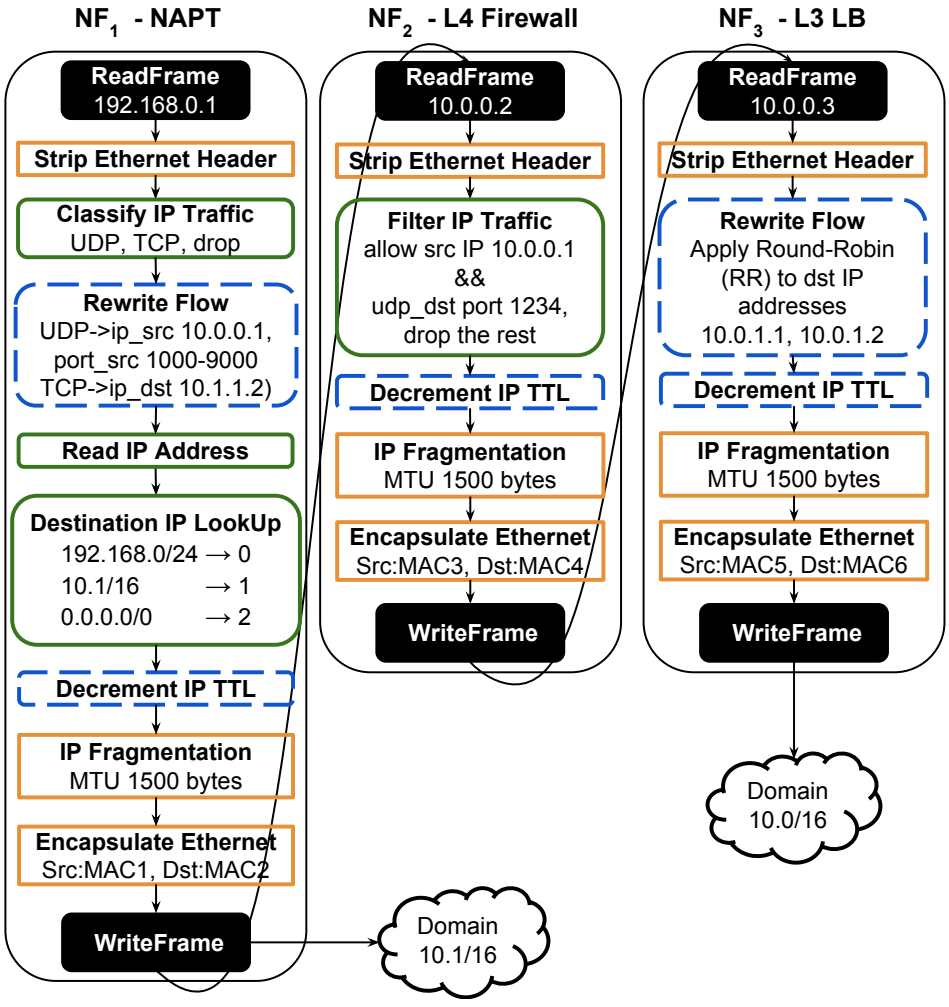


Figure 7.5: The internal components of an example NAPT→L4 Firewall→L3 LB service chain.

discussed. 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 service chain performs a single modification of all these fields. The initial service 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 service 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 service 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 service 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

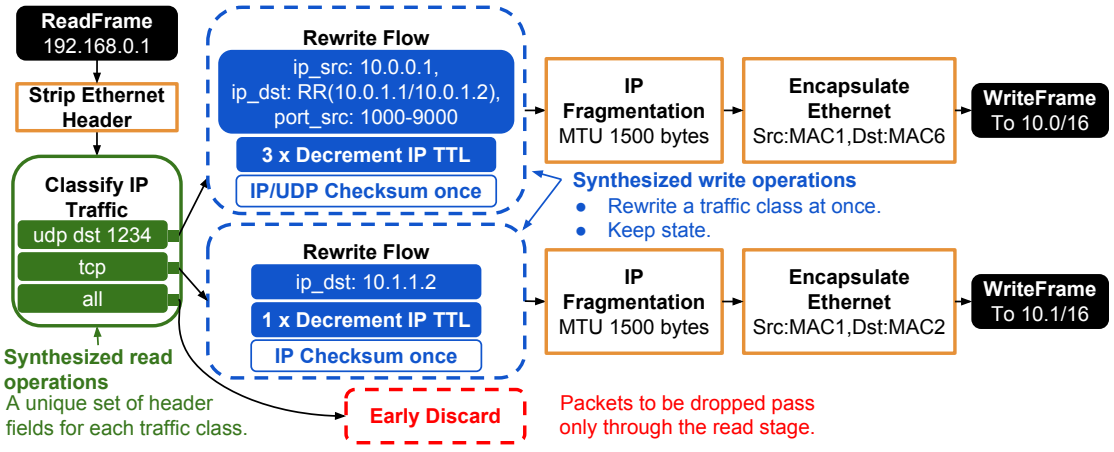


Figure 7.6: The synthesized service chain equivalent to Figure 7.5. The SNF contributions are shown in floating text.

packets is the union of the protocol and destination port numbers. Such a filter is part of a classifier whose output port is linked with a set of write operations (dashed vertices in Figure 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 service chain.

Synthesizing the counterpart of this example implies code modifications to avoid the redundancy caused by 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 routing and filtering decisions in one classifier caused this 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 service chain requires only 5 processing operations to transfer the UDP datagrams along the service chain. The initial service 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 service chain is 2014 CPU cycles/packet, while the synthesized service chain requires 3x less (roughly 695) CPU cycles/packet. If we account for the extra I/O cost per hop for the initial service 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 [122], PacketShader [123], Double-Click [110], SNAP [196], ClickOS [111], and FastClick [125].

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 [125]. *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 [194].

Extension 4: To make software-based classification more scalable, we implemented the lazy subtraction algorithm introduced by HSA [56]. 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 [197].

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 [198, 199].

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:

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

Service Chain 2: Nest multiple NAPT middleboxes.

Service 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 service chains and answer the following questions: Can (stateful) service chains with an increasing service 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,

*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.

two 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 service 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 a single 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 [122, 123, 110, 196, 111, 125], 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 service chain has a 40 Gbps link capacity). The down-pointing (green) triangular points in Figure 7.7 show the throughput achieved by these service chains as a function of the increasing length of the service 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 [125].

In this experiment, we observe that FastClick operates at the maximum throughput only for a service 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 service chain (shown with red squares) to achieve the maximum possible throughput of this hardware, despite the increasing length of the service chain. The fitted equation confirms that SNF operates at the speed of the NICs.

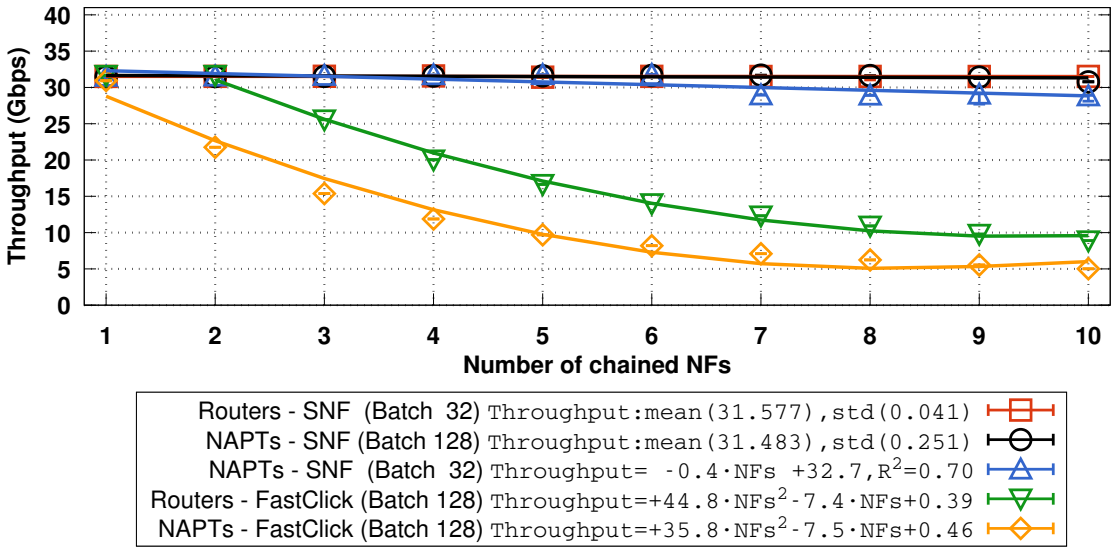


Figure 7.7: Throughput (Gbps) of chained routers and NAPTs using (*i*) FastClick and (*ii*) SNF versus the numbers of chained NFs (60-byte frames are injected at 40 Gbps). Bigger batch sizes achieve higher throughput.

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 [200] or a Carrier Grade Network Address Translator (NAT) [201]. 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 [202].

The goal of this use case is to test SNF in such a stateful context using a service 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 service 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 service chain at line-rate, despite using 8 CPU cores and 128-packet batches.

SNF requires a certain batch size to realize the NAPT service 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 service chain. State lookup and management operations executed for every packet cause this degradation. Depending on the performance targets, a network operator might tolerate a potentially increased latency to achieve the higher throughput offered by an increased batch size.

Next, the effect of different frame sizes on the service chains of routers and NAPTs is explored. We run the longest service 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.

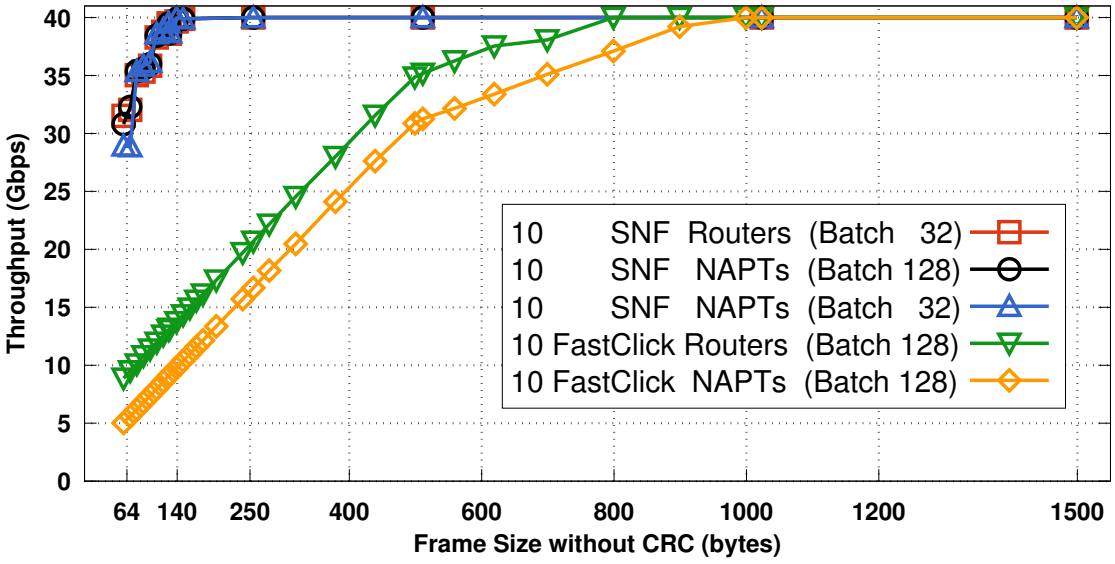


Figure 7.8: Throughput of 10 routers and NAPTs chained using (*i*) FastClick and (*ii*) SNF versus the frame size in bytes (without CRC). The different frames are injected at 40 Gbps.

7.5.4 Performance Analysis of Real Service Chains

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

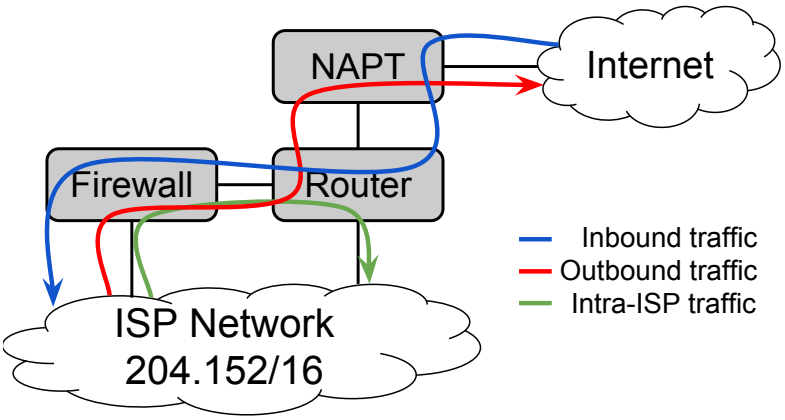


Figure 7.9: An ISP's service chain that serves inbound and outbound Internet traffic as well as intra-ISP traffic using three NFs.

A set of three ACLs [193], taken from ISPs, are utilized to deploy the service chain of Figure 7.9. The firewall 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 v3 are explored using a software-based implementation of the above service chains, while in §7.5.4.2, a hardware-assisted variant of the each of the same service 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® E5-2667 v3 CPU, according to Figure 5.2c. 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 service chains were also implemented in FastClick [125], to have a state of the art performance reference for SNF, according to Figure 5.2b.

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).

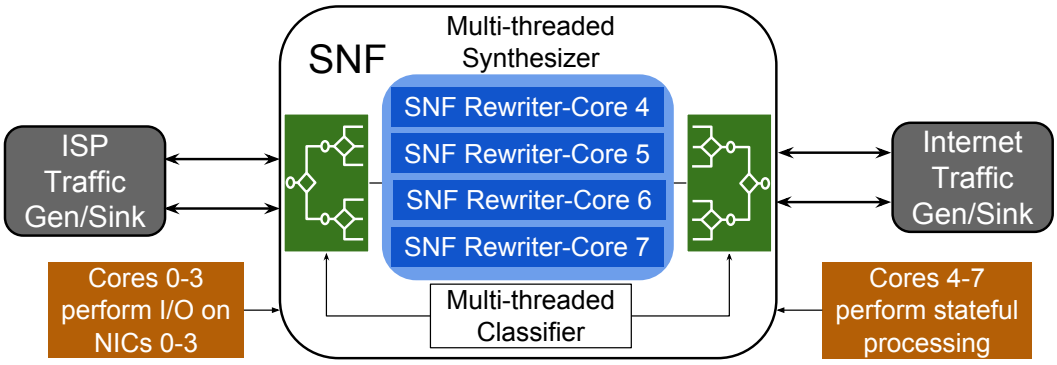
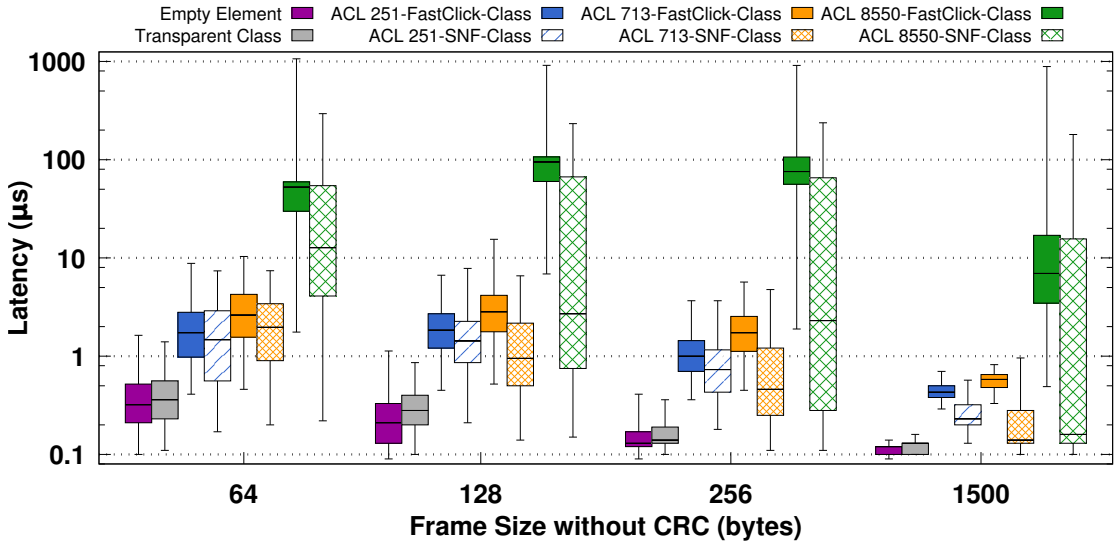


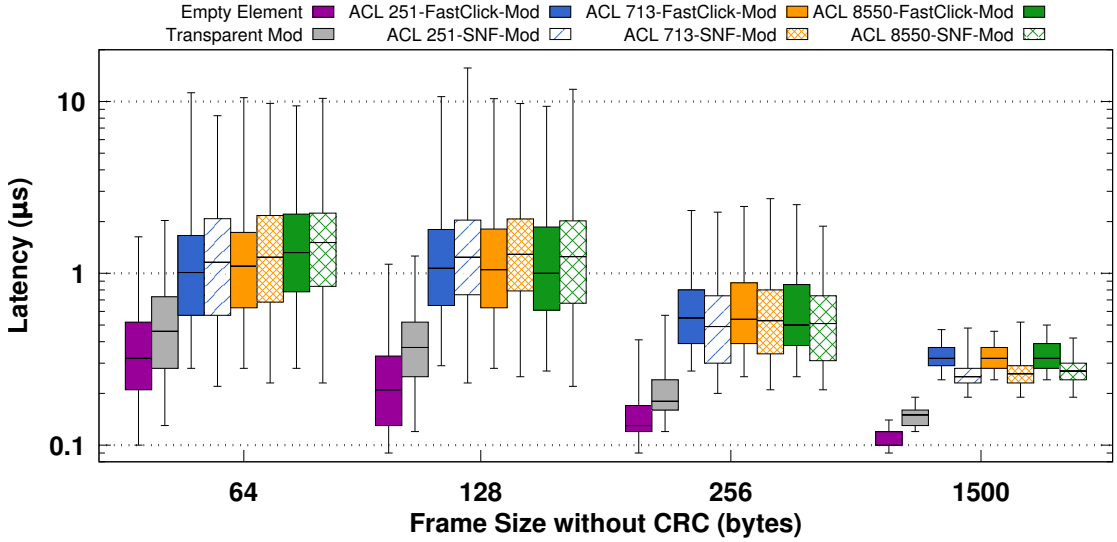
Figure 7.10: Software-based SNF testbed. The ISP and Internet domains are connected to the service chain using 2x10 Gbps NICs each. The service chain has 4x10 Gbps NICs. The machine that executes the service chain uses 4 CPU cores for I/O (one per NIC) and 4 cores (in the same socket) for stateful processing.

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 service chains with 251, 713, and 8550 rules in their ACLs. FastClick and SNF implement these service 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 service 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.

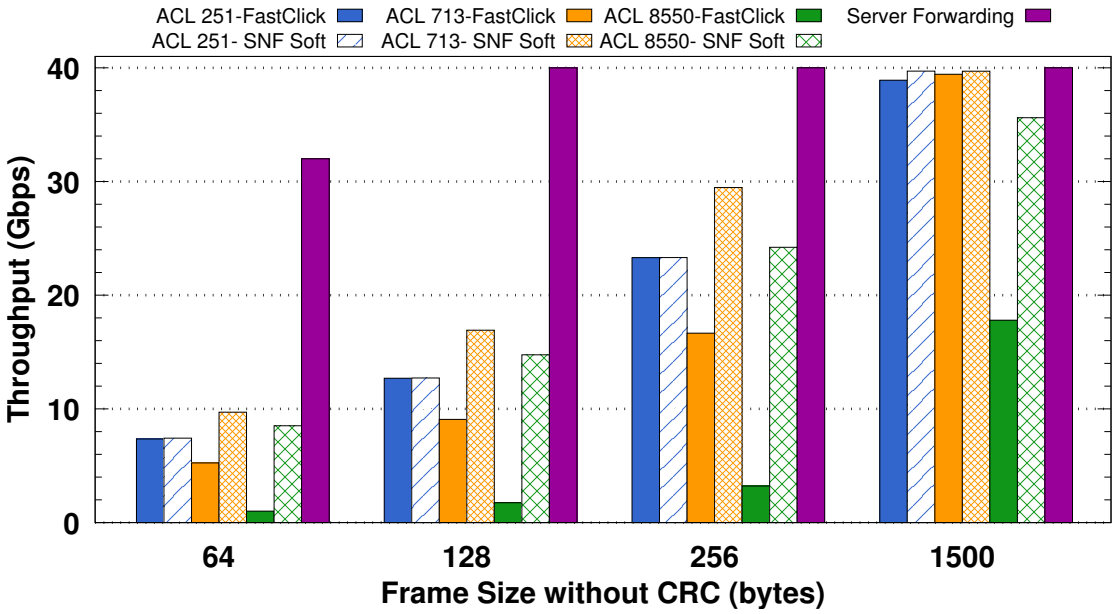
Figure 7.11b shows the latency imposed by SNF and FastClick when applying packet modifications. Generally, the cost of modifying the traffic is lower than the classification cost since the 25^{th} and 75^{th} percentiles span between $100\,ns$ and $2\,\mu s$. This cost is almost 4x greater than the cost of an empty Click element and *independent* of the complexity of the ACLs (which reside in the firewall), since the write operations of these service 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.

