UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

MARCELO CAGGIANI LUIZELLI

Scalable Cost-Efficient Placement and Chaining of Virtual Network Functions

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Advisor: Prof. Dr. Luciano Paschoal Gaspary
Coadvisor: Prof. Dr. Luciana Salete Buriol

Porto Alegre
August 2017

CIP — CATALOGING-IN-PUBLICATION

Luizelli, Marcelo Caggiani

Scalable Cost-Efficient Placement and Chaining of Virtual Network Functions / Marcelo Caggiani Luizelli. – Porto Alegre: PPGC da UFRGS, 2017.

157 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2017. Advisor: Luciano Paschoal Gaspary; Coadvisor: Luciana Salete Buriol.

1. Network Function Virtualization. 2. Service Chaining. 3. NFV Orchestration. 4. Combinatorial Optimization. 5. Mathematical Programming. 6. Math-heuristic. 7. Variable Neighborhood Search. 8. Operational Cost. 9. Open vSwitch. 10. Performance Evaluation. I. Gaspary, Luciano Paschoal. II. Buriol, Luciana Salete. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Rui Vicente Oppermann
Vice-Reitora: Profa. Jane Fraga Tutikian
Pró-Reitor de Pós-Graduação: Prof. Celso Giannetti Loureiro Chaves
Diretor do Instituto de Informática: Prof. Carla Maria Dal Sasso Freitas
Coordenador do PPGC: Prof. João Luiz Dihl Comba
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

“The formulation of a problem is often more essential than its solution, which may be merely a matter of mathematical or experimental skill. To raise new questions, new possibilities, to regard old problems from a new angle, requires creative imagination and marks real advance in science.”

— Albert Einstein

"Science is a way of life. Science is a perspective. Science is the process that takes us from confusion to understanding in a manner that’s precise, predictive and reliable – a transformation, for those lucky enough to experience it, that is empowering and emotional."

— Brian Greene

ACKNOWLEDGMENTS

First of all, I would like to thank God for giving me such an amazing opportunity in life.

I would like to thank my parents and brothers for the unconditional support. I am quite aware that while I was a student, our time hanging out has been really short and joyful moments very sporadic. If I am taking one more step ahead, this is mainly because you have always been supporting me. For that, I will always be thankful.

I also would like to especially thank Fernanda, let's say my "wife", for mainly the patience and companionship during all these years of study. Certainly, I would not have gotten this far without all your support and care. I Love you!

I would like to thank my advisors – Prof. Luciano Paschoal and Prof. Luciana Buriol – who have always been incredibly available when I most needed. Thanks a lot for the technical and philosophical discussion, as well as for that "little push" when the idle time comes about. Although this time is coming to an end, I will try to stick around.

I also really thankful to Prof. Danny Raz who hosted me incredibly well during my time abroad. My time in Israel would not have been the same without all the support gave by Jose Yallouz and his family. Thanks a lot to Itzik Ashkenazi (from the Technion) for the very good moments during the working time – it was a pleasure to work together. Further, I also would like to thank all the guys from NOKIA Bell Labs Israel whom I had the pleasure to work with – Yaniv Saar, Gil Einziger, Erez Waisbard and Shachar Beiser.

Last, my sincere gratitude goes as well to all members of the computer networking group at UFRGS.