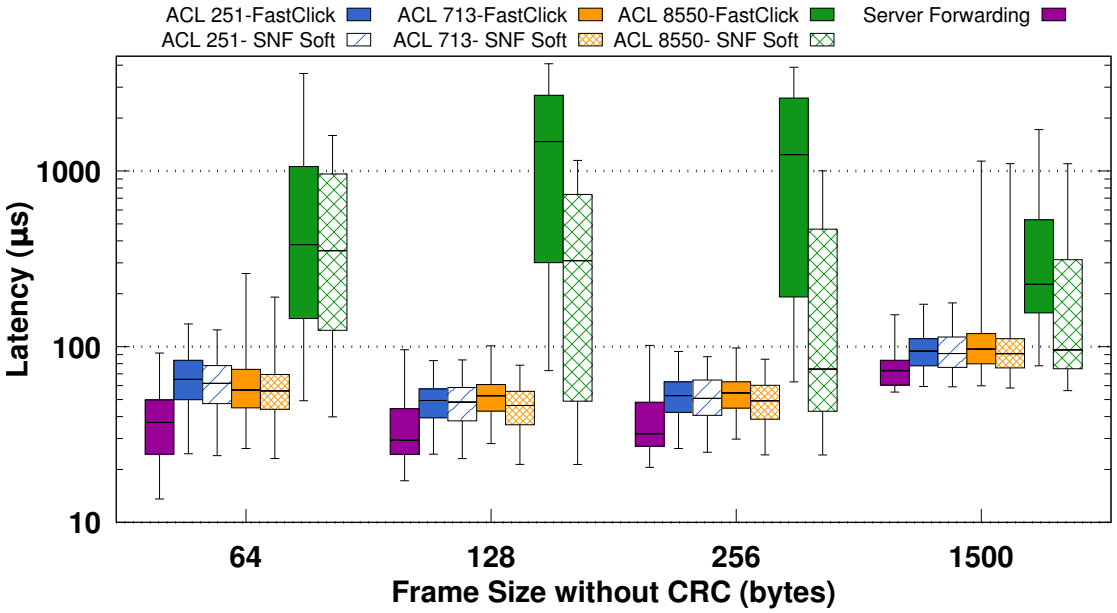
Overall Performance

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 service 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 service 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,



(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 service chains with 251, 713, and 8550 rules in their ACLs. Both frameworks implement these service 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.

we notice 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 firewall, router, and NAPT NFs causes the service 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 service 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 and corresponds to the setup previously shown in Figure 5.2e. 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.

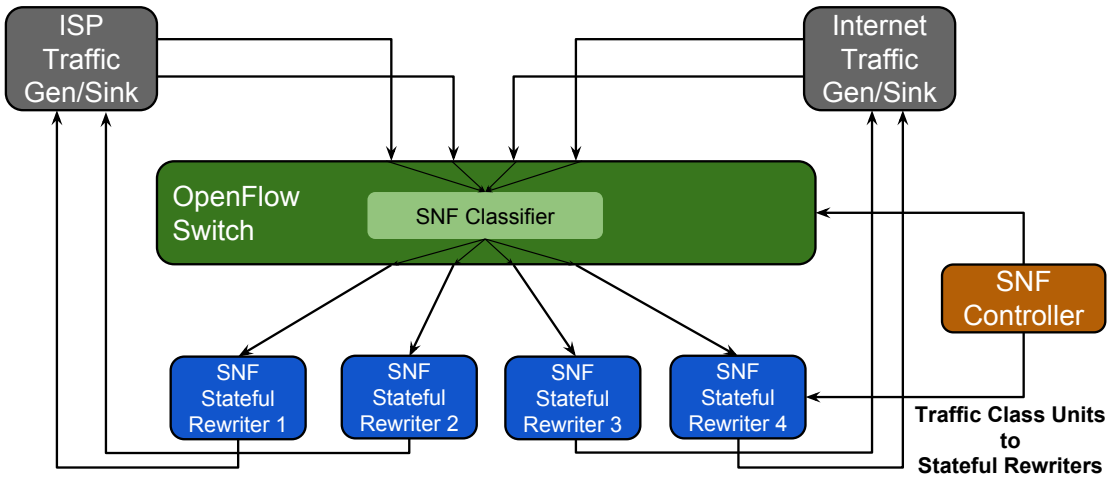


Figure 7.13: Hardware-assisted SNF testbed. Two interfaces per domain (i.e., ISP and Internet) send packets to the hardware-based classifier (ports at the top) of the service chain, realized by an OpenFlow switch. The switch classifies and dispatches input traffic to 4 different output ports connected 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

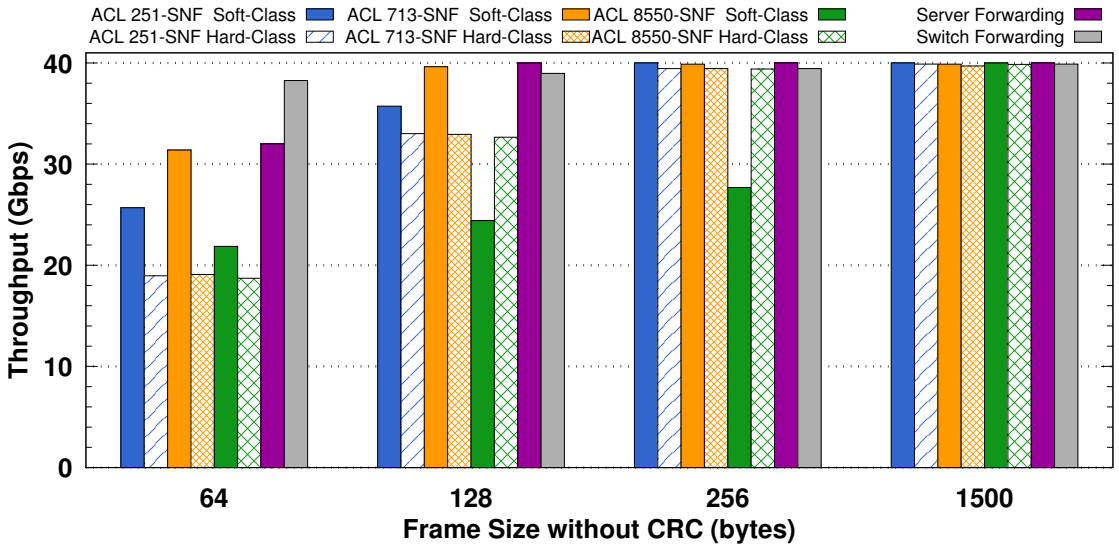
Here the focus is on the performance of the two major parts of the testbed depicted in Figure 7.13, the traffic classification and modification. These performances are measured both on the hardware-assisted and purely software-based SNFs to highlight the benefits of scaling out packet processing.

Classification

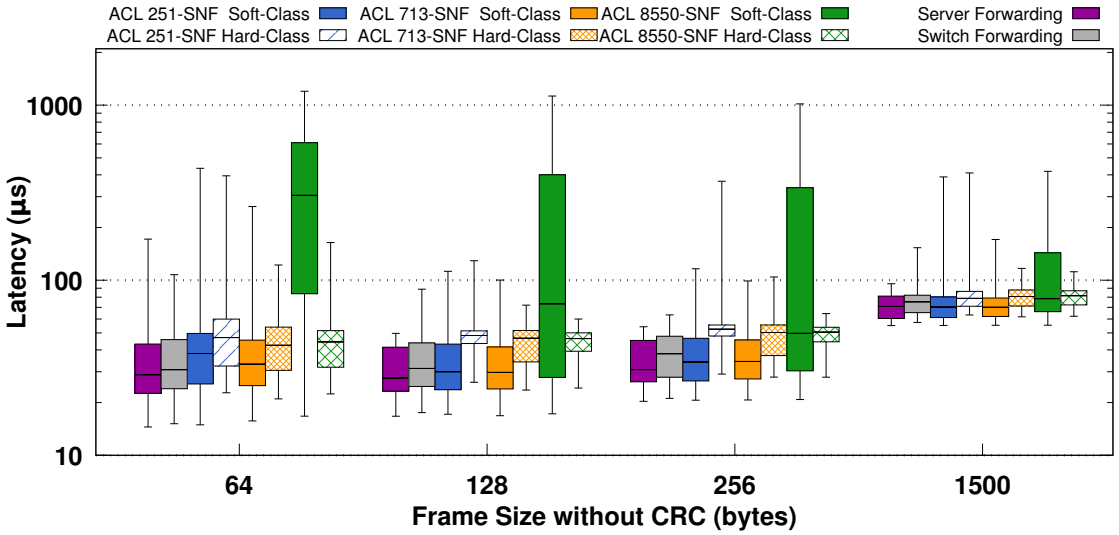
We detached the modification parts of both testbeds to isolate their classifiers. In the testbed of Figure 7.10, we directly encapsulate and output each traffic class derived by the SNF classifiers, while in the testbed of Figure 7.13, the four output ports of the switch are connected back to the origin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

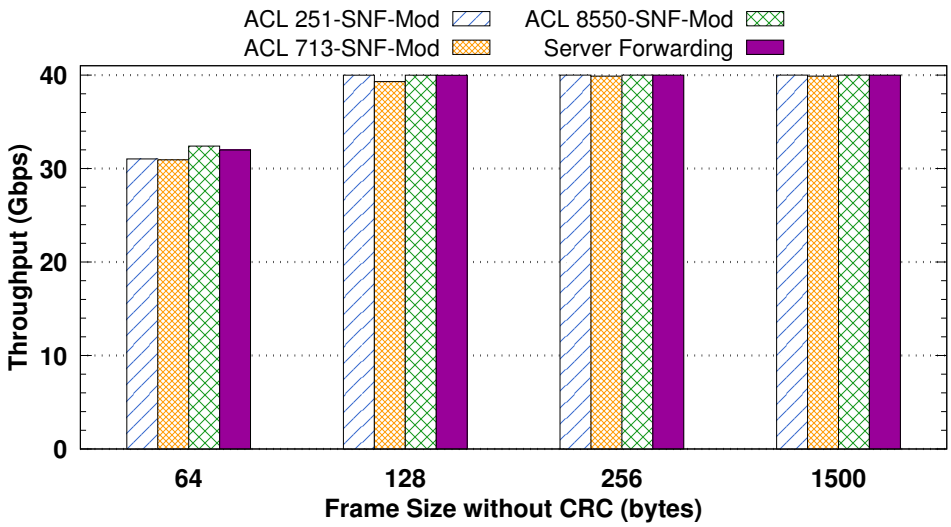
Second, for the ACLs with 251 and 713 rules, the two classifiers achieve comparable throughput (see Figure 7.14a) for all frame sizes and the latency (see Figure 7.14b) of the software-based classifier is lower than the latency of the hardware classifier (excluding some outliers).

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.

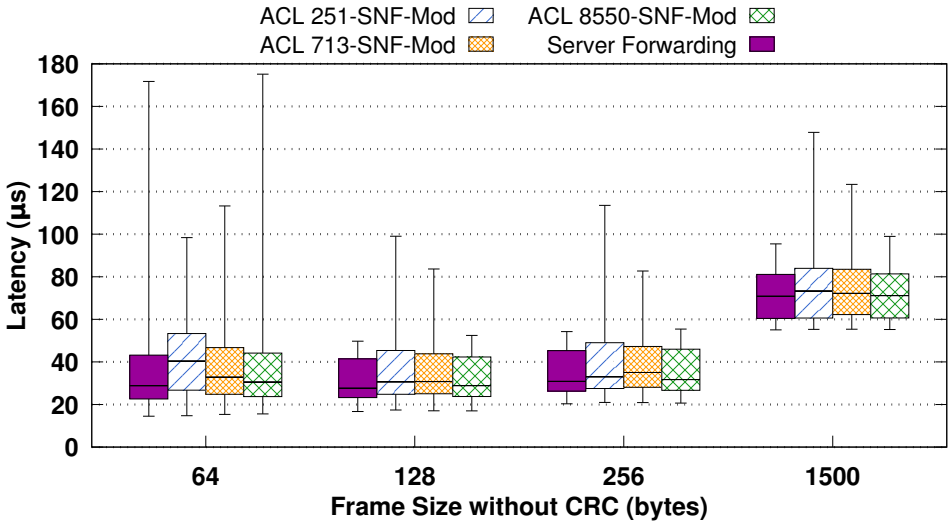
Modification

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 service 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 service chain four different frame sizes are tested, and as a reference point, the traffic modification performance of each service 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 service 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.

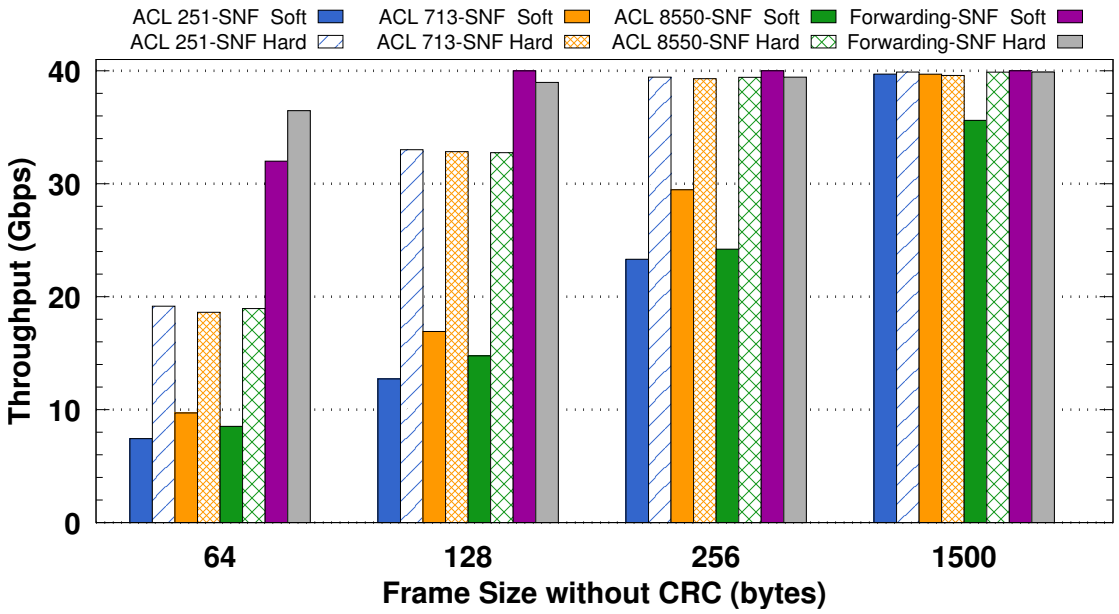
Overall Performance

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 service chains using a fully-fledged software-based and a hardware-assisted SNF.

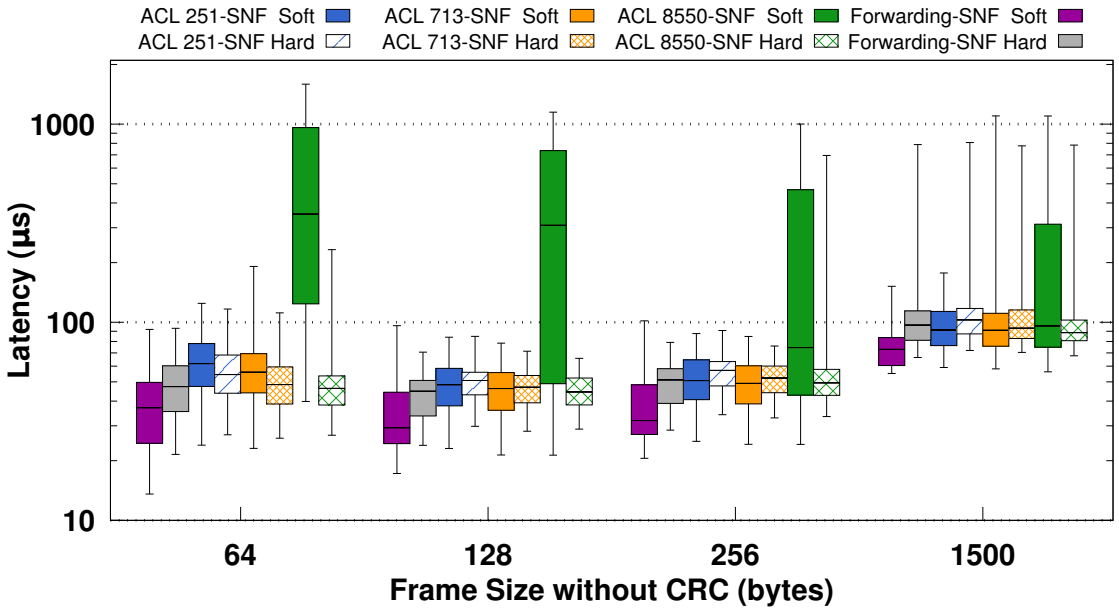
We observe that a fully-functional hardware-assisted SNF is twice as fast as the software-based version for the smallest frames. At first sight, this observation is not inline with the findings of Figures 7.14 and 7.15. Specifically, the isolated software-based SNF classifier (see Figure 7.14a) achieved 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 service 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 service 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 service 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 service 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 service chains with 251, 713, and 8550 rules in their ACLs. The classifier of the hardware-assisted SNF 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 [53, 54] 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 [62, 63] or combined with model checking [53], 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 [63] might be suitable for verifying NFV service chains. In [63], 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 [62] could be employed to test the interoperability between a service 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 [56] 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 [54] 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 service 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 §10.2).

7.7 Originality and Open Source Contributions

In §3.2 we studied modern packet processing architectures, emphasizing on network I/O and processing techniques. In §3.3 middlebox consolidation platforms were discussed. In this section, the originality of SNF is highlighted with respect to these efforts.

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 [197].

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 [111] together with fast network I/O [33, 28] and processing [110, 124, 125] 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 [137] consolidates middlebox-oriented flow processing into one machine, mainly at the session layer. Similarly, OpenNF [134] provides a programming interface to migrate NFs, which can in turn be collocated in a physical server. DPIaaS [138] reuses the costly DPI logic across multiple instances. RouteBricks [122] exploits parallelism to scale software routers across multiple servers and cores within a single server, while PacketShader [123] and NBA [124] 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) [139] 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 [140] 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 service chain. Our synthesized SNF realizes such consolidated service chains with zero context switching and zero redundancy of individual packet operations.

Very recently, Bremler-Barr, Harchol, and Hay [31] 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 service chain.

However, OpenBox does not highly optimize the result service 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 service 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 service 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 [56]. 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' caches.

7.7.5 Scheduling NFs for High Throughput

The E2 framework [32] 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 service chains that are load balanced across heterogeneous hardware components (i.e.,CPU and GPU) using NBA.

7.7.7 Summary of Open Source Contributions

A list of open source contributions related to SNF is presented below:

1. SNF's extensions to FastClick [125] are available at [197].
2. The entire SNF framework is implemented in C++11 and can be found at [203].

Chapter 8

NFV Service Chains at the True Speed of the Underlying Commodity Hardware

This chapter* fulfills the promise (and justifies the title) of this thesis by introducing the design and implementation of Metron; an approach for realizing NFV service chains at the speed of the hardware. To the best of my knowledge, Metron is the first system that automatically and dynamically leverages the joint features of the network and server hardware to achieve high performance. Metron *eliminates inter-core transfers* (unlike recent work with 4 [32], 2 [117], or 1 [31] inter-core transfers as shown in Figure 4.6), making it possible to process packets potentially *at L1 cache speeds*. Also, Metron overcomes the load balancing issues of "run-to-completion" approaches [122, 125, 31, 29], by combining smart identification, tagging, and dispatching techniques. A number of challenging problems had to be addressed to realize this vision. First, making efficient use of all the available hardware is hard because of the in-machine request dispatching overheads (described in §4.2). Second, discovering and dealing with the heterogeneous network (switches, NICs) and server hardware, in a generic way, is non-trivial from a management perspective. Third, detecting and dealing with load imbalances that reduce the performance of the initially placed service chains requires rapid and stable adaptation. The research contributions are stated below, while dealing with the aforementioned challenges:

1. We orchestrate programmable network's hardware to perform stateless processing and packet classification. We deal with hardware heterogeneity by building upon the unified management abstractions of an industrial-grade

*The work described in this chapter is based on the conference paper "Metron: NFV Service Chains at the True Speed of the Underlying Hardware" [30] (the authors of the paper retained the copyright and give their joint approval for parts of this material to appear in this thesis).

SDN controller (Open Network Operating System (ONOS) [48]). This allows Metron to leverage popular management protocols, such as OpenFlow [13] and P4 [86], and easily integrate future ones. We contributed a new driver for programmable NICs and servers to ONOS [204].

2. We overcome the network/server architecture mismatch by instructing Metron to tag packets as early as possible, enabling them to be quickly and efficiently switched and dispatched throughout the entire service chain. To do so, Metron first uses SNF [29] to identify the traffic classes of a service chain and produce a synthesized NF that performs the equivalent work of the entire service chain (see §8.1.3.1). Then, Metron divides the synthesized NF into stateless and stateful operations (see §8.1.3.3) and instructs all available programmable hardware (i.e., switches and NICs) to implement the stateless operations, while dispatching incoming packets to those CPU cores that execute their stateful operations. Metron runs stateful NFs on general purpose servers, while fully leveraging their generic processing power.
3. We propose a way to efficiently and quickly obtain the network state in order to make fast placement decisions at low cost with high accuracy (see §8.1.3.3). We devised a mechanism to coordinate load balancing among servers and their CPU cores, demonstrating that Metron provides comparable elasticity with purely software-based approaches, but at the true speed of the hardware (see §8.1.3.4).