ABSTRACT

Network Function Virtualization (NFV) is a novel concept that is reshaping the middlebox arena, shifting network functions (*e.g.* firewall, gateways, proxies) from specialized hardware appliances to software images running on commodity hardware. This concept has potential to make network function provision and operation more flexible and cost-effective, paramount in a world where deployed middleboxes may easily reach the order of hundreds. Despite recent research activity in the field, little has been done towards scalable and cost-efficient placement & chaining of virtual network functions (VNFs) – a key feature for the effective success of NFV. More specifically, existing strategies have neglected the chaining aspect of NFV (focusing on efficient placement only), failed to scale to hundreds of network functions and relied on unrealistic operational costs. In this thesis, we approach VNF placement and chaining as an optimization problem in the context of Inter- and Intra-datacenter. First, we formalize the Virtual Network Function Placement and Chaining (VNFPC) problem and propose an Integer Linear Programming (ILP) model to solve it. The goal is to minimize required resource allocation, while meeting network flow requirements and constraints. Then, we address scalability of VNFPC problem to solve large instances (*i.e.*, thousands of NFV nodes) by proposing a fix-and-optimize-based heuristic algorithm for tackling it. Our algorithm incorporates a Variable Neighborhood Search (VNS) meta-heuristic, for efficiently exploring the placement and chaining solution space. Further, we assess the performance limitations of typical NFV-based deployments and the incurred operational costs of commodity servers and propose an analytical model that accurately predict the operational costs for arbitrary service chain requirements. Then, we develop a general service chain intra-datacenter deployment mechanism (named OCM – Operational Cost Minimization) that considers both the actual performance of the service chains (e.g., CPU requirements) as well as the operational incurred cost. Our novel algorithm is based on an extension of the well-known reduction from weighted matching to min-cost flow problem. Finally, we tackle the problem of monitoring service chains in NFV-based environments. For that, we introduce the DNM (Distributed Network Monitoring) problem and propose an optimization model to solve it. DNM allows service chain segments to be independently monitored, which allows specialized network monitoring requirements to be met in a efficient and coordinated way. Results show that the proposed ILP model for the VNFPC problem leads to a reduction of up to 25% in end-to-end delays (in comparison to chainings observed in traditional infrastructures) and an acceptable resource over-provisioning limited to 4%. Also, we provide strong evidences that our fix-and-optimize based heuristic is able to find feasible, high-quality solutions efficiently, even in scenarios scaling to thousands of VNFs. Further, we provide in-depth insights on network performance metrics (such as throughput, CPU utilization and packet processing) and its current limitations while considering typical deployment strategies. Our OCM algorithm reduces significantly operational costs when compared to the de-facto standard

placement mechanisms used in Cloud systems. Last, our DNM model allows finer grained network monitoring with limited overheads. By coordinating the placement of monitoring sinks and the forwarding of network monitoring traffic, DNM can reduce the number of monitoring sinks and the network resource consumption (54% lower than a traditional method).

Keywords: Network Function Virtualization. Service Chaining. NFV Orchestration. Combinatorial Optimization. Mathematical Programming. Math-heuristic. Variable Neighborhood Search. Operational Cost. Open vSwitch. Performance Evaluation.

Posicionamento e Encadeamento Escalável e de Baixo Custo de Funções Virtualizadas de Rede

RESUMO

A Virtualização de Funções de Rede (NFV – Network Function Virtualization) é um novo conceito arquitetural que está remodelando a operação de funções de rede (*e.g.*, firewall, gateways e proxies). O conceito principal de NFV consiste em desacoplar a lógica de funções de rede dos dispositivos de hardware especializados e, desta forma, permite a execução de imagens de software sobre hardware de prateleira (COTS – Commercial Off-The-Shelf). NFV tem o potencial para tornar a operação das funções de rede mais flexíveis e econômicas, primordiais em ambientes onde o número de funções implantadas pode chegar facilmente à ordem de centenas. Apesar da intensa atividade de pesquisa na área, o problema de posicionar e encadear funções de rede virtuais (VNF – Virtual Network Functions) de maneira escalável e com baixo custo ainda apresenta uma série de limitações. Mais especificamente, as estratégias existentes na literatura negligenciam o aspecto de encadeamento de VNFs (*i.e.*, objetivam sobretudo o posicionamento), não escalam para o tamanho das infraestruturas NFV (*i.e.*, milhares de nós com capacidade de computação) e, por último, baseiam a qualidade das soluções obtidas em custos operacionais não representativos. Nesta tese, aborda-se o posicionamento e o encadeamento de funções de rede virtualizadas (VNFPC – Virtual Network Function Placement and Chaining) como um problema de otimização no contexto intra- e inter-datacenter. Primeiro, formaliza-se o problema VNFPC e propõe-se um modelo de Programação Linear Inteira (ILP) para resolvê-lo. O objetivo consiste em minimizar a alocação de recursos, ao mesmo tempo que atende aos requisitos e restrições de fluxo de rede. Segundo, aborda-se a escalabilidade do problema VNFPC para resolver grandes instâncias do problema (*i.e.*, milhares de nós NFV). Propõe-se um um algoritmo heurístico baseado em *fix-and-optimize* que incorpora a meta-heurística Variable Neighborhood Search (VNS) para explorar eficientemente o espaço de solução do problema VNFPC. Terceiro, avalia-se as limitações de desempenho e os custos operacionais de estratégias típicas de aprovisionamento ambientes reais de NFV. Com base nos resultados empíricos coletados, propõe-se um modelo analítico que estima com alta precisão os custos operacionais para requisitos de VNFs arbitrários. Quarto, desenvolve-se um mecanismo para a implantação de encadeamentos de VNFs no contexto intra-datacenter. O algoritmo proposto (OCM – Operational Cost Minimization) baseia-se em uma extensão da redução bem conhecida do problema de emparelhamento ponderado (*i.e.*, weighted perfect matching problem) para o problema de fluxo de custo mínimo (*i.e.*, min-cost flow problem) e considera o desempenho das VNFs (*e.g.*, requisitos de CPU), bem como os custos operacionais estimados. Os resultados alcaçados mostram que o modelo ILP proposto para o problema VNFPC reduz em até 25% nos atrasos fim-a-fim (em comparação com os encadeamentos observados nas infra-estruturas tradicionais) com um

excesso de provisionamento de recursos aceitável – limitado a 4%. Além disso, os resultados evidenciam que a heurística proposta (baseada em fix-and-optimize) é capaz de encontrar soluções factíveis de alta qualidade de forma eficiente, mesmo em cenários com milhares de VNFs. Além disso, provê-se um melhor entendimento sobre as métricas de desempenho de rede (*e.g.*, vazão, consumo de CPU e capacidade de processamento de pacotes) para as estratégias típicas de implantação de VNFs adotadas infraestruturas NFV. Por último, o algoritmo proposto no contexto intra-datacenter (*i.e.* OCM) reduz significativamente os custos operacionais quando comparado aos mecanismos de posicionamento típicos utilizados em ambientes NFV.
Palavras-chave: Funções Virtualizadas de Rede. Encadeamento de Serviços. Orquestração de NFV. Otimização Combinatória. Programação Matemática. Custo Operacional. Avaliação de Desempenho..

LIST OF ABBREVIATIONS AND ACRONYMS

BA-2	Albert-Barabsi Model
BPP	Bin Packing Problem
CAPEX	Capital Expenditures
DAG	Direct Acyclic Graph
DDoS	Distributed Denial of Service
DNM	Distributed Network Monitoring
DPDK	Data Plane Development Kit
DPI	Deep Packet Inspection
DUT	Design Under Test
ETSI	European Telecommunications Standards Institute
GRE	Generic Routing Encapsulation
GSO	Generic Segmentation Offload
HH	Heavy Hitter
HTTP	Hypertext Transfer Protocol
HV	Hypervisor
IETF	Internet Engineering Task Force
ILP	Integer Linear Programming
IMS	IP Multimedia Subsystem
ISP	Internet Service Provider
KVM	Kernel-based Virtual Machine
MANO	NFV Management and Orchestration
MPLS	Multiprotocol Label Switching
N-PoP	Network Point of Presence
NF	Network Function
NFV	Network Function Virtualization
NFVI	NFV Infrastructure
NFVO	NFV Orchestrator

NIC	Network Interface Card
NUMA	Non-Uniform Memory Access
NSH	Network Service Header
OCM	Operational Cost Minimization
OPEX	Operational Expenditures
OPNFV	Open Platform for NFV
OVS	Open vSwitch
OVS-DPDK	DPDK-enabled OVS
PCIe	Peripheral Component Interconnect Express
PMD	Poll Mode Driver
RAN	Radio Access Network
RSS	Receive Side Scaling
SDN	Software Defined Networking
SFC	Service Function Chaining
TLB	Translation Lookaside Buffer
TSO	TCP Segmentation Offload
VIM	Virtualized Infrastructure Manager
VNE	Virtual Network Embedding Problem
VNF	Virtualized Network Function
VNFM	VNF Manager
VNFPC	Virtual Network Function Placement and Chaining Problem
vNIC	Virtual NIC
VNS	Variable Neighborhood Search
VPN	Virtual Private Network
VXLAN	Virtual Extensible LAN