An evaluation shows that Metron realizes deep packet inspection at 40 Gbps (§8.2.3.1) and stateful service chains at the speed of a 100 GbE NIC on a single server (§8.2.3.2). This results in up to 4.7x lower latency, up to 7.8x higher throughput, and 2.75-6.5x better efficiency than the state of the art. It is difficult to improve on this performance unless stateful service chains are completely offloaded to hardware, which is impossible with today's commodity hardware.

8.1 Metron Architecture

In order to address the fourth challenge of this thesis, introduced in §4.3.4, this section describes Metron's system design, starting with a high-level overview via an illustrative example in §8.1.1. §8.1.2 describes the Metron data plane, which is configured by the Metron controller as explained in §8.1.3.

8.1.1 Overview

To understand how Metron works, consider a simple network consisting of two OpenFlow switches connected to a server as shown at the bottom of Figure 8.1.

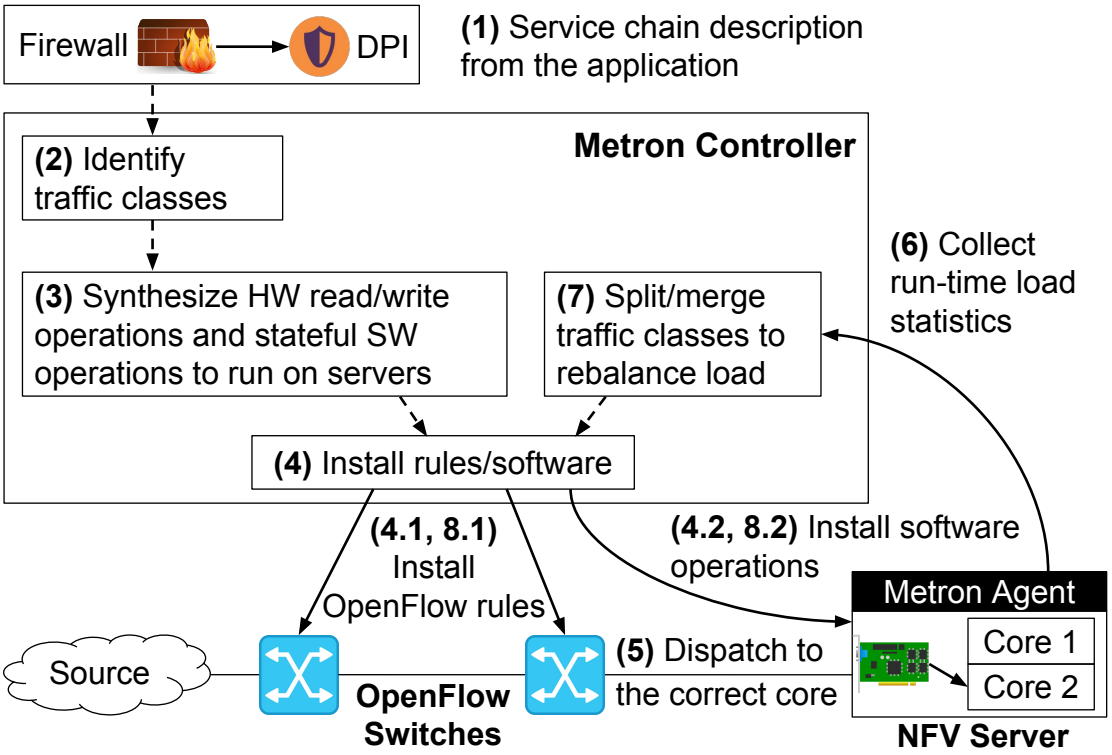


Figure 8.1: Metron overview using an example Firewall→DPI service chain.

Assume that an operator wants to deploy a Firewall→DPI service chain, as shown in Step 1 of Figure 8.1.

In Step 2, the Metron controller identifies the traffic classes* of the service chain, by parsing the packet processing graphs (each graph has a set of packet processing elements as in [42, 32, 31]) of the input NFs. In Step 3, Metron composes a single service chain-level graph by synthesizing the read and write operations of the individual graphs (see §8.1.3.1). Because Metron detects the availability of resources (i.e., the OpenFlow switches) along the path to the server, it associates stateless read and write operations with these components and automatically translates these operations into OpenFlow rules (Step 4.1). The remaining, potentially stateful, operations are translated into software instructions targeting the Metron agent at the server (Step 4.2). The key to Metron's high performance is exploiting hardware-based dispatching (Step 5) that annotates the traffic classes matched by the OpenFlow rules with tags that are subsequently matched by the server's NIC to identify the CPU core to execute the stateful operations. This is how Metron guarantees that each traffic class will be processed by a specific core, thus *eliminating* costly inter-core transfers. This guarantee

*Traffic class is a (set of) flow(s) treated identically by an NF chain.

is maintained even when a CPU core becomes overloaded (see §8.1.3.4) as the Metron agent reports run-time statistics (Step 6) that allow the Metron controller to rebalance the load (Step 7), by splitting traffic classes into multiple groups that are dispatched to different cores using different tags (Steps 8.1 and 8.2). This overview is concluded with a survey of popular NFs; noting that in Table 8.1 a substantial portion of these NFs can be fully or partially offloaded to commodity hardware.

Table 8.1: Survey of popular NFs. The offloadability of "Hybrid" NFs depends on the use case.

Network Function	Offloadable to Hardware
L2/L3 Switch, Router	Yes
Firewall/Access Control List	Hybrid
Carrier Grade NA(P)T, IPv4 to IPv6	No
Broadband Remote Access Server	Partially [205]
Evolved Packet Core	Partially
Intrusion Detection/Prevention	Partially [130]
Load Balancer	Hybrid
Flow Monitor	Yes
DDoS Detection/Prevention	Yes [206]
Congestion Control (RED, ECN)	Yes
Deep Packet Inspection	No
IP Security, Virtual Private Network	Yes [127]

8.1.2 Metron Data Plane

The Metron data plane follows the master/slave approach depicted in Figure 8.2. The master process is an agent that interacts with (*i*) the underlying hardware by establishing bindings with key components, such as NICs, memory, and CPU cores and (*ii*) the Metron controller through a dedicated channel.

The key differentiator between Metron and earlier NFV works is the tagging module shown in Figure 8.2. This module exposes a map with tag types and values that each NIC uses to interact with each CPU core of a server; this map is advertised to the Metron controller. The controller *dynamically* associates traffic classes with specific tags in order to enforce a specific flow affinity, thus controlling the distribution of the load. Most importantly, this traffic steering mechanism is applied by the hardware (i.e.,NICs), hence Metron does not require additional CPU cores (as E2 does) to perform this task, thus packets are directly dispatched to the CPU core that executes their specific packet processing graph.

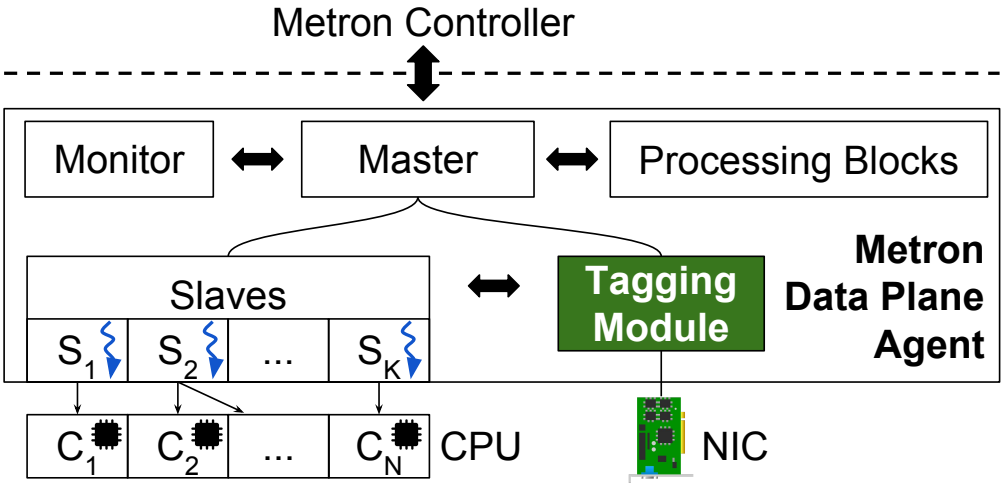


Figure 8.2: The Metron data plane.

When the master boots, it configures the hardware and registers with the controller by advertising the server's available resources and tags. Then, the master waits for controller instructions. For example, the master executes a deployment instruction by spawning a slave process that is pinned to the requested core(s) and by passing the processing graph to the slave. In the context of service chaining, a Metron slave needs to execute multiple processing graphs, each corresponding to a different NF in the service chain. Such graphs can be implemented either in hardware or software. Earlier works implement these graphs in software and use metadata to share information among NFs and to define the next hop in a service chain. Although Metron supports this type of software-based chaining, as shown in §4.2, this approach introduces unnecessary overhead due to excessive inter-core communication and potentially under-utilizes the available hardware. Next, §8.1.3 explains how we approach and solve this problem.

8.1.3 Metron Control Plane

Here, we describe the key design choices and properties of the Metron controller.

8.1.3.1 Synthesis of Packet Processing Graphs

Given a set of input packet processing graphs, one per NF, Metron combines them into a single service chain graph. To ensure low latency, the Metron controller adopts SNF [29]; a more aggressive variant of OpenBox for merging packet processing graphs, which provides a heuristic for solving the graph embedding problem (see [207, 208, 209]) in the context of NFV. Metron uses SNF to eliminate

processing redundancy by synthesizing those read and write operations that appear in a service chain as an optimized equivalent packet processing graph. SNF guarantees that each header field is read/written only once, as a packet traverses the graph.

Another benefit of SNF's integration into Metron is the ability to encode all the individual traffic classes of a service chain using a map of disjoint packet filters (Φ) to a set of operations (Ω). In §8.1.3.4 we use this feature to automatically scale packet processing in and out, providing greater elasticity than available today.

8.1.3.2 Initial Resource Allocation

To allocate resources for the synthesized graph, we allow application developers to input the CPU and network load requirements of their service chains. Alternatively, this information can be obtained by running a systematic NFV profiler, such as SCC [27], or by using more generic profilers, such as DProf [164]. Even in the absence of accurate resource requirements, Metron dynamically adapts to the input load as discussed in §8.1.3.4.

8.1.3.3 Placement

Metron needs to decide where to place the synthesized packet processing graph. Such a decision is not simple, because Metron considers both the servers and the network elements along the path to these servers.

Table 8.1 showed that a large fraction of NFs cannot be implemented in commodity hardware today, mainly because they require maintaining state. This means, that the synthesized graph of such NFs cannot be completely offloaded. To solve this, we designed a graph separation module to traverse and split the synthesized graph into two subgraphs. The first subgraph contains the packet filters and operations that can be completely offloaded to the network (we call this a stateless subgraph), while the second (stateful) subgraph will be deployed on a server. The average complexity of this task is $O(\log m)$, where m is the number of vertices of the synthesized graph.

Given these two subgraphs, Metron needs to find a pair of nodes (a server and a network element) that satisfy two requirements: (i) the server has enough processing capacity to accommodate the stateful subgraph and (ii) the network element has enough capacity to store the hardware instructions (e.g., rules) that encode the stateless subgraph. Metron's placement scheme deals with logical servers and network elements, hence allowing further partitioning of the graphs when no single server or network element has sufficient resources. Future work will allow Metron to prevent service chain placement when there is not enough network throughput available for the traffic between a network element and a Metron server.

Scalable Placement with Minimal Overhead

In large networks with a large number of servers and switches, it is both expensive and risky to obtain load information from all the nodes. This is expensive because a large number of requests need to be sent frequently and this would occupy bandwidth to each node, generate costly interrupts to fetch the data, and occupy additional bandwidth to return responses to the controller. This is risky because the round-trip time required to obtain the monitoring data is likely to render this data stale, leading to herd behaviors and suboptimal decisions. To make a placement decision with minimal overhead, we use the simple, yet powerful, opportunistic scheme of "the power of two random choices" [210]. According to Mitzenmacher, this number offers exponentially better load balancing than a single random choice, while the additional gain of three random choices only corresponds to a constant factor.

Algorithm 3 outlines our server placement scheme. Metron queries the load of two randomly selected servers (line 5) and selects the least loaded of these two servers (lines 7 and 8), provided that the necessary resource requirements (i.e., number of NICs and CPU cores) can be met. If the first two choices fail, then these two servers are removed from the list (line 12) and the process is repeated until a server is found (line 11). Note that this algorithm indirectly prioritizes service chain deployments that exhibit spatial correlation with respect to the processing location because spreading this processing might result in lower performance, which is undesirable. Network operators can input a desired server per service chain to the Metron controller, thus completely bypass the random server selection process described by Algorithm 3.

Algorithm 3 Server selection for placing the stateful part of a service chain.

```
1:  function SELECTSERVER(T,NbN,NbC)
2:      serversList ← T.metronServers()
3:      t ← ∅                                   ▷ t will store the chosen server
4:      while size(serversList) > 0 do
5:          choices ← twoRandomChoices(serversList)
6:          for s ∈ choices do
7:              if (s.nics ≥ NbN) && (s.freeCores ≥ NbC) then
8:                  if (t == ∅) || (t.load < s.load) then
9:                      t ← s                          ▷ Better choice
10:         if t ≠ ∅ then
11:             return t                               ▷ Server found
12:         serversList.remove(choices)
13:     return ∅
```

Having randomly or explicitly selected a server also greatly simplifies the second placement decision (i.e., the network element(s) to offload processing to). Well-designed networks, such as datacenters, provision several fixed shortest paths between ingress nodes (e.g., core switches) and servers, where each server might be associated with a single core switch [211, 212]. Given this, we use Algorithm 4 to find the most suitable network element to offload the stateless graph, using the following inputs:

1. the topology graph (T);
2. the server (Srv) where the stateful subgraph will be deployed (chosen by Algorithm 3 or explicitly set by the network operator), and;
3. the rule capacity (D) required to offload the stateless subgraph.

Our algorithm intentionally prioritizes selection of the very first switch (ingress) toward the target server (line 3). There are two reasons for this. First, applying the classification operations of a service chain at the earliest possible stage, greatly simplifies traffic steering for this service chain at all subsequent network elements on the path to the NFV server. This is done by tagging the packets targeting this service chain at the ingress node and using only this tag to match traffic at any successor of the ingress node. Second, popular network protocols, such as multi-protocol label switching [213], consider ingress and egress switches to be more sophisticated, thus more powerful, by design. After a target switch has been selected, the rules that encode the stateless subgraph are installed in this switch and a unique tag is appended to each of the rule actions. To establish the path between the selected switch and server, one rule is installed in each subsequent switch along the path; this rule matches the tag of the offloaded service chain

Algorithm 4 Switch selection for placing the stateless part of a service chain.

```
1:  function selectSwitch(T,Srv,D)
2:      C ← T.availableCapacityMatrix()
3:      inSw ← T.ingressNodeToward(Srv)
4:      return recursiveSwitchSelection(T,inSw,Srv,C,D)
5:  function recursiveSwitchSelection(T,Sw,Srv,C,D)
6:      if D ≤ (C_Sw · CAP_THR) then   ▷ Demand is a fraction of the capacity
7:          return Sw
8:      else
9:          nextSw ← T.nextNodeInPath(Sw, Srv)
10:         if nextSw == ∅ then
11:             return ∅
12:         setupPath(Sw,nextSw)
13:         return recursiveSwitchSelection(T,nextSw,Srv,C,D)
```

and selects the port that leads to the server that executes the stateful part of this service chain.

According to Algorithm 4, if the capacity requirements of the ingress switch cannot meet the requirements for offloading a service chain (line 6), our algorithm selects a subsequent node along the path to the NFV server (line 9), sets up forwarding between the current and next node (line 12), and applies the same logic recursively (line 13). Our decision is currently based on a single criterion (line 6); that is, the rule capacity of a candidate switch must be greater than required and at the same time the resources required must not exceed some measure of the capacity (CAP_THR). The latter condition ensures that this switch has enough space to accommodate future rule updates.

Handling Partial Offloading and Rule Priorities
Metron carefully treats the cases when (*i*) a stateless subgraph contains rules with different priorities and (*ii*) one or more rules of such a subgraph cannot be offloaded to hardware. The latter can occur, e.g., due to the hardware's inability to match specific header fields. In such a case, Metron will selectively offload only the supported rules, while respecting rule priorities. To exemplify these two cases, assume a service chain that needs to be deployed on the topology shown in Figure 8.1. Assume that this service chain implements four rules that can be offloaded to the first programmable switch, while the remaining part of the service chain will be deployed on the server. If rule 3 cannot be offloaded and all of the rules have the same priority, then Metron will offload rules 1, 2, and 4. However, if these rules have, e.g., decreasing priorities (i.e., rule 3 has a higher priority than rule 4), then Metron will offload only the first two rules, to guarantee that the server applies rule 4 after rule 3.

8.1.3.4 Dynamic Scaling

Section 8.1.3.1 explained how Metron encodes a service chain as a set of traffic classes, where each traffic class is a set of packet filters mapped to write operations. This abstraction gives great flexibility when scaling a service chain in/out. As an example, when E2 detects an overloaded NF, it scales this NF by introducing an additional (duplicate) instance of the entire NF and then evenly splits the flows across the two instances. In contrast, Metron splits the traffic classes of this NF across two instances, such that each instance executes the code responsible for each of its traffic classes (rather than the code of the entire NF).

To trigger a scaling decision, Metron gathers port statistics from key locations in the network in order to detect load changes. Such a change results in Metron asking for instantaneous CPU load and network statistics from the affected service chains. Because the logic to react to overloads (load splitting) and underloads

(load contraction) is symmetric, here we only focus on the former case. Given this information, Metron applies the following, globally orchestrated, scaling strategy to react to load imbalances.

Traffic Class-level Scaling

We leverage a grouping technique when creating a service chain's traffic classes. A set of T traffic classes $\{ TC_i^j \mid j \in [1,T] \}$ that belong to service chain i can be grouped together, if and only if their packet filters $\{ \Phi_i^j \mid j \in [1,T] \}$ are mapped to the same write operations: $\forall k,l \in [1,T], \Omega_i^k = \Omega_i^l$

For example, a HTTP and a File Transfer Protocol traffic classes heading to a NAT will both exhibit the same stateful write operations from this NF, thus they can be grouped together. The Metron controller has this information available once the traffic classes of a service chain are created (see §8.1.3.1). To dynamically scale out a group of traffic classes, Metron needs to split this group into two or more subgroups, where the first subgroup remains on the same CPU core as the original group, while the other subgroup(s) are deployed and scheduled on a different (set of) CPU core(s). These new traffic classes are annotated with different tags, such that the NIC at the server can dispatch them to the appropriate CPU cores. We call this mechanism "traffic class deflation" to differentiate it from the opposite "traffic class inflation" process, where two or more groups of traffic classes that exhibit the same write operations are merged together, when Metron detects low CPU utilization.

To simplify load balancing, while keeping a reasonable degree of flexibility, the split and merge processes always use a static factor of 2 (i.e., one group is split into two, or two groups are merged into one). This decision also minimizes the amount of state that Metron needs to transfer across CPUs. A fully dynamic solution with additional visibility into the load of each traffic class would achieve better load distribution; however, such a solution is considered impractical in the case of large networks with potentially millions of traffic classes. Split and merge operations may repeat until Metron can no longer split/merge a traffic class. A single flow is an example of non-splittable traffic class. The reaction time of this strategy is mainly affected by the time required for the controller to monitor and reconfigure the data plane. In §8.2.4 we show how this strategy performs in practice.

Once an inflation/deflation decision has been made, Metron needs to guarantee that the state of the affected traffic classes (e.g., those being redirected to a different CPU core in the case of deflation) will remain consistent. To do so we adopt a scheme that quickly duplicates the stateful tables of a group of traffic classes across the involved CPU cores, when inflation occurs. Similarly, we merge the stateful tables of two groups during the inflation process. Although this scheme introduces some redundancy (entries of migrated traffic classes will still occupy space in the memory of the previous CPU core until they expire), it offers a quick solution to a

problem that is beyond the scope of this work. StateAlyzr [135], OpenNF [134], or the work by Olteanu and Raiciu [214] could be integrated into Metron to provide more efficient state management solutions. Alternatively, state management could be delegated to a remote distributed store as per Kablan, et al. [136].

8.1.3.5 Distributed Control Plane

Metron follows a very different design than existing NFV controllers, such as OpenBox and E2. Metron is a *distributed* NFV framework that enables elasticity at both the control and data planes.

A distributed control plane provides fault-tolerance and resilience in the face of failures that might compromise the NFV services. At the same time, the system as a whole can meet performance targets that are far higher than what a single instance might be able to handle, thus allowing the control plane to scale together with the data plane. Metron's elastic control plane allows us to partition the network into multiple segments with different controller instances managing different devices, while maintaining a globally consistent network state.

Metron can be configured to operate with strong or eventual consistency guarantees, depending on the performance targets of the operator. To provide strong guarantees, Metron's distributed engine is based on the Raft [215] consensus protocol. Eventual guarantees trade some consistency for superior performance by reading and writing local state, while updates are subsequently propagated to other replicas in the background.