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

CONTENTS

1 INTRODUCTION

Network Functions (NF) play an essential role in today's networks, as they support a diverse set of functions ranging from security (*e.g.*, firewalling and intrusion detection) to performance (*e.g.*, caching and proxying) (MARTINS et al., 2014). As currently implemented, middleboxes are difficult to deploy and maintain. This is mainly because cumbersome procedures need to be followed, such as dealing with a variety of custom-made hardware interfaces and manually chaining middleboxes to ensure the desired network behavior. Further, studies show that the number of middleboxes in enterprise networks (as well as in datacenter and ISP networks) is similar to the number of forwarding devices (BENSON; AKELLA; SHAIKH, 2011; SEKAR et al., 2012). Thus, the aforementioned difficulties are exacerbated by the complexity imposed by the high number of network functions that a network provider has to cope with, leading to high operational expenditures. Moreover, in addition to costs related to manually deploying and chaining middleboxes, the need for frequent hardware upgrades adds up to substantial capital investments.

Network Function Virtualization (NFV) has been proposed to shift middlebox processing from specialized hardware appliances to software running on commoditized hardware (GROUP, 2012). In addition to potentially reducing acquisition and maintenance costs, NFV is expected to allow network providers to make the most of the benefits of virtualization on the management of network functions (*e.g.*, elasticity, performance, flexibility, etc.). In this context, Software-Defined Networking (SDN) can be considered a convenient complementary technology, which, if available, has the potential to make the chaining of the aforementioned network functions much easier. In fact, it is not unreasonable to state that SDN has the potential to revamp the Service Function Chaining (SFC) problem.

In short, the problem consists of making sure network flows go efficiently through end-to-end paths traversing sets of network functions (MECHTRI et al., 2017). In the NFV/SDN realm and considering the flexibility offered by this environment, the problem consists of (sub)optimally defining how many instances of virtual network functions (VNF) are necessary and where to place them in the infrastructure. Furthermore, the problem encompasses the determination of end-to-end paths over which known network flows have to be transmitted so as to pass through the required placed network functions.

1.1 Problem Statement

There have been significant achievements in NFV, addressing aspects from effective planning and deployment (LEWIN-EYTAN et al., 2015; KUO et al., 2016; LUIZELLI et al., 2017a) to efficient operation and management (HWANG; RAMAKRISHNAN; WOOD, 2014; GEMBER-JACOBSON et al., 2014; ZHANG et al., 2016). Nevertheless, NFV is a relatively new and yet maturing paradigm, with various research questions open. As mentioned, one of

the most challenging aspects is how to efficiently find a proper VNF placement and chaining.

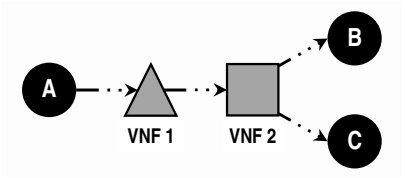
This problem is particularly challenging for many reasons. First, NFV is an inherently distributed network design based on small cloud nodes spread over the network infrastructure. Therefore, depending on how VNFs are positioned and chained in the infrastructure, end-to-end latencies may become intolerable. This problem is aggravated by the fact that processing times tend to be higher, due to the use of virtualization, and may vary, depending on the type of network function and the hardware configuration of the device hosting it. Second, even when deploying VNFs on a single (small) data center, network services might face performance penalties (and limitations) on critical network metrics (such as throughput, latency, and jitter) depending on how network functions are chained and deployed onto physical servers. Third, resource allocation must be performed in a cost-effective manner, preventing over- or under-provisioning of resources. Therefore, placing network functions and programming network flows in a cost-effective manner while ensuring the required network service performance (*e.g.*, maximum tolerable end-to-end delays) represent an essential step towards enabling the use of NFV in production environments. In the following paragraphs, we provide an overview of the virtual network function placement and chaining (VNFPC) problems tackled in this thesis.

Inter-datacenter Virtual Network Function Placement and Chaining Problem. We begin with a general description of the Inter-datacenter placement and chaining problem with an example, illustrated in Figure 1.1. It involves the deployment of two Service Function Chaining requests (SFCs, also referred to as "composite services") onto a backbone network. For the first one, incoming flows must be passed through (an instance of) virtual network function (VNF) 1 (*e.g.*, a firewall), and then VNF 2 (*e.g.*, a load balancer). The second specifies that incoming flows must also be passed through VNF 1, and then VNF 3 (*e.g.*, a proxy). Both SFCs are sketched in Figures 1.1(a) and 1.1(b), respectively.