When an application registers a service chain with Metron, its packet processing graphs are stored and replicated across all Metron instances, while one primary instance undertakes to place and deploy the service chain in the network segment with the highest availability, as explained in §8.1.3.3.

8.1.3.6 Integrating Blackbox NFs

Some NF providers might not wish to disclose the source code of their NFs. In this case we offer two integration strategies: (*i*) partially synthesize a service chain, while using DPDK ring buffers to interconnect synthesized NFs with blackbox NFs or (*ii*) input only an NF configuration (e.g., DPI rules, omitting DPI logic) using Metron's high-level API, thus let Metron use its own data plane elements to realize this NF (see §8.2.3).

8.1.4 Routing (Updates) and Failures

To explain how Metron's routing and dispatching works and how Metron reacts to routing updates and failures, we use the example shown in Figure 8.3. We assume a software-defined* network on which the network operator has deployed a routing application that routes HTTP traffic† between source and destination (through the path s1→s3). The routing is done using the information shown within green dashed-dotted outlines.

A policy change forces the network operator to further process the HTTP traffic before it reaches its destination. Thus, she deploys an HTTP service chain (described by the top box with dotted outline in Figure 8.3) using Metron. When

*Metron can also operate in legacy networks by adding one or more programmable switches before the NFV servers.

†We assume only HTTP traffic to keep the example simple.

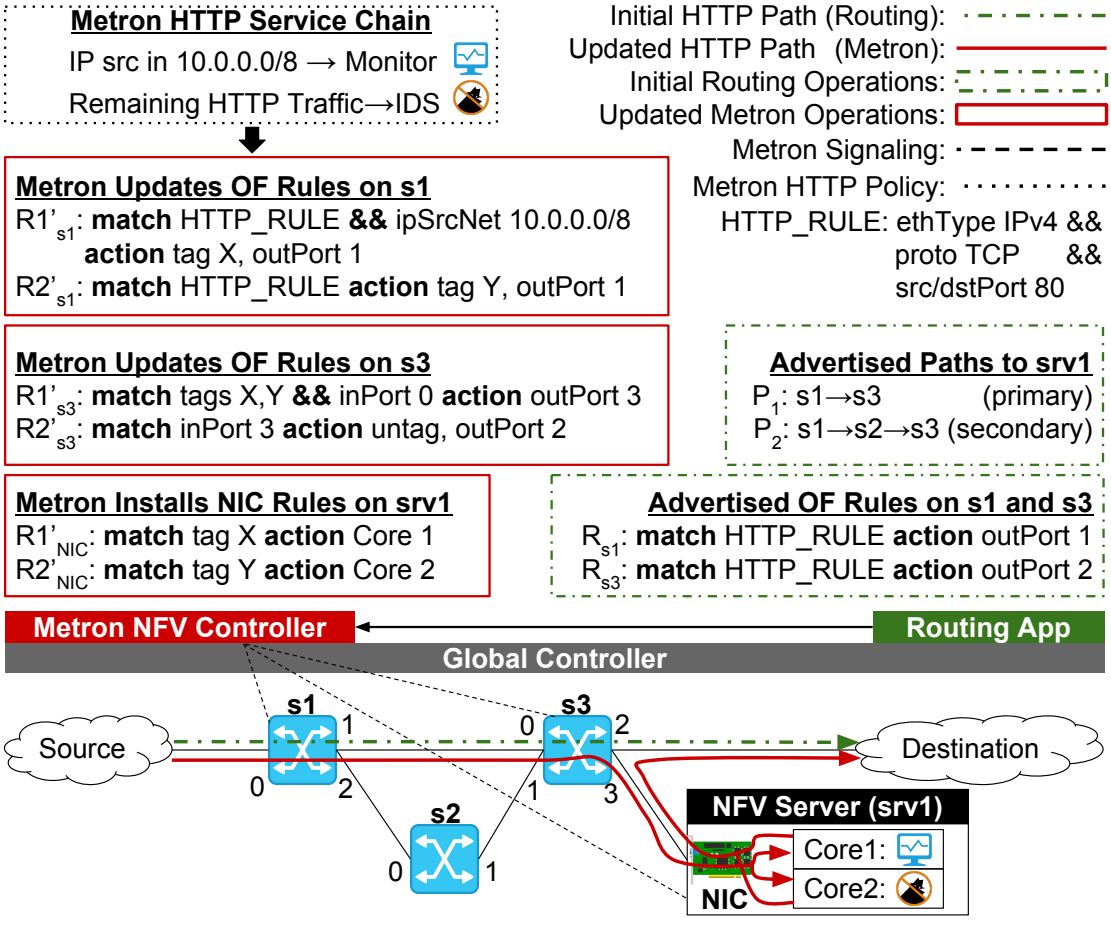


Figure 8.3: Metron's routing & CPU dispatching scheme.

Metron boots it obtains the current routing policy and paths for the HTTP traffic, as advertised by the routing application. Next, the Metron controller performs a set of updates (see the left-side boxes with solid outlines, where OF stands for OpenFlow). The updates focus on two aspects: (*i*) to extend the existing HTTP rules (i.e., R_{s1} and R_{s3} at the bottom right box with dashed-dotted outline) with rules that also perform part of the service chain's operations (i.e., $R1'_{s1}$ and $R2'_{s1}$) and (*ii*) to tag the HTTP traffic classes to allow the NFV server to dispatch them to different CPU cores.

In this example, Metron identifies two traffic classes and tags them with tags X and Y. The tagging is applied by the first switch (i.e., s1 as per Algorithm 4) using the rules $R1'_{s1}$ and $R2'_{s1}$ (top left box with solid outline). The next switch (s3) uses the tags (i.e., rule $R1'_{s3}$) to redirect the HTTP traffic classes to the NFV server, where Metron has installed NIC rules (i.e., $R1'_{NIC}$ and $R2'_{NIC}$) to dispatch packets with tags X and Y to CPU cores 1 and 2 respectively. The first core executes a monitoring NF, while the second core runs an IDS NF. After traversing the service chain, the packets return to s3, where another Metron rule (i.e., $R2'_{s3}$) redirects them to their destination.

If not carefully addressed, a routing change or failure might introduce inconsistencies. Metron avoids these problems by using the paths to the NFV server (i.e., P_1 and P_2), as advertised by the routing application, to precompute: (*i*) alternative switches that can be used to offload part of a service chain's packet processing operations (see §8.1.3.3) and (*ii*) the actual rules to be installed in these switches. In this example, a routing change from path P_1 to P_2 (due to a routing update or a link failure between s1 and s3) will result in Metron installing 2 additional rules in s2 (these rules follow same logic with the rules in s3). Metron also updates the first rule of s3 by changing the inPort value to 1 rather than 0.

Backup configurations are kept in Metron's distributed store and are replicated across all the Metron controller instances in order to maintain a global network view. When a routing change or failure occurs, Metron applies the appropriate backup configuration. In §8.2.5 we show that Metron can install 1000 rules in less than 200 ms, hence quickly adapting to routing changes and (anticipated) failures, even those requiring a large number of rule updates.

8.2 Evaluation

In this section an evaluation strategy is presented, focusing on performance and scalability aspects of Metron. §8.2.1 describes how we implemented Metron, while §8.2.2 outlines the testbed used to conduct the experiments. In §8.2.3, §8.2.4, and §8.2.5 Metron's data plane performance, dynamic scaling, and deployment micro-benchmarks are presented respectively.

8.2.1 Implementation

The Metron controller is built on top of ONOS [48, 49], an open source, industrial-grade network operating system that is designed to scale well. Key to this decision was the fact that ONOS exposes unified abstractions for a large variety of network drivers that cover popular network configuration protocols, such as OpenFlow [13], P4 [86], Network Configuration protocol/YANG [216, 217], REST, and Simple Network Management Protocol (SNMP) [218]. ONOS was extended with a new driver that remotely monitors and configures NFV servers and their NICs. This driver is available at [204].

Metron's data plane extends FastClick [125]. The virtual machine device queues of DPDK [28] 17.08 is used to implement the hardware dispatching based on the values of the destination MAC address or VLAN ID fields. The Metron prototype (available at [219]) uses the former header field as a filter, because the large address space of a MAC address provides unique tags for trillions of service chains. To scale to 100 Gbps, Metron instructs the hardware classifier of a Mellanox NIC (§8.2.3.2).

8.2.2 Testbed

The testbed used for the experiments in this chapter consists of five identical servers, the technical characteristics of which were described earlier in Chapter 5.

Testbed at 10/40 Gbps
A testbed with a NoviFlow 1132 OpenFlow switch [220] (described in Chapter 5) is deployed and two servers are attached to this switch. The four ports of the first server are connected to the first four ports of the switch to inject traffic at 40 Gbps. Then, ports 5-8 of the switch are connected to the four ports of the second server, where traffic is processed by the NFV service chains being tested and sent back to the origin server through the switch. This testbed is shown in Figures 5.2d (i.e., normal service chains) and 5.2e (i.e., synthesized service chains by Metron using SNF). All the sub-sections in this section, but for §8.2.3.2 and §8.2.3.3, use this testbed.

Testbed at 100 Gbps
In §8.2.3.2, a 100 Gbps testbed is deployed using two back-to-back connected servers as discussed in Chapter 5 and shown in Figures 5.2b (i.e., normal service chains) and 5.2c (i.e., synthesized service chains by Metron using SNF).

The last server is used to run the Metron controller. §8.2.5 studies how switch diversity might affect Metron, by comparing the performance and capacity of a NoviFlow 1132 switch with an HP 5130 EI Switch [181] (described in Chapter 5), and the popular OVS [97] software switch. Each experiment was conducted 10 times and we report the 10^{th}, 50^{th} (i.e., median), and 90^{th} percentiles.

8.2.3 Metron Large-Scale Deployment

In this section we test Metron's performance at scale, focusing on two aspects: First, Metron's data plane performance is stressed using complex service chains with a large number of deeply-inspected (§8.2.3.1) and stateful (§8.2.3.2) traffic classes at 40 and 100 Gbps respectively. §8.2.3.3 evaluates Metron's placement on a set of topologies with a large number of nodes, on which hundreds to thousands of service chains are deployed.

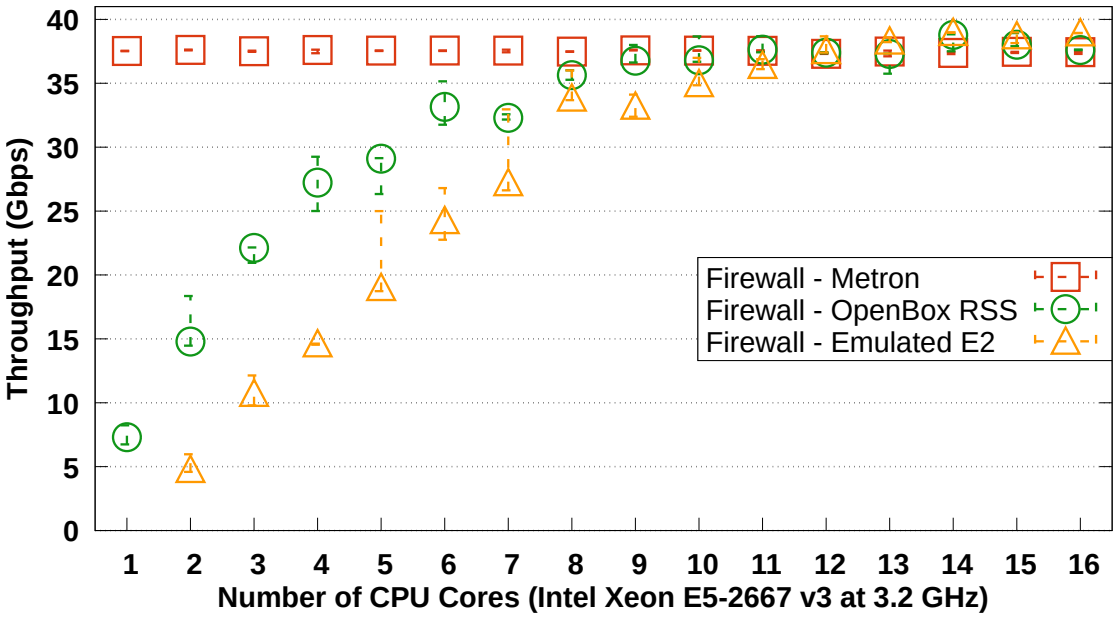
8.2.3.1 Deep Packet Inspection at 40 Gbps

To test the overall system performance at scale, a service chain of a campus firewall, followed by a DPI is deployed. The firewall implements access control using a list of 1000 rules, derived from an actual campus trace. The output of the firewall is sent to a DPI NF that uses a set of regular expressions similar to Snort (see [31]). Metron is compared against the same two state of the art systems discussed and evaluated in §4.1.2.2: (*i*) an accelerated version of OpenBox based on RSS and (*ii*) an emulated version of E2.

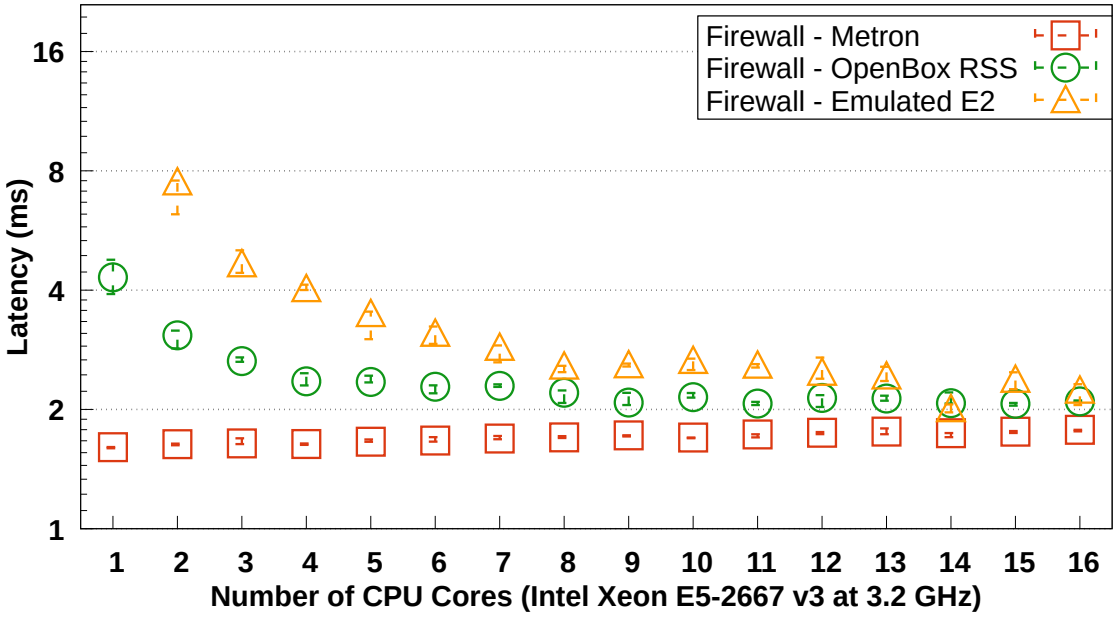
We injected a campus trace, obtained from University of Liège, that exercises all the rules of the firewall at 40 Gbps and measured the performance of the three approaches. First, only the firewall NF of this service chain is deployed to quantify the overhead of running this NF in software, as compared to an offloaded firewall (i.e., Metron). To fairly compare Metron against the other two approaches, a simple forwarding NF is started in the server, such that all packets follow the exact same path (generator, switch, server, switch, and sink) in all three experiments.

Figure 8.4a shows that OpenBox and the emulated E2 can realize this large firewall at line-rate. However, this is only possible if more than half of the server's CPU cores are utilized. Specifically, OpenBox requires 9 CPU cores, while the emulated E2 requires 11 CPU cores. In contrast, Metron completely offloads the firewall to the switch, hence easily realizing its ACL at line-rate; thus one core of the server is enough to achieve maximum throughput.

Looking at the latency of the three approaches in Figure 8.4b, it is evident that software-based dispatching (yellow triangles) incurs a large amount of unnecessary latency. Hardware dispatching using RSS (green circles) achieves substantially lower latency because it involves less inter-core communication. However, since the firewall executes heavy classification computations in software, OpenBox still exhibits high latency that cannot be decreased by simply increasing the number of cores. Specifically, using 16 CPU cores has comparable latency to 4 CPU cores. In contrast, Metron achieves nearly constant low latency (red squares) by exploiting the switch's ability to match a large number of rules at line-rate. This latency is 2.9-4.7x lower than the latency of the OpenBox and emulated E2 respectively,



(a) Throughput (Gbps) versus number of CPU cores.



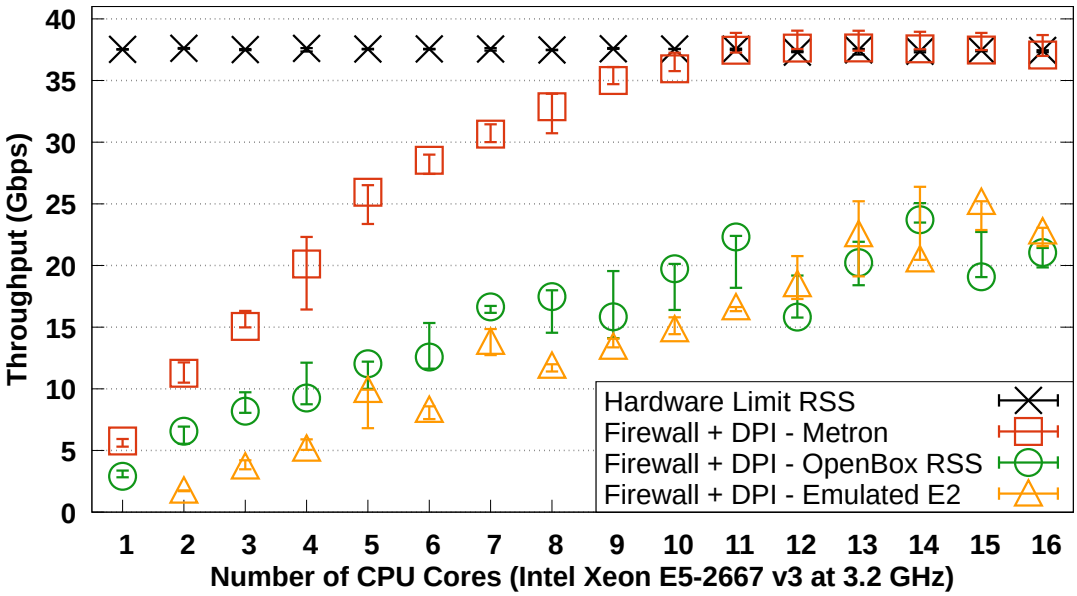
(b) Latency (ms) on a logarithmic scale versus number of CPU cores.

Figure 8.4: Performance of a campus firewall with 1000 rules using: (*i*) Metron with the firewall being offloaded, (*ii*) an accelerated version of OpenBox using RSS, and (*iii*) a software-based dispatcher emulating E2.

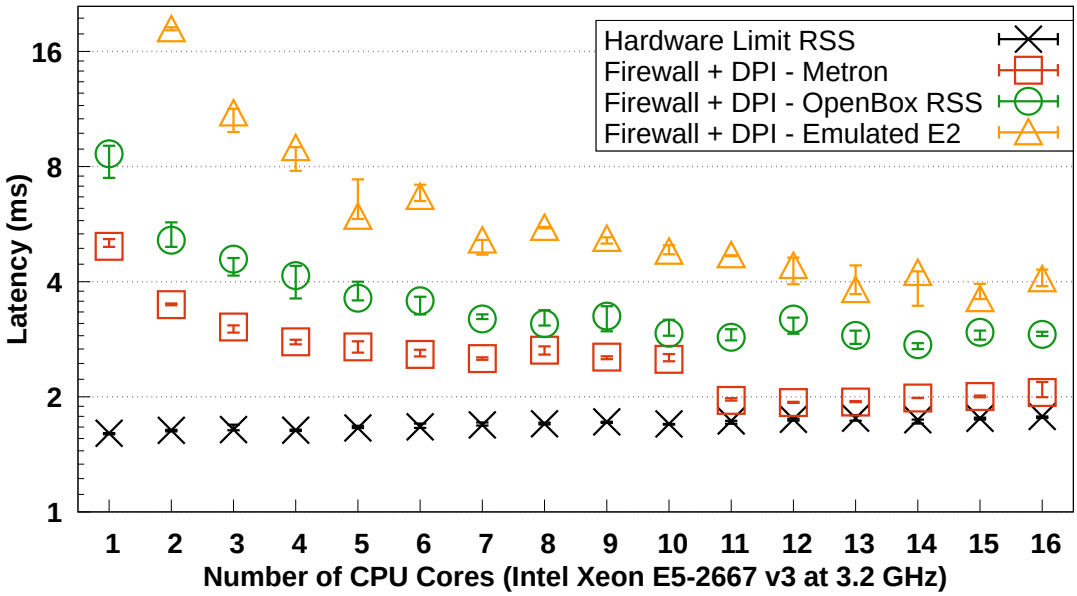
when each system uses one core for processing the NF (the emulated E2 requires 2 CPU cores in this case). At the full capacity of the server, the latency among the three systems is comparable; but Metron outperforms the emulated E2 and OpenBox by 30% and 19% respectively.

Next, the example campus firewall is chained with a DPI NF in order to realize the entire service chain. This chaining further pushes the performance limits of the three approaches as shown in Figure 8.5. In this scenario, Metron implements the DPI in software. First, we observe that even at the full capacity of the server, OpenBox and the emulated E2 can only achieve at most 25 Gbps (see Figure 8.5a). This performance is more than sufficient for a 10 Gbps deployment, hence some operators might not need the complex machinery of Metron. However, several studies indicate that large networks have already migrated from 10 to 40 Gbps deployments [165], while 100 Gbps networks are increasingly gaining traction [166]. In these higher data rate environments, these alternatives would require more than 16 CPU cores (and potentially more than one server) to achieve sufficient throughput, and are not guaranteed to scale because of the heavy processing requirements of large service chains.

Metron exploits the joint network and server capacity to scale even complex NFs, such as DPI, at the speed of the hardware. This can be confirmed by comparing the red squares (i.e., "Metron") with the black crosses (i.e., "Hardware Limit RSS") in Figure 8.5a). Most importantly, Metron requires only 10 CPU cores in a single machine to achieve this result, thus substantially shifting the scaling point for large service chains. The latency results in Figure 8.5b further highlight Metron's abilities. With 16 CPU cores, the Metron server deeply inspects all packets for this service chain at the cost of only 15.5% higher latency than the minimum latency of this testbed, shown with black crosses. At the same time, OpenBox and the emulated E2 incur 35-97% more latency than Metron, while achieving almost half of Metron's throughput. This difference increases rapidly when fewer CPU cores are utilized. For example, when each system uses one CPU core Metron achieves 75% lower latency than OpenBox and 358% lower latency than the emulated E2 respectively.



(a) Throughput (Gbps) versus number of CPU cores.



(b) Latency (ms) on a logarithmic scale versus number of CPU cores.

Figure 8.5: Performance of a campus firewall with 1000 rules followed by a DPI at 40 Gbps, using: (*i*) Metron with the firewall being offloaded, (*ii*) an accelerated version of OpenBox using RSS, and (*iii*) a software-based dispatcher emulating E2. "Hardware Limit RSS" showcases the speed of the hardware, using the firewall NF offloaded into hardware followed by an RSS-based forwarding NF.

8.2.3.2 Stateful Service Chaining at 100 Gbps

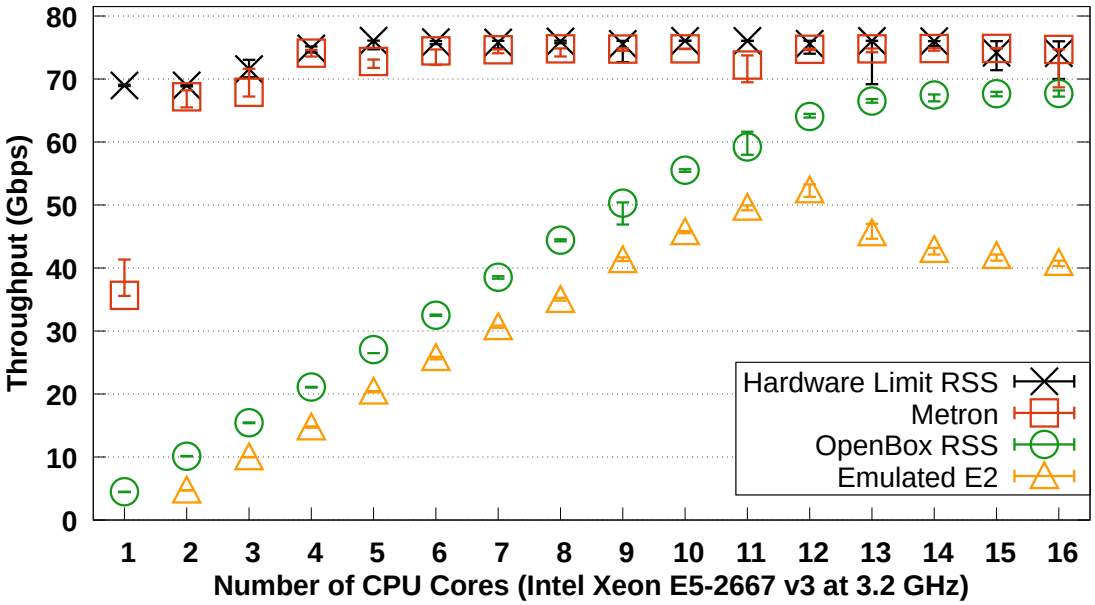
In this section we further stress the performance of Metron, OpenBox, and the emulated E2 systems by conducting an experiment at 100 Gbps. To achieve this new performance target we use a different testbed, as described in §8.2.2. Specifically, two of our servers are equipped with a 100 GbE Mellanox ConnectX-4 MT27700 card and are connected back-to-back. The first server acts as a traffic generator and receiver, while the second server is the device under test.

Four million packets were analyzed from the campus trace used in §8.2.3.1 and 3117 distinct destination IP addresses were found. Then, a standards-compliant router was implemented and its routing table was populated with these addresses. The router was chained with two stateful NFs: a NAPT and an LB that implements a flow-based round robin policy. In this scenario, Metron can only offload the routing table of the router to the Mellanox NIC using DPDK's flow director. Unlike the NIC RSS redirection table, flow director provides explicit flow control and substantially larger capacity. The remaining functions of the router (e.g., Address Resolution Protocol handling, IP fragmentation, TTL decrement, etc.) together with the stateful NFs (i.e., NAPT and LB) are executed in software.

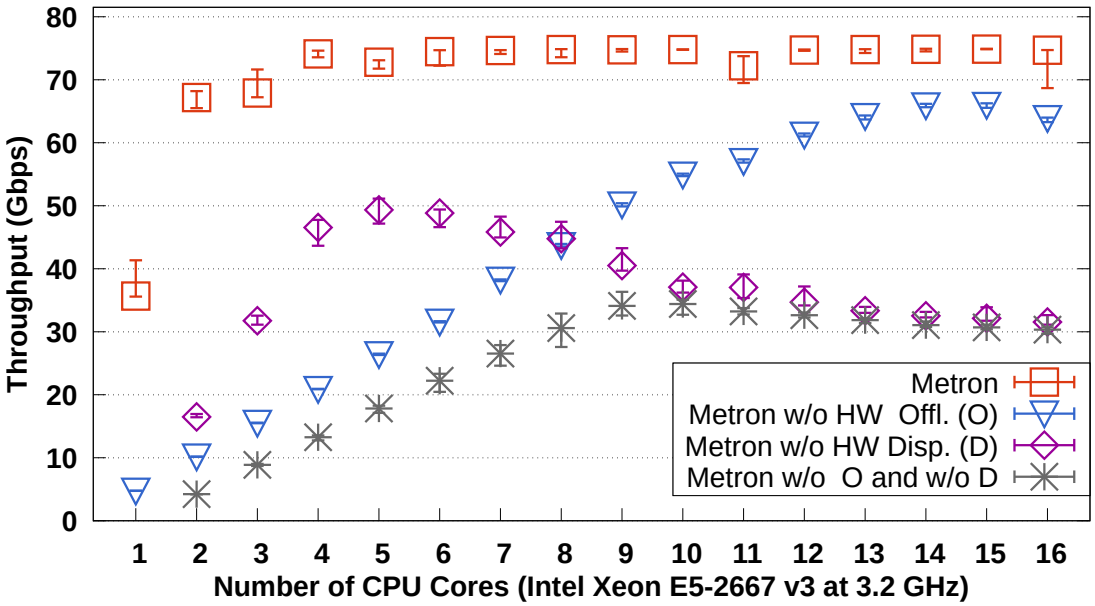
Metron vs. State of the art

The throughput achieved by the three systems is shown in Figure 8.6a. For comparison, we also show the throughput of the server when a simple RSS-assisted forwarding NF is used to send traffic back to its origin. These results show a slow but linear increase of the throughput with an increasing number of CPU cores for both OpenBox and the emulated E2 approaches. Using linear regression on the medians between 1 and 12 CPU cores (the emulated E2 starts from 2 CPU cores), we found that the throughput of OpenBox increases by 5.37 Gbps with each additional core, while the emulated E2 increased by 4.91 Gbps per core. However, in both cases using more than 12 CPU cores does not bring further performance gains. Specifically, the throughput of OpenBox plateaus around 67 Gbps, while the performance of the emulated E2 drops (from 53 to 41 Gbps). Moreover, with 13-16 CPU cores, the latency of the two systems increases up to 56% for OpenBox and up to 25% for the emulated E2.

In contrast, Metron achieves 75 Gbps throughput using only a small fraction of the server's CPU cores. Key to this performance is Metron's hardware dispatcher in the NIC, which offers two advantages: (*i*) it saves CPU cycles by performing the lookup operations of the router and (*ii*) it load balances the traffic classes matched by the hardware classifier across the available CPU cores. Exploiting these advantages allows Metron (i.e., red squares in Figure 8.6) to quickly match the performance of the "Hardware Limit RSS" case (i.e., black points in Figure 8.6a) using only two CPU cores, despite running several stateful operations (i.e., NAPT and LB). Moreover, according to a performance report by Mellanox [221], our NIC



(a) Comparison of: (*i*) Metron, (*ii*) OpenBox with RSS, and (*iii*) a software-based dispatcher emulating E2. "Hardware Limit RSS" shows the forwarding speed of the server (i.e., no service chain) using RSS as a traffic dispatcher.



(b) Metron's hardware offloading (O) and hardware dispatching (D) contributions to the overall system's performance. The word "without" is abbreviated as "w/o".

Figure 8.6: Throughput (Gbps) of a stateful service chain (Router→NAPT→LB) at 100 Gbps.

achieves line-rate throughput with frames greater than 512 bytes. Therefore, the 75 Gbps limit reached in this experiment with the campus trace is mainly due to the large number of small frames (26.9% of the frames are smaller than 100 bytes, while 11.8% of them are in (100, 500]). Finally, Metron's latency plateaus at a sub-millisecond level, which is 21-38% lower than the lowest latency achieved by the other two systems (see Figure 1.7b).

Dissecting Metron's Performance

To quantify the factors that contribute to Metron's high performance, an additional experiment was conducted using the same testbed, input trace, and service chain. The results are depicted in Figure 8.6b. Note that the red curves (i.e., Metron's throughput) in Figures 8.6a and 8.6b are identical. The purpose of Figure 8.6b is to showcase what performance penalties occur when one starts removing our key contributions from Metron, as follows:

1. Metron without hardware offloading (i.e., blue triangles in Figure 8.6b).
2. Metron without hardware dispatching to the correct core (purple rhombs in Figure 8.6b).
3. Metron without both hardware offloading and dispatching (gray stars in Figure 8.6b).

Comparing "Metron" vs. "Metron w/o HW Offl." quantifies the benefits of Metron's hardware offloading feature. In the "Metron w/o HW Offl." case input packets are still dispatched to the correct core (using the Flow Director component of the Mellanox NIC), but each core executes the entire service chain logic in software. The throughput achieved in this case (i.e., blue triangles in Figure 8.6b) is comparable with the throughput of the "OpenBox RSS" case shown in Figure 8.6a. A key difference between these cases is that "Metron w/o HW Offl." performs the routing table lookup twice; once in the NIC for traffic dispatching and the second in software (to disable hardware offloading), after packets are dispatched to the correct CPU core. In contrast, OpenBox uses RSS for dispatching and implements the routing table only once in software. Neither of these cases exploits the available capacity of the NIC to offload the routing operations, thus costing CPU cycles.

Next, the comparison between "Metron" and "Metron w/o HW Disp." cases highlights the cost of inter-core communication, by deliberately choosing an incorrect core. "Metron w/o HW Disp." implements the routing lookup in hardware (i.e., hardware offloading is enabled), hence reducing the processing requirements of the software part of the service chain. However, this case exhibits a serious bottleneck compared to Metron, as it requires a software component to (re-)classify input packets to decide which CPU core processes them (i.e., software dispatching similar to the emulated E2 case in Figure 8.6a). As shown in Figure 8.6, both "Metron w/o HW Disp." and the emulated E2 cases exhibit similar performance degradation as their software dispatcher communicates with

an increasing number of CPU cores. This degradation appears earlier for "Metron w/o HW Disp." (i.e., after 5 cores versus 12 cores for the emulated E2 case). This is because "Metron w/o HW Disp." offloads part of the service chain's processing to the NIC, hence the inter-core communication bottleneck appears sooner. In contrast, Metron exploits the ability of the NIC to directly dispatch traffic to the correct core, thus avoiding the need for a software dispatcher and the concomitant inter-core communication.

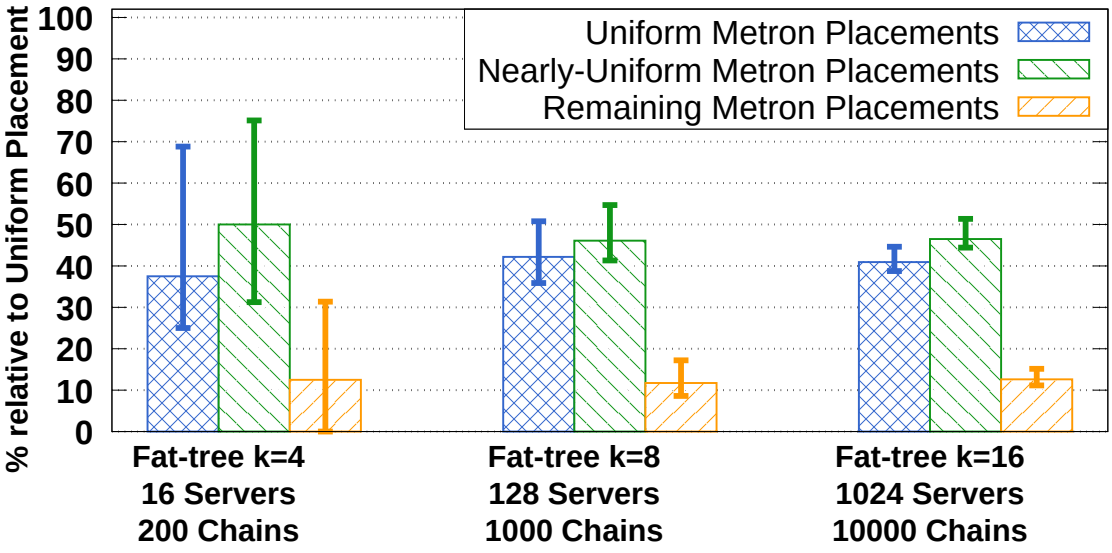
Finally, the "Metron w/o O and w/o D" case in Figure 8.6b shows the throughput attainable when both hardware offloading and accurate dispatching features are disabled. In this case, input packets are always sent to an "incorrect" core (specifically the core where the software dispatcher runs) and the entire service chain runs in software. The inter-core communication bottleneck manifests itself once again, this time after using 9 or more CPU cores.

Key Outcome
As explained in §8.1, Metron's ability to scale complex (i.e., DPI) and stateful (i.e., NAPT and LB) NFs is due to the way that the incoming traffic classes are identified, tagged, and dispatched to the CPU cores in a load balanced fashion. Metron's ability to realize these service chains at the *NIC's hardware limit* with a single server is an important achievement.

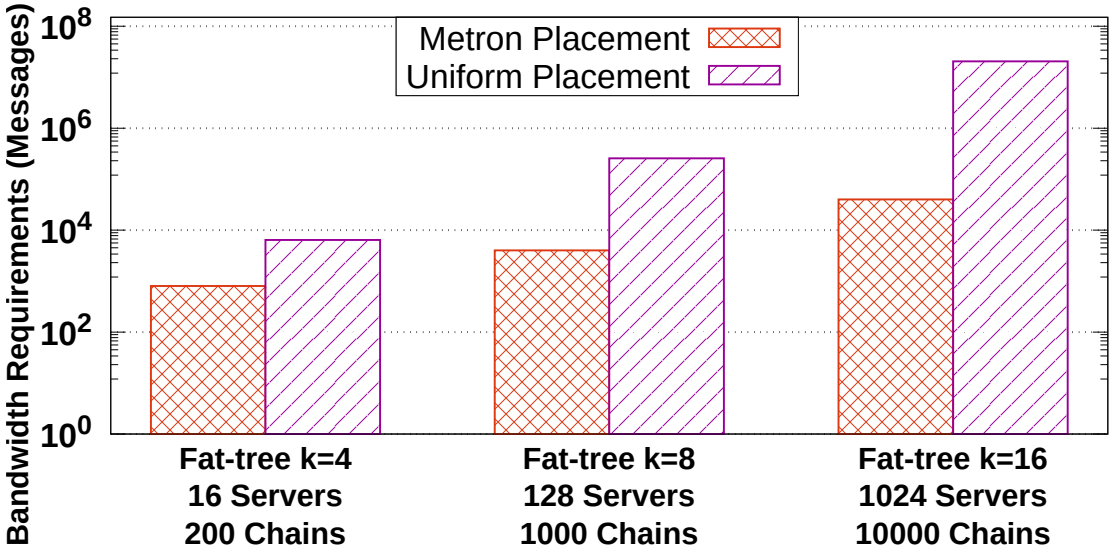
8.2.3.3 Metron's Placement in Large Networks

To verify that the performance of Metron's placement scheme (see §8.1.3.3) can be generalized to real and potentially large networks, experiments that emulate Metron's service chain placement in datacenters are conducted, using fat-tree topologies of increasing sizes (see Figure 8.7). Our analytic study shows how close Metron's placement decisions are compared to uniform placement and what bandwidth requirements each approach demands for a large number of service chains. Note that the uniform placement allocates equal number of CPUs from the available servers, while a nearly uniform placement exhibits the least distance from the uniform. Note also that our approach is not restricted to datacenter topologies; Metron's placement is topology-agnostic.

Figure 8.7a compares Metron's placement with the uniform placement policies with increasing number of servers (i.e., 16, 128, and 1024) and service chains (i.e., 200, 1000, and 10000). The first of each set of bars indicate that Metron's placement decisions match the uniform ones with ~40% median probability, regardless of the network's size and number of service chains to be placed. For 16 servers, the upper percentile indicates that Metron makes a uniform decision with 70% probability. According to the other two sets of bars, most of the remaining decisions made by Metron fall very close to uniform (i.e., middle set of



(a) Metron's placement relative to the uniform placement policy. Metron makes uniform or nearly uniform (with the least distance from uniform) placements with ~90% median probability.



(b) Bandwidth requirements on a logarithmic scale with increasing number of servers and service chains.

Figure 8.7: Placement performance and bandwidth requirements on three fat-tree topologies of increasing number of servers (i.e., 16, 128, and 1024), when using (*i*) Metron or (*ii*) the uniform (equal number of CPU cores per server) placement scheme to deploy a large number of service chains.

bars), confirming that our placement policy makes reasonably balanced decisions, despite its "limited" randomness.

Figure 8.7b shows the bandwidth savings of our placement policy, compared to the uniform one. To make a uniform placement decision, a controller has to query the CPU availability from all the available servers, thus, incurring a communication overhead proportional to the network size (which quickly becomes infeasible for large networks). This overhead is shown by the second of each set of bars in Figure 8.7b. To reduce this overhead, we trade-off some accuracy in placement to minimize Metron's bandwidth requirements. The first of each set of bars in Figure 8.7b shows that Metron requires orders of magnitude less bandwidth than the uniform policy to place a large number of service chains on these networks. An indirect (but important) benefit of our low overhead placement is that, by querying only 2 servers at a time, a minimal number of events at the servers is generated, hence preserving processing cycles for other tasks. The next section showcases how each Metron server dynamically adapts to the input load, even when the placement does not perfectly balance the load.

8.2.4 Metron's Dynamic Scaling

Next, we evaluate Metron's dynamic scaling strategy (introduced in §8.1.3.4) using a scenario with a service chain configuration taken from an ISP [193], targeting a 10 Gbps network. The target service chain consists of an ACL with 725 rules, followed by a NAPT gateway that interconnects the ISP with the Internet, while performing source and destination address and port translation and routing.

This service chain was deployed on a single server connected to a NoviFlow switch (see §8.2.2), to which a real trace was injected at variable bitrates. The solid curve in Figure 8.8 shows the throughput corresponding to the rate at which the trace was injected, while the dashed curve depicts the throughput achieved by Metron. To highlight Metron's ability to provision resources on demand, we plot the number of cores allocated by Metron over the course of the experiment (yellow circles and right-hand scale).