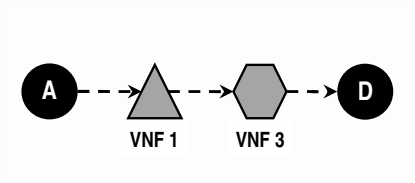
The VNF instances required for each composite service must be placed onto Network Points of Presence (N-PoPs). N-PoPs are infrastructures (*e.g.*, servers, clusters or even datacenters) spread in the network and on top of which (virtual) network functions can be provisioned. Without loss of generality, we assume the existence of one N-PoP associated with every major forwarding device comprising a backbone network (see circles in Figure 1.1(c)). Each N-PoP has a certain amount of resources (*e.g.*, computing power) available. Likewise, each SFC has its own requirements. For example, functions composing an SFC (*e.g.*, caching) are expected to sustain a given load, and thus are associated with a computing power requirement. Also, traffic between functions can be expected to reach some peak throughput, which must be handled by the physical path connecting the N-PoPs hosting those functions.

Given the context above, the first problem we approach is finding a proper placement of VNFs onto distributed N-PoPs and chaining of placed functions, so that overall network resource commitment is minimized. The placement and chaining must ensure that each of the SFC requirements, as well as network constraints, are met. In our illustrating example, a possible deployment is shown in Figure 1.1(c). The endpoints, represented as filled circles, denote

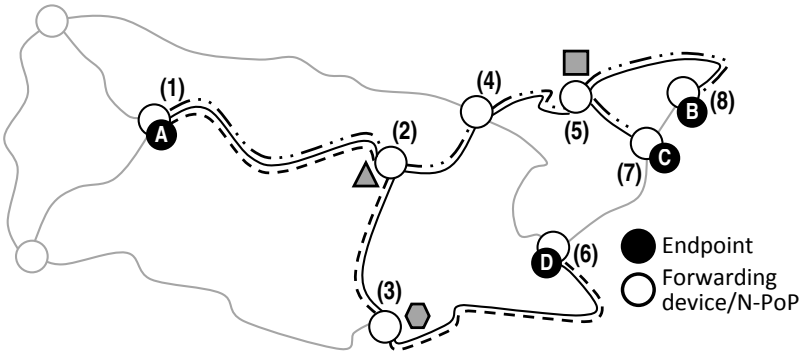
Figure 1.1: Examples of SFCs, and partial view of the network backbone (focusing on the set of N-PoPs available for placing VNFs) considered in our scenario.



(a) Branch flow SFC



(b) Single flow SFC



(c) Network backbone, and deployment view of SFCs

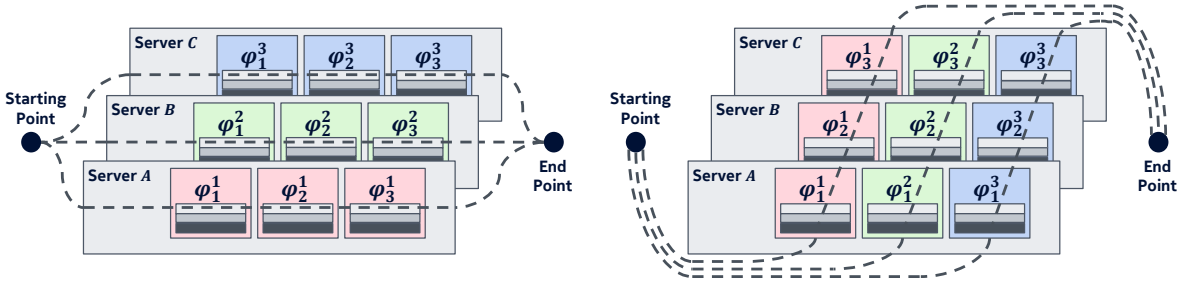
Source: by author (2016).

flows originated/destinated from/to devices/networks attached to core forwarding devices. Observe that both composite services share a same instance of VNF 1, placed on N-PoP (2), therefore minimizing resource allocation as desired. Although relatively simple for small instances, the complexity of solving a VNF placement and chaining problem is NP-complete (LUIZELLI et al., 2017a), as later discussed in Chapter 3.

Intra-datacenter Virtual Network Function Placement and Chaining Problem. The second problem we approach deals specifically with the placement of VNFs onto servers in datacenters (N-PoPs). A solution for the above problem (*i.e*, the Inter-datacenter VNFPC) involves the placement and chaining of SFC requests into multiple (distributed) locations. Particularly, this is the case when SFCs have stringent network requirements (*e.g*, very low end-to-end delays). As a resulting of the planning, SFCs are broken down into sub-chains (*i.e.*, subgraphs) individually placed and chained into specific datacenters. Then, these partial SFC requests are deployed on top of available commodity servers.

In the context of intra-datacenter VNFPC, identifying deployment mechanisms that minimize the provisioning cost of service chains has recently received significant attention from both academia and industry (CLAYMAN et al., 2014; GHAZNAVI et al., 2015; LEWIN-EYTAN et al., 2015; LUIZELLI et al., 2015; BOUET; LEGUAY; CONAN, 2015; LUKOVSZKI; SCHMID,

Figure 1.2: Example of strategies to deploy three given Service Function Chaining.



(a) Gather placement strategy (b) Distribute placement strategy

Source: by author (2017).

2015; LUKOVSZKI; ROST; SCHMID, 2016; LUIZELLI et al., 2017a). However, to the best of our knowledge, existing studies neglected the actual operational cost of NFV deployments. Therefore typical proposed models (*e.g.*, those implemented in NFV orchestrators) might either lead to infeasible solutions (*e.g.*, in terms of CPU requirements) or suffer high penalties on the expected performance.

In an attempt to address this gap, we focused on evaluating and modeling the virtual switching cost in an NFV-based infrastructure (LUIZELLI et al., 2017b). In this environment, virtual switching is an essential building block that enables flexible communication between VNFs. However, its operation comes with an extra cost in terms of computing resources that are allocated specifically to software switching in order to steer the traffic through running services (in addition to computing resources required by VNFs). This cost depends primarily on the way VNFs are internally chained, packet processing requirements, and accelerating technologies (*e.g.*, packet acceleration such as Intel DPDK (INTEL, 2016a)).

Figure 1.2 illustrates possible deployments of service chains on three identical physical servers (A, B and C). As one can see, all service chains φ are composed of VNFs – $\varphi^{1,2,3} = \langle \varphi_1 \rightarrow \varphi_2 \rightarrow \varphi_3 \rangle$. For simplicity, we assume that all traffic steering is done by servers' internal virtual switching, all VNFs require the same amount of processing power to perform their tasks, and that each service chain is associated with a known amount of traffic it has to process. Figures 1.2(a) and 1.2(b) illustrate two widely applied VNF deployment strategies in OpenStack Cloud Orchestrator (ONF, 2015). In Figure 1.2(a), we depict a deployment strategy where all VNFs of a single SFC are deployed on the same server (referred to as "gather"). In contrast, Figure 1.2(b) illustrates a deployment strategy where each VNF of a same SFC is deployed onto different servers (referred to as "distribute"). Observe that all servers have the same number of deployed VNFs and, therefore, they are under the same CPU processing requirements. However, determining the amount of processing resources required for switching inside each server is far from being straightforward even when considering simple deployment strategies. Therefore, understanding these operational overheads in real NFV deployments is

of paramount importance for three main reasons: (*i*) to ensure performance requirements of deployed network services in light of performance limitations (*e.g.*, maximum throughput of a given SFC); (*ii*) to design efficient placement strategies and accurately estimate their operational cost for any arbitrary deployments; and (*iii*) to reduce the operational cost (in particular, CPU consumption of software switching) of NFV providers.

1.2 Hypothesis

The hypothesis we formulate as the fundamental research issue to guide this Ph.D. research work is the following:

In order to take full advantage of the benefits provided by NFV, efficient and scalable strategies should be used to orchestrate the deployment of Service Function Chaining.