The experiment begins with an allocation of 4 CPU cores (precalculated based upon the initial injection rate). Following this, the Metron controller makes dynamic decisions based on monitoring data gathered from the data plane and dynamically modifies the mapping of traffic classes to tags (thus affecting load distribution). In this experiment Metron requires between 1 and 6 CPU cores to accommodate the input traffic. In some cases, Metron fails to immediately adapt to sudden spikes, thus we observe a slight lag in Metron's reactions (e.g., as shown in the interval between 84 and 90 seconds). This occurs due to two reasons. First, because Metron's scaling approach involves interaction between

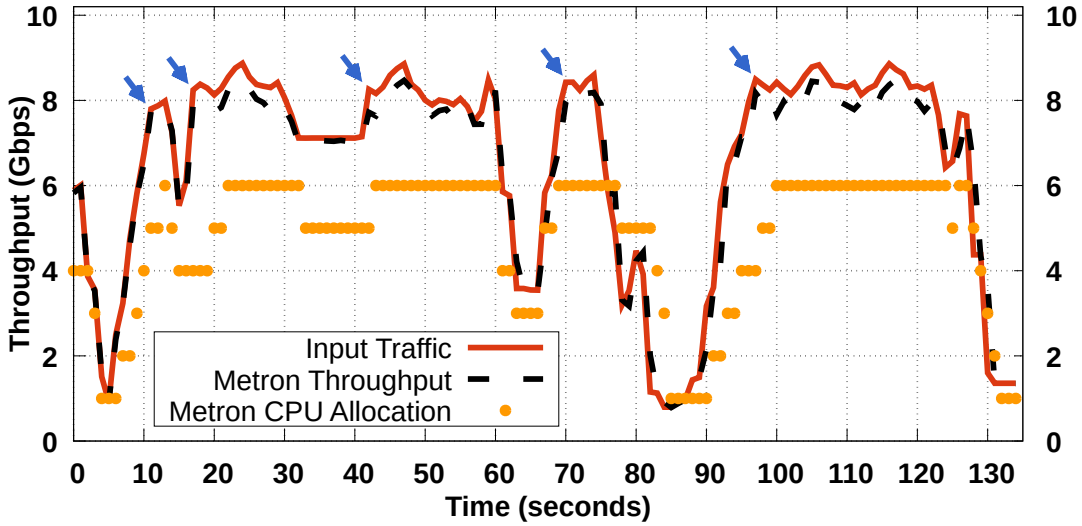


Figure 8.8: Metron under dynamic workload. Blue arrows indicate load spikes throughout the experiment.

the controller and the involved nodes (i.e., the server and the switch) in order to establish the CPU affinity of the traffic classes. To avoid overloading the controller, this interaction occurs every 500 ms, which contributes to the observed lag. Second, every newly-allocated CPU core can accommodate a large amount of input traffic, therefore CPU core allocation changes need not be as frequent as input load changes. However, Metron's throughput is not substantially affected by this lag (the blue arrows indicate the upward spikes in load at 10, 17, 42, 70, and 97 seconds). As confirmed in §8.2.5.2, Metron is able to quickly install the necessary rules to enforce the traffic class affinity.

8.2.5 Deployment Micro-benchmarks

This section benchmarks how quickly Metron carries out important control and data plane tasks, such as hardware and software (re)configuration, in a fully automated fashion.

8.2.5.1 Impact of Increasing Number of Traffic Classes

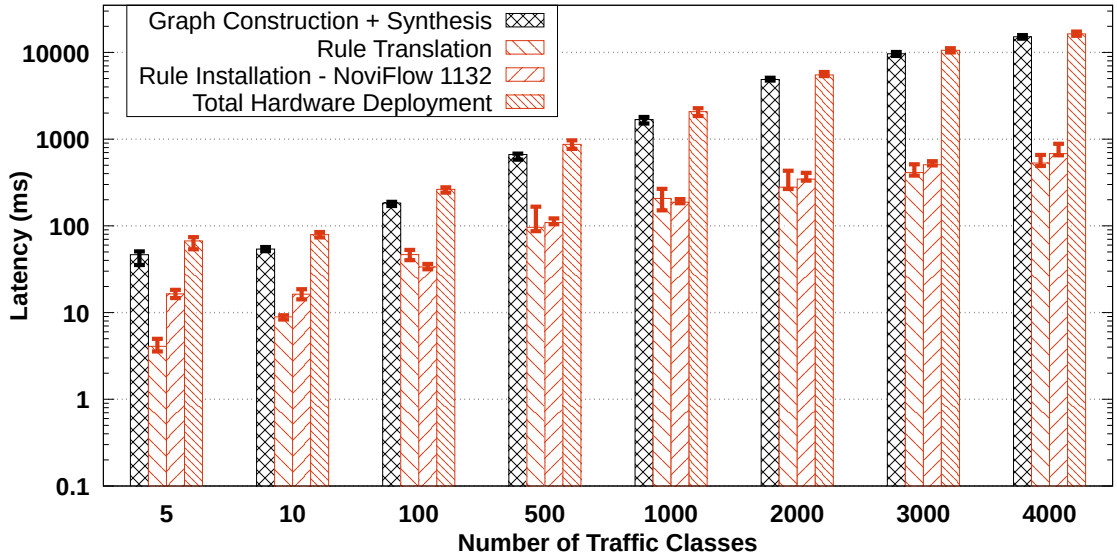
To study the impact of increasingly complex service chains on Metron's deployment latency, a firewall with an increasing number of rules (up to 4000, derived from actual ISP firewalls [193]) is used. The time between when a request to deploy this NF is issued by an application and the actual NF deployment either in hardware or in software is measured.

In either case, the first task of Metron is to construct and synthesize the packet processing graph of the service chain (as per §8.1.3.1), as depicted in the first of each group of bars (in black) in Figures 8.9a and 8.9b. This latency is the dominant latency in both hardware and software-based deployments (see the last set of bars in each figure). Fortunately, this is a one time overhead for each unique service chain; considering the importance of generating such an optimized processing graph, Metron precomputes and stores the synthesized graph for a given input in its distributed database.

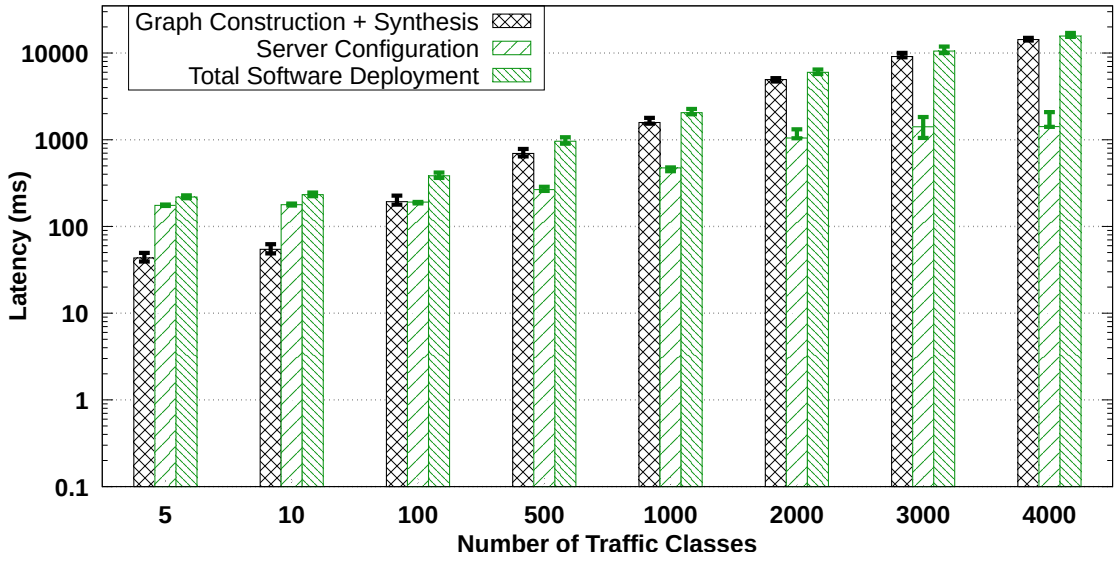
Apart from this fixed latency operation, a purely hardware-based deployment, requires two additional operations, as shown in Figure 8.9a. The first operation is the automatic translation of the firewall's synthesized packet processing graph into hardware instructions targeting our OpenFlow switch (the second bar in each set of bars). This operation involves building a classification tree that encodes all the conditions of the firewall rules, therefore it has logarithmic complexity with the number of traffic classes. For example, under the specified experimental conditions, the median time to encode a large firewall with 4000 traffic classes is around 500 ms. The last operation in the hardware-based deployment is the rule installation in the OpenFlow switch (the third bar in each set of bars in Figure 8.9a). Note that even entry-level OpenFlow switches, such as the one used, can install thousands of rules per second; a more thorough study is provided in §8.2.5.2, where the effects of hardware diversity on Metron are discussed.

For a purely software-based deployment of this same service chain, the time following graph construction and synthesis until the service chain is deployed at a designated server is considered. This latency is labeled "Server Configuration" in Figure 8.9b. Note that it takes longer per rule than for the corresponding hardware-based case for a small number of traffic classes because there is a fixed overhead to start a secondary DPDK process (i.e., a Metron slave) at the server. This overhead is ~180 ms as can be seen from the case of 5 traffic classes. However, the (median) deployment time is 0.471 ms/rule (versus 0.411 ms/rule for the hardware case shown in Table 8.2), hence a large firewall deployment takes a comparable amount of time either in software or hardware.

Overall, apart from the one-time precomputation overhead for constructing and synthesizing a service chain, the worst case deployment time of a firewall with 4000 traffic classes is less than 1200 ms, whereas only 100-200 ms is required for hundreds of traffic classes.



(a) Hardware-based deployment on a NoviFlow 1132 switch.



(b) Software-based deployment on a 16-core Intel Xeon E5-2667 v3.

Figure 8.9: Latency (ms) on a logarithmic scale for different Metron deployments of a service chain with increasing complexity.

8.2.5.2 Diversity of Network Elements' Capabilities

Network elements from different vendors and of different price levels might offer different possibilities for NFV offloading. In this section the hardware-based deployment shown in Figure 8.9a is repeated, where the NoviFlow switch is replaced with either a hybrid HP 5130 El hardware switch or the software-based OVS. Figure 8.10 shows the rule installation latency of these three switches. Table 8.2 summarizes these results along with key characteristics of these switches, as they affect Metron's deployment choices and performance.

The NoviFlow switch contains 55 OpenFlow tables, each with 4096 entries (i.e., 225280 rules in total), while the HP switch has a single OpenFlow table with either 512/256 entries for IPv4/IPv6-based rules or 16384 entries for L2 rules. The capacity of OVS depends on the amount of memory that the host machine provides; modern servers provide ample DRAM capacity to store millions of rules.

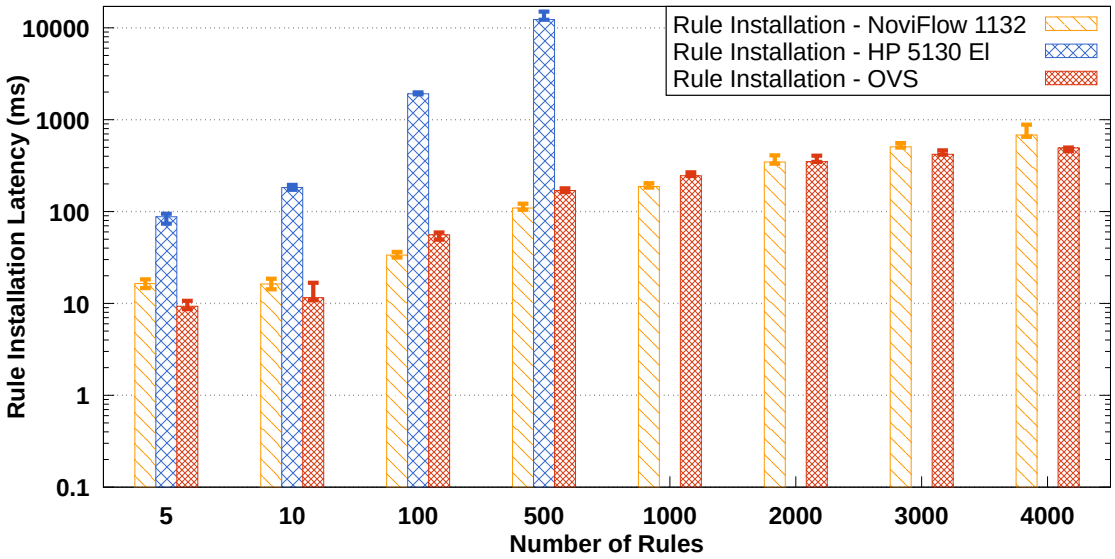


Figure 8.10: Rule installation latency of (*i*) a NoviFlow 1132, (*ii*) an HP 5130 El, and (*iii*) the OVS switches. The HP switch does not have enough capacity to accommodate more than 512 IPv4-based traffic classes.

Table 8.2: Comparison of 3 switches used by Metron. The last column states the median rule installation speed of these switches.

Switch		Capacity (# of Rules)	Rule Installation Speed (ms/rule)
Model	Type		
NoviFlow 1132 [220]	HW	225280	0.411
HP 5130 El [181]	HW	256/512/16384	50.250
OVS [97] v2.5.2	SW	Memory-bound	0.263

The median rule installation speed of the NoviFlow switch is substantially higher than HP (0.411 vs. 50.25 ms/rule), with the difference being more than two orders of magnitude. However, this difference is partially reflected in the price difference between the two switches (approximately US$ 15000 vs. US$ 2000). OVS is open source, achieves lower data plane performance, but outperforms both hardware-based switches in terms of median rule installation speed (0.263 ms/rule), when running on the processor described for the testbed in §8.2. This finding is confirmed by earlier studies [222, 223], where the rule installation speed varied especially when priorities are involved. In this test, Metron installed rules of the same priority and low variance was observed.

In summary, today's OpenFlow switches provide Metron with fast median rule installation speed and sufficient capacity at different price/performance levels.

8.3 Originality and Open Source Contributions

In this section, the originality of Metron is highlighted with respect to earlier efforts, discussed in Chapter 3.

8.3.1 NFV Management

E2 [32] and Metron manage service chains mapped to clusters of servers interconnected via programmable switches. E2 only partially exploits OpenFlow switches to perform traffic steering.

> In contrast, Metron fully exploits the network (i.e., OpenFlow switches and NICs) to both steer traffic and to offload and load balance NFV service chains, while deliberately avoiding E2's inter-core transfers.

8.3.2 NFV Consolidation

OpenBox [31] merges similar packet processing elements into one, thus reducing redundancy. Our earlier work SNF [29] eliminates processing redundancy by synthesizing multiple NFs as an optimized equivalent NF. Slick [140] and CoMb [137] propose NF consolidation schemes, although these schemes reside higher in the network stack.

> We integrated SNF into Metron, since this is the most extensive consolidation scheme to date. Metron effectively coordinates these optimized pipelines at a large-scale, while exploiting the hardware.

8.3.3 Hardware Programmability

Several approaches for hardware programmability were discussed in §3.2. In summary, OpenFlow switches and NICs provide APIs for stateless match-action rules, while RMT [82], P4 [86], OpenState [83], OPP [224], and FlexNIC [104] introduce stateful packet processing in the data plane.

> All these works have made phenomenal progress towards exposing hardware configuration knobs. Metron acts as an umbrella to foster the integration of this diverse set of programmable devices into a common management plane. In fact, our prototype [204, 219] integrates OpenFlow switches, DPDK-compatible NICs, and commodity servers. Thanks to ONOS's abstractions, additional network drivers can be easily integrated.

8.3.4 Hardware Offloading

As discussed in §3.2.3, earlier research efforts have proposed ways to offload specific functions into commodity hardware. These functions range from simple e.g., IP checksum offloading [126] to cryptographic functions [127] and key-value store operations [128, 129, 104], sometimes using auxiliary hardware components such as GPUs [123, 124, 130, 131] or Smart NICs [105, 106, 108, 107].

> We envision these works as future components of Metron to extend its offloading abilities.

ClickNP [225] showed how to achieve high performance packet processing by completely migrating NFV into reconfigurable, but specialized, FPGA-based hardware devices. Microsoft Azure also uses a combination of FPGA-based chips and software to implement VM network policies in their datacenters [167, 19].

> In contrast, our philosophy is to explore the boundaries of commodity hardware. Therefore, Metron performs stateful processing in software but combines it with smart offloading and dispatching using commodity hardware. Metron meets performance levels achieved by specialized hardware at the cost of commodity hardware.

8.3.5 Server-level Solutions

A summary of server-level NFV solutions, introduced in §3.2.2, is provided in this section. Flurries [117] builds atop OpenNetVM [115] to provide software-based service chains on a per-flow basis, while ClickOS [111] and NetVM [112] offer NFs running in VMs. NFP [118] extends OpenNetVM to allow NFs in a service chain to be executed in parallel. Dysco [119] proposes a distributed protocol for steering traffic across the NFs of a service chain. NFVnice [154] and our earlier work SCC [27, 155] are efficient NFV schedulers. Click-based [42] approaches have proposed techniques to exploit multi-core architectures [125, 196, 110].

None of these works have explored the possibility of using hardware to offload parts of a service chain, nor do they support our optimized flow affinity approach.

8.3.6 Industrial Efforts

As discussed in §3.4, CORD [149] and OPNFV [150] are industrial NFV projects based on OpenStack [148].

Metron and CORD share common controller abstractions (i.e., ONOS); however, we avoid OpenStack's virtualization by integrating native DPDK-based solutions. Unlike CORD, the Metron controller leverages placement techniques with minimal overhead (see §8.1.3.3 and §8.2.5) and sophisticated NF consolidation (see §8.1.3.1) to achieve high performance.

8.3.7 Summary of Open Source Contributions

A list of open source contributions related to Metron is presented below:

1. Metron's high performance data plane is available at [219], as an extension to FastClick [226].
2. A driver for managing CPU and NIC resources on commodity servers is part of the Metron controller [204]. The Metron controller itself is developed on top of the ONOS SDN controller [49].

A tutorial on how to use Metron controller's driver along with Metron's data plane is available at [227].

Chapter 9

Contributions

This chapter challenges the hypotheses of this thesis in §9.1, while §9.2 describes the publication status for parts of this thesis.

9.1 Challenging the Hypotheses

Table 9.1 summarizes the contributions of this thesis, with an emphasis on the results of SCC, SNF, and Metron as presented in Chapters 6, 7, and 8 respectively. Using this table, I challenge the hypotheses stated in §4.5 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, complexity, and processing model of the service chain", as defined in §4.5. Then, in the same section I aimed to disprove H_0, by showing the hypothesis H_1: "Some service chains can be realized without their performance deteriorating despite the length and complexity of the chain, when using an appropriate processing model".

SCC Contributions

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 insufficient to support H_1, I found several encouraging outcomes:

Outcome 1: SCC substantially reduces the latency and jitter of the service chains under test (see Table 9.1). This result has direct impact on popular service chains that rely on commodity network drivers [173].

Outcome 2: The SCC Profiler's results suggest that only service chains based on fast network drivers, such as DPDK, can achieve the desired outcomes, if their own performance problems can be overcome.

Table 9.1: A summary of the contributions of this doctoral thesis on the performance of NFV service chains.

Contribution	Achievements
Profiling [27]	• 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.
I/O multiplexing [27]	• 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.
Scheduling [27]	• 30-300% lower latency and up to 40x lower jitter for service chains interconnected with OVSK. • 10-25% lower latency and 2x lower jitter for service chains interconnected B2B.
Synthesis of I/O and processing operations [29]	• Multi-core NFV with zero I/O and processing redundancy. • 10 chained routers or NAPTs at the cost of one, achieving line-rate 40 Gbps throughput. • ISP-level service chains in software with bounded median latency between 100-500 μs. • ISP-level service chains with hardware assistance with bounded median latency below 100 μs and line-rate 40 Gbps throughput.
Offloading and accurate traffic dispatching [30]	• Synthesized NFs globally coordinated by a controller. • Early hardware offloading and tagging of a service chain's stateless part. • Tag-based hardware dispatching to the correct CPU cores. • Deep packet inspection at the speed of a 40 Gbps testbed. • Stateful packet processing at the speed of a 100 Gbps testbed. • Up to 4.7x lower latency, up to 7.8x higher throughput, and 2.75-6.5x better efficiency than the state of the art.
Load balancing [30]	• Traffic class-level load balancing by dynamically manipulating the map of traffic classes' tags to CPU cores.
Placement [30]	• Low-cost probabilistic service chain placement.

The SCC I/O multiplexing results shown in Table 9.1 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 I/O 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 suggest that more drastic approaches are necessary.

SNF and Metron Contributions

Using my first contribution as a base, we then attempted to radically revise the way NFV service chains are realized, by proposing SNF and Metron in Chapters 7 and 8 respectively. The synthesis approach presented in §7.2 realized service chains with zero I/O and processing redundancy, while Metron effectively mapped the operations of synthesized service chains to available hardware resources (see §8.1). As a result, Metron and a hardware-assisted SNF realize service chains at the speed of the hardware, as shown in §8.2 and §7.5 respectively. In the following paragraphs I make a connection between these results and the hypothesis H_1.

The H_1 hypothesis has three distinct parts. The first part relates the performance of a service chain with its length. §7.5.2 demonstrated how SNF realizes service chains of *increasing length* without performance degradation. As also shown in Table 9.1, these results support the first part of H_1 for service chain lengths in the range $[1, 10]$.

The second and third parts of H_1 relate the performance of a service chain with its complexity and processing model. In §7.5.4 three example ISP-level service chains of increasing complexity are measured using two processing models: a software-based and a hardware-assisted SNF. As shown in §7.5.4.1, the software-based SNF cannot realize these complex service chains without performance degradation at high input rates (i.e., 40 Gbps). This occurs because of two reasons: (*i*) the complex traffic classifiers of these service chains incur performance penalties when realized in software and (*ii*) the processing model of the software-based SNF might involve costly inter-core communication, which further impacts the performance of these service chains.

In contrast, a hardware-assisted SNF, as proposed in §7.5.4.2, solved these two problems by offloading traffic classification into an OpenFlow switch and by statically load balancing the output traffic classes of the hardware classifier across 4 servers, each using RSS to distribute the load across its CPU cores. Consequently, the hardware-assisted SNF provides some evidence that supports 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).