Previous work on similar optimization problems have failed to properly consider the placement and chaining of SFCs. In particular, they do not satisfy simultaneously quality, efficiency, and scalability on their solutions. We advocate that the placement and chaining steps on deploying SFCs must be optimized together so as each subproblem (placing VNF and chaining network flows) can benefit from each other. Additionally, existing approaches to place VNFs do not scale to the envisioned extent of NFV deployments (*i.e*, thousands of N-PoPs). Further, as previously mentioned, existing approaches also disregard operational costs and real constraints of NFV environment, potentially threatening the correct operation of deployed network services and increasing their operational cost. We emphasize that scalability in the context of this thesis refers to the ability to solve large instances of the problem in a timely manner (*e.g.*, computing time in the order of minutes/hours). In turn, cost-efficient deployments refers to the ability to come up with optimized solutions in terms of operational cost while meeting quality requirements (such as end-to-end delay).

The objective of this research is to confirm this hypothesis and answer the research questions presented below.

Research Question 1. When performing the deployment of NFV-based SFCs, what are the gains regarding network metrics and resource consumption that can be attained in comparison to traditional network service deployments (*i.e.*, based on physical network functions)?

Research Question 2. Since the SFCP problem is NP-complete, how to provide efficient and near-optimal solutions to SFPC instances on large-scale NFV infrastructures?

Research Question 3. On deploying SFCs onto physical servers, there is a non-negligible CPU cost associated with the service operation. Which are these costs and how to properly estimate them in an NFV environment?

Research Question 4. How to minimize operational costs in SFC deployments? Is it possible to efficiently guide NFV orchestrators on deploying VNFs intra- and inter-server?

Research Question 5. The proper operation of network services requires to constantly monitor them. Is it possible to take advantage of NFV/SDN technologies to efficiently deploy virtualized monitoring services?

1.3 Goals and Contributions

The study proposed here has five main goals: (*i*) formalize the Inter- and Intra-datacenter VNFPC problem; (*ii*) design efficient and scalable algorithmic methods in order to timely compute quality-wise solutions to the VNFPC problem; (*iii*) measure and model operational costs of SFC deployments, as well as network performance limitations, on an NFV-based environment; (*iv*) minimize incurred operational costs in NFV infrastructures; and (*v*) optimize how network traffic steered through service chains is monitored in NFV-based deployments.

The aforementioned goals unfold into a set of contributions of this thesis, described below.

The first set of contributions encompasses the formalization of the virtual network function placement and chaining problem (VNFPC) by means of an Integer Linear Programing model. The devised model considers a wide range of NFV requirements (*e.g.*, network function computing power, flow capacity requirement, etc) (LUIZELLI et al., 2015). Additionally, in order to cope with medium-size NFV infrastructures, we also propose a heuristic procedure which efficiently guides commercial solvers throughout the exploration of good solutions. We compare both optimal and heuristic approaches considering different use cases and metrics, such as the number of instantiated virtual network functions, physical and virtual resource consumption, and end-to-end latencies.

As the second set of contributions, we address the scalability of the VNFPC problem by proposing a novel fix-and-optimize-based heuristic algorithm (LUIZELLI et al., 2017a). It combines mathematical programming (Integer Linear Programming) and a heuristic method (Variable Neighborhood Search (HANSEN; MLADENOVIć, 2001)), so as to produce high-quality solutions for large scale network setups in a timely fashion. We provide strong evidence, supported by an extensive set of experiments, that our heuristic algorithm scales to environments comprised of hundreds of network functions. It produces timely results on average only 20% far from a computed lower bound, and outperforms the existing algorithmic solutions, quality-wise, by a factor of 5. As another important contribution, we prove the NP-completeness nature of the VNFPC problem. As far as we are aware of, this is the first study to formally validate such an important claim.

The third set of contribution of this thesis comprises measuring and modeling operational costs and network metrics of different SFC deployment strategies in real NFV infrastructures (LUIZELLI et al., 2017b). We conduct an extensive and in-depth evaluation, measuring the performance and analyzing the impact of SFC deployment on *Open vSwitch* – the de facto standard software switch for cloud environments (PFAFF et al., 2009; PFAFF et al., 2015; ONF, 2016). Based on our evaluation, we then craft a generalized and abstract cost function

that accurately captures the CPU cost of network switching. We measure and estimate the software switching costs for two widely applied VNF deployment strategies, namely, distribute and gather. On using the distribute deployment, each VNF of an SFC is deployed on top of a different server. In contrast, in the gather, the entire SFC (*i.e.*, all VNFs) is deployed on the same server.