Metron introduced an even more efficient processing model than the hardware-assisted SNF. Metron's processing model exploits programmable switches and NICs to classify, tag, and accurately dispatch input traffic across a server's CPU cores with zero inter-core communication. Doing so allows Metron to realize (*i*) Firewall→DPI at the speed of a 40 Gbps testbed (see §8.2.3.1) and (*ii*) a stateful service chain (i.e., Router→NAPT→LB) at the speed of a 100 GbE NIC on a single server (see §8.2.3.2). These results demonstrate multi-fold latency reduction and throughput increase (at the maximum attainable levels of throughput) compared to the state of the art.

Meeting the Performance Requirements of the 5G Networks
Both Metron and SNF showed that a variety of *complex* service chains can be realized without performance degradation, when an appropriate packet processing model is utilized. As a result, I believe that SNF and Metron provide sufficient evidence to support H_1. To put this thesis into perspective, in §4.3.3 I 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 9.1, this thesis 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 only 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 exceeds 800 μs for the example service chains (see §7.5.4.2).

Outcome 5: Stateful service chains at the speed of a 100 GbE NIC incur sub-millisecond median latency of $\sim$ 650 μs (see Figure 1.7b and §8.2.3.2) using only a small fraction of a single server's CPU cores.

9.2 Thesis Publications

SCC and SNF were published to international journals. These two journal articles were also part of the licentiate stage of this thesis [155]. At the doctoral stage of this thesis, Metron was accepted at a top networked systems conference. The submission details and the status of each work are shown in Table 9.2.

Table 9.2: Thesis publications and their status.

Contribution	**Target Venue/Journal or Institution**	**Submission Date**	**Status**
SCC	Elsevier Journal of Systems and Software	September 3, 2016	Accepted (Available at [27])
SNF	PeerJ Computer Science Journal	September 23, 2016	Accepted (Available at [29])
Licentiate Thesis	KTH Royal Institute of Technology	November 3, 2016	Successfully defended (Available at [155])
Metron	15th USENIX Symposium on Networked Systems Design and Implementation (NSDI) 2018	September 25, 2017	Accepted (Available at [30])

Chapter 10

Limitations and Future Work

SCC, SNF, and Metron are the three main pillars of this dissertation (see Chapters 6, 7, and 8 respectively). §10.1 discusses the limitations of these three works. In §10.2 a plan for future work that builds upon this thesis is sketched.

10.1 Limitations

The limitations of this dissertation are described in the following paragraphs.

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 specifically designed to collect data using Intel-specific performance counters. We adopted this hardware-specific approach trading generality for high-precision profiling results. These results can assist cloud providers by automatically uncovering performance problems of NFV services running on Intel platforms; however, the emergence of ARM and other processor platforms in datacenters cannot be analyzed using the current implementation of SCC.

SNF Limitations

SNF does not attempt to synthesize arbitrary software components, but rather targets a broad but finite set of middlebox-specific NFs that operate on a packet's header. SNF makes three 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.
3. An NFV controller must undertake to configure a target data plane and deploy a desired SNF service chain. This limitation is eliminated by our latest work Metron [30], which integrates SNF in its control plane and performs network-wide placement of NFV service chains.

Metron Limitations
Since Metron uses SNF to synthesize NFV service chains, it inherits the first two limitations of SNF. Unlike SNF, Metron supports both Click-based and blackbox NFs. However, the current Metron prototype exhibits another limitation related to blackbox NFs. Specifically, when a blackbox NF is located between two Metron service chains, Metron's stateless (i.e., flow classification) and stateful (i.e., flow modifications) parts need to be applied twice (i.e., before and after the blackbox NF). This breaks the zero processing redundancy principle that Metron inherits from SNF. §10.2 suggests a future research direction to solve this problem.

10.2 Future Work

This section sketches some plans for future work, as a follow up on this thesis.

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 [95] 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.

Metron Extensions In §10.1, we explained how a blackbox NF between two Metron service chains might introduce processing redundancy due to the fact that traffic classification and modification operations need to be performed at least * twice. This is an important problem that network operators might face while integrating NFV solutions with legacy infrastructure at scale. We will extend Metron to allow deep service chain integration with blackbox NFs. This is a challenging task because blackbox NFs introduce uncertainty in the way packets are modified. To clarify this, consider a case in which Metron has reserved a particular header field (e.g., destination MAC address) to act as tag. When traffic enters this blackbox NF, after being tagged by a Metron instance, the tagging information is overwritten, hence a subsequent Metron instance following this blackbox NF will not be able to properly load balance input traffic based upon the value of the tag. Metron will be extended to solve this consistency issue allowing Metron deployments at scale.

Vision: We envision end-to-end "logical" Metron service chains which begin from the packet gateways at the access part of a network and end at the servers of a datacenter. Our future goal is to realize such Metron deployments with zero (or minimal) processing redundancy at the speed of the slowest link in the network.

*The blackbox NF might perform its own traffic classification and modification operations, which results in additional packet processing redundancy.

Other Future Plans

Although Metron dealt with the performance challenges of the emerging 100 GbE era, we already foresee new challenges stemming from multi-100 GbE connections. This is not a far fetched scenario as existing P4 Tofino switches support multiple 100 GbE ports and an aggregate bandwidth of up to 6.5 Tb/s [145]. In this context, networked systems with commodity servers will face new performance issues, leading to several interesting questions:

Future Question 1: Is it possible to co-design programmable DMA controllers and (Smart) NICs to explicitly control a server's memory region to which packets are transferred?

Future Question 2: How can we exploit end hosts to facilitate *true end-to-end NFV packet processing* in the presence of both programmable and legacy equipment?

Future Question 3: Is it possible to increase modern commodity NICs' programmability without increasing their cost (e.g., by extending their firmware)?

Future Question 4: Is it possible to realize the concepts proposed by this thesis with increased security, without compromising performance?

Chapter 11

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, sustainability, ethical & security issues regarding this thesis are discussed in §11.1 and §11.2 respectively.

11.1 Sustainability

On a daily basis, people make 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. The Brundtland report, from the United Nations World Commission on Environment and Development [228], defines 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. Academic research ought to contribute to a sustainable global ecosystem, hence contributing to each of these components. Therefore, academic research must solve research problems with solutions that are both (*i*) effective and (*ii*) sustainable.

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's 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 demonstrated this in §6.5. Consequently, the more data SCC fits into a CPU core's cache(s), the fewer transfers are required with main memory, hence the fewer CPU cycles are spent by the system to process this data.

SNF's 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.

Metron's contribution to sustainability
The third contribution of this thesis (called Metron in Chapter 8) drastically increases the performance and efficiency of NFV systems by realizing service chains at the speed of the hardware, while using only a small fraction of servers' CPU cores. The contributions of Metron are summarized in Figures 1.6 and 1.7, while more thorough results are provided in §8.2. Overall, Metron is an even more drastic service chain consolidation scheme than SNF.

Using the results of this thesis in practice
To understand the implications of using SCC, SNF, and Metron 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, SNF, or Metron. Processing power in computer systems is highly-correlated with energy consumption, hence all of our contributions have a direct impact on reducing the energy consumption of NFV systems. This means that SCC, SNF, and Metron contribute to reducing the power consumption of datacenters, contributing to environmental benefits.

With regard to societal aspects of sustainability, throughput and latency are two important aspects of quality in communication systems. We demonstrated that SCC, SNF, and Metron reduce the end-to-end latency and that SNF and Metron 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 satisfied customers 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. Savings 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. Metron utilizes the least possible amount of server resources by exploiting other networking hardware. Therefore, all of the contributions of this thesis can save resources, hence bring increased profits to NFV stakeholders.

Finally, although all of the contributions of this thesis approach the same problem, they employ orthogonal solutions. This means that an NFV stakeholder might benefit from applying SCC, SNF, and Metron at the same time. However, comparing these contributions, from a sustainability point of view, we conclude that Metron and SNF are more sustainable solutions than SCC. Here is our reasoning. As the demands for ultra low latency services continue to increase [175, 176, 177], it is more and more necessary to utilize highly-optimized NFV solutions that consolidate traffic processing. This need served as an inspiration for both SNF and Metron, as stated in §4.3. 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 and Metron are 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.

11.2 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 SCC, SNF, and Metron. All these systems 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, SNF, or Metron 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. In §10.2 we sketched a plan for future work which will address this security issue.

Chapter 12

Conclusions

This thesis has introduced substantial novelty in the area of programmable networked systems, by realizing NFV service chains at the challenging speeds of 40 Gbps and 100 Gbps networks. The novelty of this thesis can be decomposed into four unique scientific contributions as follows.

The first contribution of this thesis is a tool that telecommunications' stakeholders can use to quickly and reliably identify performance bottlenecks in NFV service chains. This tool uncovered I/O and scheduling issues that greatly degrade the performance of service chains using unmodified network drivers.

The second contribution of this thesis is an NFV platform that leverages I/O and scheduling techniques to address the issues uncovered by the profiler, thus achieving (*i*) 3x lower end-to-end latency and (*ii*) 2x (up to 40x for certain percentiles) lower latency variance compared to a baseline solution. This performance is achieved by reducing both cache misses and scheduling overheads.

The third contribution of this thesis further improves the performance of service chains by introducing a packet processing synthesis framework. This framework 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. Synthesized long and stateful service chains were shown to operate at line-rate 40 Gbps. Using an OpenFlow switch as a classifier, three synthesized ISP-level service chains were realized at 40 Gbps.

The fourth contribution of this thesis is an NFV platform that eliminates costly inter-core communication between the NFs of a service chain by delegating packet processing and CPU core dispatching operations to programmable hardware devices. Combining this powerful property with service chain synthesis offers dramatic hardware efficiency and performance increases over the state of the art. With commodity hardware assistance, this platform fully exploits the processing capacity of a single server, to deeply inspect traffic at 40 Gbps and execute stateful service chains at the speed of a 100 GbE NIC.

Bibliography

[1] G. Anagnostopoulos, *A Companion to Aristotle*, ser. Blackwell Companions to Philosophy. Wiley, 2009, ISBN 9781444305678. [Online]. Available: https://books.google.se/books?id=_BhqbWOnnK4C

[2] 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

[3] F. Prampolini, "Telco 2015: five telling years, four future scenarios," 2010. [Online]. Available: http://www-05.ibm.com/cz/gbs/study/pdf/GBE03259USEN.PDF

[4] 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

[5] 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/

[6] Security Essen Press, "Figures, Data and Facts About the Security Industry: Further Strong Growth in the International Security Market," 2018. [Online]. Available: https://www.security-essen.de/press/press-texts/detail-sec/figures-data-and-facts-about-the-security-industry-further-strong-growth-in-the-international-security-market-2536

[7] 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

[8] 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

[9] 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

[10] 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

[11] 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

[12] 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

[13] 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

[14] 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

[15] 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

[16] 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

[17] 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

[18] 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

[19] D. Firestone, "VFP: A Virtual Switch Platform for Host SDN in the Public Cloud," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and*

Implementation, ser. NSDI'17. Berkeley, CA, USA: USENIX Association, 2017, pp. 315–328. [Online]. Available: http://dl.acm.org/citation.cfm?id=3154630.3154656

[20] D. Meyer, "RCR Wireless News: Many telecom operators planning NFV deployments this year," 2015, Accessed: September 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: May 31, 2018. [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] Gilad Shainer, 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

[25] J. F. Kim, "Mellanox Blog: 25 Is the New 10, 50 Is the new 40, 100 Is the New Amazing," Mar. 2016. [Online]. Available: http://www.mellanox.com/blog/2016/03/25-is-the-new-10-50-is-the-new-40-100-is-the-new-amazing/

[26] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A Case for Intelligent RAM," *IEEE Micro*, vol. 17, no. 2, pp. 34–44, Mar. 1997. [Online]. Available: https://doi.org/10.1109/40.592312

[27] 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: https://doi.org/10.1016/j.jss.2017.01.005

[28] DPDK, "Data Plane Development Kit (DPDK)," Accessed: May 31, 2018. [Online]. Available: http://dpdk.org

[29] 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

[30] G. P. Katsikas, T. Barbette, D. Kostić, R. Steinert, and G. Q. Maguire Jr., "Metron: NFV Service Chains at the True Speed of the Underlying Hardware," in *15th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'18. Renton, WA: USENIX Association, 2018, pp. 171–186. [Online]. Available: https://www.usenix.org/system/files/conference/nsdi18/nsdi18-katsikas.pdf

[31] 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

[32] 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

[33] 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

[34] 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

[35] J. v. Neumann, "First Draft of a Report on the EDVAC," Tech. Rep., 1945.

[36] 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

[37] Intel, "Data Direct I/O Technology," 2012, Accessed: May 31, 2018. [Online]. Available: http://www.intel.co.jp/content/dam/www/public/us/en/documents/white-papers/data-direct-i-o-technology-overview-paper.pdf

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

[39] 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

[40] 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

[41] 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

[42] 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

[43] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking Control of the Enterprise," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 4, pp. 1–12, Aug. 2007. [Online]. Available: http://doi.acm.org/10.1145/1282427.1282382

[44] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: Towards an Operating System for Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, pp. 105–110, Jul. 2008. [Online]. Available: http://doi.acm.org/10.1145/1384609.1384625

[45] "POX SDN Controller," Accessed: May 31, 2018. [Online]. Available: https://openflow.stanford.edu/display/ONL/POX+Wiki

[46] "Ryu SDN Controller," Accessed: May 31, 2018. [Online]. Available: https://osrg.github.io/ryu/

[47] L. Foundation, "OpenDaylight: Open Source SDN Platform," Accessed: May 31, 2018. [Online]. Available: https://www.opendaylight.org/

[48] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar, "ONOS: Towards an Open, Distributed SDN OS," in *Proceedings of the 3rd Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14. New York, NY, USA: ACM, 2014, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/2620728.2620744

[49] ON.Lab, "Open Network Operating System (ONOS)," 2018, Accessed: May 31, 2018. [Online]. Available: http://onosproject.org/

[50] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A Network Programming Language," in *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '11. New York, NY, USA: ACM, 2011, pp. 279–291. [Online]. Available: http://doi.acm.org/10.1145/2034773.2034812

[51] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing Software-defined Networks," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 1–14. [Online]. Available: http://dl.acm.org/citation.cfm?id=2482626.2482629

[52] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak, "Maple: Simplifying SDN Programming Using Algorithmic Policies," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 87–98. [Online]. Available: http://doi.acm.org/10.1145/2486001.2486030

[53] 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

[54] 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

[55] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the Data Plane with Anteater," in *Proceedings of the ACM SIGCOMM 2011 Conference*, ser. SIGCOMM '11. New York, NY, USA: ACM, 2011, pp. 290–301. [Online]. Available: http://doi.acm.org/10.1145/2018436.2018470

[56] 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

[57] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real Time Network Policy Checking Using Header Space Analysis," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 99–112. [Online]. Available: http://dl.acm.org/citation.cfm?id=2482626.2482638

[58] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying Network-wide Invariants in Real Time," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 15–28. [Online]. Available: http://dl.acm.org/citation.cfm?id=2482626.2482630

[59] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat, "Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 87–99. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616448.2616457

[60] A. El-Hassany, J. Miserez, P. Bielik, L. Vanbever, and M. Vechev, "SDNRacer: Concurrency Analysis for Software-defined Networks," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. New York, NY, USA: ACM, 2016, pp. 402–415. [Online]. Available: http://doi.acm.org/10.1145/2908080.2908124

[61] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "OFLOPS: An Open Framework for Openflow Switch Evaluation," in *Proceedings of the 13th International Conference on Passive and Active Measurement*, ser. PAM'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 85–95. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-28537-0_9

[62] M. Kuźniar, P. Perešíni, 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

[63] 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

[64] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756

[65] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel Symbolic Execution for Automated Real-world Software Testing," in *Proceedings of the 6th Conference on Computer Systems*, ser. EuroSys '11. New York, NY, USA: ACM, 2011, pp. 183–198. [Online]. Available: http://doi.acm.org/10.1145/1966445.1966463

[66] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "Automatic Test Packet Generation," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '12. New York, NY, USA: ACM, 2012, pp. 241–252. [Online]. Available: http://doi.acm.org/10.1145/2413176.2413205

[67] K. Agarwal, E. Rozner, C. Dixon, and J. Carter, "SDN Traceroute: Tracing SDN Forwarding Without Changing Network Behavior," in *Proceedings of the 3rd Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14. New York, NY, USA: ACM, 2014, pp. 145–150. [Online]. Available: http://doi.acm.org/10.1145/2620728.2620756

[68] P. Perešíni, M. Kuźniar, and D. Kostić, "Monocle: Dynamic, Fine-grained Data Plane Monitoring," in *Proceedings of the 11th International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '15. New York, NY, USA: ACM, 2015, pp. 32:1–32:13. [Online]. Available: http://doi.acm.org/10.1145/2716281.2836117

[69] K. Bu, X. Wen, B. Yang, Y. Chen, L. E. Li, and X. Chen, "Is every flow on the right track?: Inspect SDN forwarding with RuleScope," in *35th Annual IEEE International Conference on Computer Communications, INFOCOM 2016, San Francisco, CA, USA, April 10-14, 2016*, 2016, pp. 1–9. [Online]. Available: http://dx.doi.org/10.1109/INFOCOM.2016.7524333

[70] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann, "OFRewind: Enabling Record and Replay Troubleshooting for Networks," in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 29–29. [Online]. Available: http://dl.acm.org/citation.cfm?id=2002181.2002210

[71] M. Canini, V. Jovanović, D. Venzano, B. Spasojević, O. Crameri, and D. Kostić, "Toward Online Testing of Federated and Heterogeneous Distributed Systems," in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 20–20. [Online]. Available: http://dl.acm.org/citation.cfm?id=2002181.2002201

[72] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown, "Where is the Debugger for My Software-defined Network?" in *Proceedings of the 1st Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN '12. New York, NY, USA: ACM, 2012, pp. 55–60. [Online]. Available: http://doi.acm.org/10.1145/2342441.2342453

[73] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for Network Update," 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. 323–334. [Online]. Available: http://doi.acm.org/10.1145/2342356.2342427

[74] N. P. Katta, J. Rexford, and D. Walker, "Incremental Consistent Updates," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13. New York, NY, USA: ACM, 2013, pp. 49–54. [Online]. Available: http://doi.acm.org/10.1145/2491185.2491191

[75] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz, "zUpdate: Updating Data Center Networks with Zero Loss," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 411–422. [Online]. Available: http://doi.acm.org/10.1145/2486001.2486005

[76] P. Perešíni, M. Kuźniar, M. Canini, and D. Kostić, "ESPRES: Transparent SDN Update Scheduling," in *Proceedings of the 3rd Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14. New York, NY, USA: ACM, 2014, pp. 73–78. [Online]. Available: http://doi.acm.org/10.1145/2620728.2620747

[77] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic Scheduling of Network Updates," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, pp. 539–550. [Online]. Available: http://doi.acm.org/10.1145/2619239.2626307

[78] M. Kuźniar, P. Perešíni, and D. Kostić, "Providing Reliable FIB Update Acknowledgments in SDN," in *Proceedings of the 10th International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '14. New York, NY, USA: ACM, 2014, pp. 415–422. [Online]. Available: http://doi.acm.org/10.1145/2674005.2675006

[79] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, "Onix: A Distributed Control Platform for Large-scale Production Networks," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 351–364. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924968

[80] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling Flow Management for High-performance Networks," in *Proceedings of the ACM SIGCOMM 2011 Conference*, ser. SIGCOMM '11. New York, NY, USA: ACM, 2011, pp. 254–265. [Online]. Available: http://doi.acm.org/10.1145/2018436.2018466

[81] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Can the Production Network Be the Testbed?" in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 365–378. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924969

[82] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN," in *Proceedings of the ACM SIGCOMM 2013 Conference on*, ser.

SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 99–110. [Online]. Available: http://doi.acm.org/10.1145/2486001.2486011

[83] 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

[84] European Union Horizon 2020 project, "BEhavioural BAsed forwarding (BEBA)," 2015-2017, Accessed: September 15, 2017. [Online]. Available: http://www.beba-project.eu/

[85] 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

[86] 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

[87] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling Packet Programs to Reconfigurable Switches," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 103–115. [Online]. Available: http://dl.acm.org/citation.cfm?id=2789770.2789778

[88] A. Sivaraman, C. Kim, R. Krishnamoorthy, A. Dixit, and M. Budiu, "DC.P4: Programming the Forwarding Plane of a Data-center Switch," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR '15. New York, NY, USA: ACM, 2015, pp. 2:1–2:8. [Online]. Available: http://doi.acm.org/10.1145/2774993.2775007

[89] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "HULA: Scalable Load Balancing Using Programmable Data Planes," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '16. New York, NY, USA: ACM, 2016, pp. 10:1–10:12. [Online]. Available: http://doi.acm.org/10.1145/2890955.2890968

[90] S. Abdi, U. Aftab, G. Bailey, B. Boughzala, F. Dewal, S. Parsazad, and E. Tremblay, "PFPSim: A Programmable Forwarding Plane Simulator," in *Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '16. New York, NY, USA: ACM, 2016, pp. 55–60. [Online]. Available: http://doi.acm.org/10.1145/2881025.2881029

[91] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford, "PISCES: A Programmable, Protocol-Independent Software Switch," in *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 525–538. [Online]. Available: http://doi.acm.org/10.1145/2934872.2934886

[92] L. Molnár, G. Pongrácz, G. Enyedi, Z. L. Kis, L. Csikor, F. Juhász, A. Kőrösi, and G. Rétvári, "Dataplane Specialization for High-performance OpenFlow Software Switching," in *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*,

ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 539–552. [Online]. Available: http://doi.acm.org/10.1145/2934872.2934887

[93] 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

[94] 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

[95] 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

[96] DPDK, "Packet Framework," Accessed: May 31, 2018. [Online]. Available: http://dpdk.org/doc/guides/prog_guide/packet_framework.html

[97] Open vSwitch, "An Open Virtual Switch," Accessed: May 31, 2018. [Online]. Available: http://openvswitch.org

[98] 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

[99] 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

[100] 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

[101] 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

[102] Intel, "Receive-Side Scaling (RSS)," 2007, Accessed: May 31, 2018. [Online]. Available: http://www.intel.com/content/dam/support/us/en/documents/network/sb/318483001us2.pdf

[103] C. B. Robison, "How to Set Up Intel Ethernet Flow Director," Jun. 2017. [Online]. Available: https://software.intel.com/en-us/articles/setting-up-intel-ethernet-flow-director

[104] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy, "High Performance Packet Processing with FlexNIC," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 67–81. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872367

[105] Netronome, "Agilio CX SmartNICs," 2018. [Online]. Available: https://www.netronome.com/products/agilio-cx/

[106] ——, "Agilio LX 1x100GbE SmartNIC," 2017. [Online]. Available: https://www.netronome.com/media/documents/PB_Agilio_Lx_1x100GbE.pdf

[107] Netcope Technologies, "Netcope P4 (NP4): Cloud based platform for development of FPGA firmware in P4 language," 2018. [Online]. Available: https://www.netcope.com/getattachment/058b62e4-54d3-420a-927c-ac28e2e1677b/Netcope-P4.aspx

[108] R. Renwick, "Increase Application Performance with SmartNICs," 2017. [Online]. Available: https://www.openstack.org/assets/presentation-media/Netronome-OpenStack-Summit-Marketplace-presentation.pdf

[109] D. Zacks and P. Jones, "Cisco Enterprise Silicon: Delivering Innovation for Advanced Routing and Switching," Jun. 2017, Accessed: May 31, 2018. [Online]. Available: http://clnv.s3.amazonaws.com/2017/usa/pdf/BRKARC-3467.pdf

[110] 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

[111] 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

[112] 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

[113] Xen, "The Xen Project," Accessed: May 31, 2018. [Online]. Available: http://www.xenproject.org/

[114] KVM, "Kernel-based Virtual Machine (KVM)," Accessed: May 31, 2018. [Online]. Available: http://www.linux-kvm.org/page/Main_Page

[115] 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

[116] Docker, "Docker Containers," Accessed: May 31, 2018. [Online]. Available: https://www.docker.com/

[117] W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, and T. Wood, "Flurries: Countless Fine-Grained NFs for Flexible Per-Flow Customization," in *Proceedings of the 12th ACM International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '16. New York, NY, USA: ACM, 2016, pp. 3–17. [Online]. Available: http://doi.acm.org/10.1145/2999572.2999602

[118] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu, "NFP: Enabling Network Function Parallelism in NFV," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: ACM, 2017, pp. 43–56. [Online]. Available: http://doi.acm.org/10.1145/3098822.3098826

[119] P. Zave, R. A. Ferreira, X. K. Zou, M. Morimoto, and J. Rexford, "Dynamic Service Chaining with Dysco," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: ACM, 2017, pp. 57–70. [Online]. Available: http://doi.acm.org/10.1145/3098822.3098827

[120] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "NetBricks: Taking the V out of NFV," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 203–216. [Online]. Available: http://dl.acm.org/citation.cfm?id=3026877.3026894

[121] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

[122] 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

[123] 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

[124] 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 10th 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

[125] T. Barbette, C. Soldani, and L. Mathy, "Fast Userspace Packet Processing," in *Proceedings of the 11th 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

[126] DPDK, "Network Interface Controller Drivers' Features," Accessed: May 31, 2018. [Online]. Available: http://dpdk.org/doc/guides/nics/features.html

[127] D. Raumer, S. Gallenmüller, P. Emmerich, L. Märdian, F. Wohlfart, and G. Carle, "Efficient serving of VPN endpoints on COTS server hardware," in *2016 IEEE 5th International Conference on Cloud Networking (CloudNet'16)*, Pisa, Italy, Oct. 2016.

[128] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A Holistic Approach to Fast In-memory Key-value Storage," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 429–444. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616448.2616488

[129] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman, "Be Fast, Cheap and in Control with SwitchKV," in *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 31–44. [Online]. Available: http://dl.acm.org/citation.cfm?id=2930611.2930614

[130] M. A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park, "Kargus: A Highly-scalable Software-based Intrusion Detection System," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12, 2012. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382232

[131] Y. Go, M. A. Jamshed, Y. Moon, C. Hwang, and K. Park, "APUNet: Revitalizing GPU as Packet Processing Accelerator," in *14th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'17. USENIX Association, 2017, pp. 83–96. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/go

[132] 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

[133] 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

[134] 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

[135] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella, "Paving the Way for NFV: Simplifying Middlebox Modifications Using StateAlyzr," in *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'16. USENIX Association, 2016, pp. 239–253. [Online]. Available: http://dl.acm.org/citation.cfm?id=2930611.2930628

[136] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless Network Functions: Breaking the Tight Coupling of State and Processing," in *14th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'17, 2017, pp. 97–112. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/kablan

[137] 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

[138] 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

[139] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat, "xOMB: Extensible Open Middleboxes with Commodity Servers," in *Proceedings of the 8th 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

[140] 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

[141] Apache, "Hadoop," Accessed: May 31, 2018. [Online]. Available: http://hadoop.apache.org/

[142] T. Barbette, C. Soldani, R. Gaillard, and L. Mathy, "A low-level dive into building a high-speed NFV dataplane for service chaining," 2018. [Online]. Available: http://hdl.handle.net/2268/223111

[143] ——, "Building a chain of high-speed VNFs in no time," in *Proceedings of the IEEE International Conference on High Performance Switching and Routing*, ser. HPSR'18, 2018.

[144] T. Barbette, "Architecture for programmable network infrastructure," Doctoral thesis, University of Liege, Faculty of Applied Sciences, Department of Electricity, Electronics and Informatics, Liege, Belgium, Jul. 2018. [Online]. Available: http://hdl.handle.net/2268/226257

[145] Barefoot, "Tofino World's fastest P4-programmable Ethernet switch ASICs," 2018, Accessed: May 31, 2018. [Online]. Available: https://barefootnetworks.com/products/brief-tofino/

[146] European Telecommunications Standards Institute, "Network Functions Virtualisation," 2017. [Online]. Available: http://www.etsi.org/technologies-clusters/technologies/689-network-functions-virtualisation

[147] European Telecommunications Standards Institute (ETSI), "Open Source NFV Management and Orchestration (MANO) ," 2017. [Online]. Available: https://osm.etsi.org/

[148] OpenStack, "Open Source Cloud Computing Software," 2018, Accessed: May 31, 2018. [Online]. Available: https://www.openstack.org/

[149] ON.Lab, "Central Office Re-architected as a Datacenter (CORD)," 2018, Accessed: May 31, 2018. [Online]. Available: http://opencord.org/

[150] The Linux Foundation, "Open Platform for NFV (OPNFV)," 2018, Accessed: May 31, 2018. [Online]. Available: https://www.opnfv.org/

[151] 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

[152] 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

[153] 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

[154] S. G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. K. Ramakrishnan, T. Wood, M. Arumaithurai, and X. Fu, "NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: ACM, 2017, pp. 71–84. [Online]. Available: http://doi.acm.org/10.1145/3098822.3098828

[155] G. P. Katsikas, "Realizing High Performance NFV Service Chains," Licentiate thesis, KTH Royal Institute of Technology, School of Information and Communication Technology, Kista, Sweden, Nov. 2016, TRITA-ICT 2016:35. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-195352

[156] 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

[157] V. Viswanathan, "Intel Memory Latency Checker (MLC) v3.1a," 2013, Updated: March 9, 2018. [Online]. Available: https://software.intel.com/en-us/articles/intelr-memory-latency-checker

[158] J. Levon, "OProfile: A System Profiler for Linux," July 2018, Accessed: May 31, 2018. [Online]. Available: http://oprofile.sourceforge.net/news/

[159] Perf, "Linux profiling with performance counters," 2018, Accessed: May 31, 2018. [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

[160] 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

[161] J. Weidendorfer, "KCachegrind," 2018, Accessed: May 31, 2018. [Online]. Available: https://kcachegrind.github.io/html/Home.html

[162] Intel, "Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors," 2009, Accessed: May 31, 2018. [Online]. Available: https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf

[163] likwid, "Performance monitoring and benchmarking suite," 2018, Accessed: May 31, 2018. [Online]. Available: https://github.com/RRZE-HPC/likwid

[164] 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

[165] Cisco, "Migrate to a 40-Gbps Data Center with Cisco QSFP BiDi Technology," 2013. [Online]. Available: http://www.cisco.com/c/en/us/products/collateral/switches/nexus-9000-series-switches/white-paper-c11-729493.html

[166] A. Viejo, "QLogic and Broadcom First to Demonstrate End-to-End Interoperability for 25Gb and 100Gb Ethernet," 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

[167] Firestone, Daniel, "Hardware-Accelerated Networks at Scale in the Cloud," Aug. 2017, Accessed: May 31, 2018. [Online]. Available: https://conferences.sigcomm.org/sigcomm/2017/files/program-kbnets/keynote-2.pdf

[168] Schmidtke, Katharine, "Facebook: Designing 100G optical connections," Mar. 2017, Accessed: May 31, 2018. [Online]. Available: https://code.facebook.com/posts/1633153936991442/designing-100g-optical-connections/

[169] A. Eckert, L. MartinGarcia, R. Niazmand, and X. Wang, "Wedge 100: More open and versatile than ever," Oct. 2016, Accessed: May 31, 2018. [Online]. Available: https://code.facebook.com/posts/1802489260027439/wedge-100-more-open-and-versatile-than-ever/

[170] Intel, "Xeon E5 2667 v3 processor," Accessed: May 31, 2018. [Online]. Available: http://ark.intel.com/products/83361/Intel-Xeon-Processor-E5-2667-v3-20M-Cache-3_20-GHz

[171] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "SoftNIC: A Software NIC to Augment Hardware," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-155, May 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html

[172] T. Barbette, "Repository with DPDK extensions for OpenBox," 2018. [Online]. Available: https://github.com/tbarbette/fastclick/tree/openbox

[173] Amazon, "Compute and Networking Services for Amazon Web Services," 2018, Accessed: May 31, 2018. [Online]. Available: http://docs.aws.amazon.com/gettingstarted/latest/awsgsg-intro/gsg-aws-compute-network.html

[174] ——, "EC2 Container Service (ECS)," 2018, Accessed: May 31, 2018. [Online]. Available: https://aws.amazon.com/ecs/

[175] H. Benn, "Vision and Key Features for 5th Generation (5G) Cellular," Jan. 2014. [Online]. Available: http://www.academia.edu/21567615/Vision_and_Key_Features_for_5_th_Generation_5G_Cellular

[176] Nokia, "5G use cases and requirements," Jul. 2014, white paper with product code C401-01016-WP-201407-1-EN. [Online]. Available: https://www.ramonmillan.com/documentos/bibliografia/5GUseCases_Nokia.pdf

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

[178] 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.

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

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

[181] Hewlett Packard, "HPE FlexNetwork 5130 EI Switch Series," Jan. 2017. [Online]. Available: https://h50146.www5.hpe.com/products/networking/datasheet/HP_5130EI_Switch_Series_J.pdf

[182] 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

[183] The Click Modular Router, "Academic and industrial contributors," 2018. [Online]. Available: https://github.com/kohler/click/blob/master/AUTHORS

[184] Amazon, "Enhanced Networking with SR-IOV," 2018, Accessed: May 31, 2018. [Online]. Available: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/enhanced-networking.html

[185] G. P. Katsikas, "FastClick user-space I/O multiplexing using standard Linux network drivers." 2016, Accessed: May 31, 2018. [Online]. Available: https://github.com/gkatsikas/fastclick/tree/mmap

[186] ——, "System benchmarks," 2016, Accessed: May 31, 2018. [Online]. Available: https://github.com/gkatsikas/system-bench

[187] 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

[188] Intel, "Hardware events for Intel Haswell processors," 2018, Accessed: May 31, 2018. [Online]. Available: https://download.01.org/perfmon/index/haswell.html

[189] ——, "High Precision Event Timers (HPET)," 2004, Accessed: May 31, 2018. [Online]. Available: http://www.intel.se/content/dam/www/public/us/en/documents/technical-specifications/software-developers-hpet-spec-1-0a.pdf

[190] 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.

[191] L. Deri, "Direct NIC Access with PF_RING," 2011, Accessed: May 31, 2018. [Online]. Available: http://www.ntop.org/pf_ring/

[192] 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:nbn:se:kth:diva-180397

[193] 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

[194] 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

[195] 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

[196] W. Sun and R. Ricci, "Fast and Flexible: Parallel Packet Processing with GPUs and Click," in *Proceedings of the 9th 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

[197] G. P. Katsikas, "SNF extensions of FastClick's stateful flow processing elements." 2016, Accessed: May 31, 2018. [Online]. Available: https://github.com/gkatsikas/fastclick/tree/snf

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

[199] 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/

[200] 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

[201] 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

[202] 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

[203] G. P. Katsikas, "An implementation of the SNF framework," 2016, Accessed: May 31, 2018. [Online]. Available: https://github.com/gkatsikas/snf

[204] ——, "Metron controller's southbound driver for managing commodity servers," 2018. [Online]. Available: https://github.com/gkatsikas/onos/tree/metron-driver

[205] T. Dietz, R. Bifulco, F. Manco, J. Martins, H. Kolbe, and F. Huici, "Enhancing the BRAS through virtualization," in *Proceedings of the 1st IEEE Conference on Network Softwarization, NetSoft 2015*, 2015, pp. 1–5. [Online]. Available: https://doi.org/10.1109/NETSOFT.2015.7116144

[206] R. Krishnan, M. Durrani, and P. Phaal, "Real-time SDN and NFV Analytics for DDoS Mitigation," 2014. [Online]. Available: https://blog.sflow.com/2014/02/nfd7-real-time-sdn-and-nfv-analytics_1986.html

[207] M. Yu, Y. Yi, J. Rexford, and M. Chiang, "Rethinking Virtual Network Embedding: Substrate Support for Path Splitting and Migration," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 17–29, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1355734.1355737

[208] J. He, R. Zhang-Shen, Y. Li, C.-Y. Lee, J. Rexford, and M. Chiang, "DaVinci: Dynamically Adaptive Virtual Networks for a Customized Internet," in *Proceedings of the 2008 ACM CoNEXT Conference*, ser. CoNEXT '08. New York, NY, USA: ACM, 2008, pp. 15:1–15:12. [Online]. Available: http://doi.acm.org/10.1145/1544012.1544027

[209] M. Chowdhury, M. R. Rahman, and R. Boutaba, "ViNEYard: Virtual Network Embedding Algorithms with Coordinated Node and Link Mapping," *IEEE/ACM Trans. Netw.*, vol. 20, no. 1, pp. 206–219, Feb. 2012. [Online]. Available: http://dx.doi.org/10.1109/TNET.2011.2159308

[210] M. Mitzenmacher, "The Power of Two Choices in Randomized Load Balancing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 10, pp. 1094–1104, Oct. 2001. [Online]. Available: http://dx.doi.org/10.1109/71.963420

[211] M. Al-Fares, A. Loukissas, and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture," in *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, ser. SIGCOMM '08. New York, NY, USA: ACM, 2008, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/1402958.1402967

[212] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic Flow Scheduling for Data Center Networks," in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 19–19. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855711.1855730

[213] E. C. Rosen, A. Viswanathan, and R. Callon, "Multiprotocol Label Switching Architecture," Internet Request for Comments (RFC) 3031, RFC Editor, Fremont, CA, USA, pp. 1–61, Jan. 2001, updated by RFCs 6178, 6790. [Online]. Available: https://www.rfc-editor.org/rfc/rfc3031.txt

[214] V. A. Olteanu and C. Raiciu, "Efficiently Migrating Stateful Middleboxes," in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '12. ACM, 2012, pp. 93–94. [Online]. Available: http://doi.acm.org/10.1145/2342356.2342376

[215] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 305–320. [Online]. Available: http://dl.acm.org/citation.cfm?id=2643634.2643666

[216] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, "Network Configuration Protocol (NETCONF)," Internet Request for Comments (RFC) 6241 (Proposed Standard), RFC Editor, Fremont, CA, USA, pp. 1–113, Jun. 2011, updated by RFC 7803. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6241.txt

[217] M. Bjorklund, "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)," Internet Request for Comments (RFC) 6020 (Proposed Standard), RFC Editor, Fremont, CA, USA, pp. 1–173, Oct. 2010. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6020.txt

[218] J. Case, M. Fedor, M. L. Schoffstall, and J. Davin, "Simple Network Management Protocol (SNMP)," Internet Request for Comments (RFC) 1157 (Historic), Internet Engineering Task Force, May 1990. [Online]. Available: http://www.ietf.org/rfc/rfc1157.txt

[219] T. Barbette and G. P. Katsikas, "Metron data plane," 2018, https://github.com/tbarbette/fastclick/tree/metron.

[220] NoviFlow, "NoviSwitch 1132 High Performance OpenFlow Switch," 2013. [Online]. Available: https://noviflow.com/wp-content/uploads/NoviSwitch-1132-Datasheet-V2_1.pdf

[221] Mellanox Technologies, "Mellanox NIC's Performance Report with DPDK 17.05," 2017, Document number MLNX-15-52365, Revision 1.0, 2017. [Online]. Available: https://fast.dpdk.org/doc/perf/DPDK_17_05_Mellanox_NIC_performance_report.pdf

[222] M. Kuźniar, P. Perešíni, and D. Kostić, "What You Need to Know About SDN Flow Tables," in *Passive and Active Measurement (PAM)*, ser. Lecture Notes in Computer Science, vol. 8995, 2015, pp. 347–359, https://doi.org/10.1007/978-3-319-15509-8_26. [Online]. Available: http://wan.poly.edu/pam2015/

[223] M. Kuźniar, P. Perešíni, D. Kostić, and M. Canini, "Methodology, Measurement and Analysis of Flow Table Update Characteristics in Hardware OpenFlow Switches," *Computer Networks: The International Journal of Computer and Telecommunications Networking, Elsevier,*, vol. vol. 26, 2018, https://doi.org/10.1016/j.comnet.2018.02.014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128618300811

[224] G. Bianchi, M. Bonola, S. Pontarelli, D. Sanvito, A. Capone, and C. Cascone, "Open Packet Processor: a programmable architecture for wire speed platform-independent stateful in-network processing," *arXiv preprint arXiv:1605.01977*, 2016.

[225] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen, "ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. ACM, 2016, pp. 1–14. [Online]. Available: http://doi.acm.org/10.1145/2934872.2934897

[226] T. Barbette, "FastClick's github repository," 2018. [Online]. Available: https://github.com/tbarbette/fastclick

[227] G. P. Katsikas and T. Barbette, "ONOS Server Device Driver Tutorial," 2018. [Online]. Available: https://wiki.onosproject.org/display/ONOS/Server+Device+Driver+Tutorial

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

[229] Intel, "64 and IA-32 Architectures Developer's Manual," 2016, Accessed: May 31, 2018. [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

[230] Linux, "Perf events' for Intel chipsets in Linux kernel sources," 2002, Accessed: May 31, 2018. [Online]. Available: https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/arch/x86/kernel/cpu/perf_event_intel.c?id=dea4f48a0a301b23c65af8e4fe8ccf360c272fbf

[231] D. Miller, "Linux ethtool," Accessed: May 31, 2018. [Online]. Available: https://linux.die.net/man/8/ethtool

[232] Linux, "Receive Offload," Accessed: May 31, 2018. [Online]. Available: https://lwn.net/Articles/358910/

Appendix A

Appendix

A.1 Collecting Performance Counters

This appendix discusses how Perf [159] 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 [229] 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" [229] provides very specific information that in conjunction with the processor's event types in the Linux kernel sources [230] 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 [229].

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 I/O to handle.
PERF_COUNT_SW _PAGE_FAULTS_MAJ	Number of major page faults. These did not require disk I/O 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 I/O 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 I/O 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 the licentiate part of this 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 §2.5. 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 Appendices A.2.2.4 and 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 good results in terms of throughput and latency. With the DPDK network driver, the respective number of Tx and Rx descriptors was 256. However, different types of applications might have different memory requirements, therefore the number of buffer descriptors indicated above might change accordingly.

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 [103] or RSS [102]. With Flow Director, we can issue *ethtool* [231] 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 using 1 Tx and 16 Rx queues provides high performance. In the case of the DPDK driver, using as many hardware Tx/Rx queues as the number of available CPU cores is usually a good practise.

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.

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 [232].
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 [103].
receive-hashing	Distributes frames to hardware queues by applying a hash function on a header space as per Intel's RSS [102].
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.

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.

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.

Suggested ixgbe configuration for packet processing applications
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