As the fourth set of contributions, we develop a general service chain deployment algorithmic mechanism that considers both the actual performance of the service chains as well as the required extra internal switching resource. This is done by decomposing service chains into sub-chains and deploying each sub-chain on a (possibly different) physical server, in a way that minimizes the total switching overhead cost. We introduce a novel algorithm based on an extension of the well-known reduction from weight matching to min-cost flow problem, and show that it gives an almost optimal solution, with much more efficient run time comparing to the exhaustive search. We evaluate the performance of this algorithm against the fully distribute or fully gather solutions, which are very similar to the placement of standard mechanism commonly utilized on cloud schedulers (*e.g.*, `nova-scheduler` module in OpenStack with load balancing or energy conserving weights) and show that our algorithm significantly outperforms OpenStack (can be up to a factor of 4 in some cases) with respect to operational costs.

As the fifth set of contributions, we tackle the problem of monitoring network traffic in service chains – another important building block for the proper operation of NFV-based network services. We first formalize the DNM (Distributed Network Monitoring) problem, and then we propose an ILP model to solve it. Our optimization model is able to effectively coordinate the monitoring of service chains in NFV environments. The model is aware of SFC topological components, which allows to independently monitor elements within a network service with low overheads in terms of deployed monitoring and network traffic consumption.

We group the main contributions of this thesis into the five areas below:

1. **Inter-Datacenter Placement & Chaining of VNF.**
    - We formalize the virtual network function placement and chaining problem (VNFPC) by means of an Integer Linear Programming model.
    - We prove the NP-completeness nature of the VNFPC problem.
    - We proposed a heuristic procedure that dynamically and efficiently guides the search for solutions performed by commercial solvers. Further, we demonstrate that our heuristic scales to medium-size infrastructures while still finding solutions that are close to optimality.
2. **Limited Scalability of State-of-the-Art Solutions.**
    - We address scalability of VNFPC by proposing a novel fix-and-optimize-based heuristic algorithm.
    - We demonstrate that our proposed method scales to large NFV infrastructures (*i.e.*,

thousands of NFV nodes). Results demonstrates that our proposed method come up with solution on average 5x better than the considered baseline.

3. **Operational Costs.**
    - We provide in-depth insights of SFC deployment strategies in a real NFV infrastructure by measuring overheads and incurred costs.
    - We develop an analytical model to properly estimate the cost of software switching (operational cost) for different placement strategies (and requirements) in NFV infrastructures.
4. **Intra-Datacenter Placement & Chaining of VNF.**
    - We generalize our previous analytical model in order to properly estimate any arbitrary SFC deployment placement strategy.
    - We develop an online algorithm aiming at minimizing operational costs (software switching overheads) for incoming service chain requests.
5. **Efficient Network Traffic Monitoring of SFCs.**
    - We formalize the Distributed Network Monitoring problem and propose and Integer Linear Programming model.
    - We conducted a set experiments in an NFV-based environment in order to assess the cost of sampling network traffic to monitoring systems.
    - We provide insights of the gains attained to DNM solutions.

1.4 Organization

The remainder of this proposal is organized as follows.

- Chapter 2 presents fundamental concepts for fully comprehending our proposal. This background reviews the topics of Network Function Virtualization, Service Function Chaining and NFV enabling technologies. Further, it also discusses state-of-the-art solutions regarding SFC deployment.
- Chapter 3 defines formally the Service Function Chaining Problem (SFPC) as an Integer Linear Programming model and introduces a heuristic algorithm to solve medium-size problem instances.
- Chapter 4 proposes a novel fix-and-optimize math-heuristic to the SFPC problem so as to scale the resolution to the order of thousands of NFV nodes.
- Chapter 5 assesses and accurately estimates the operational costs of SFC deployments in real NFV infrastructures.
- Chapter 6 explores operational cost functions to guide the efficient intra-server deployment of SFCs.

- Chapter 7 defines formally the Distributed Network Monitoring problem as an Integer Linear Programming model.
- Chapter 8 presents the final considerations and directions for future work. The chapter also presents additional academic contributions originated from this thesis.

2 BACKGROUND AND STATE-OF-THE-ART

In this chapter, we first provide an overview on Network Function Virtualization (NFV), Service Function Chaining (SFC) and NFV enabling technologies. Afterwards, we revisit the most prominent approaches regarding SFC deployment in the context of NFV environments.

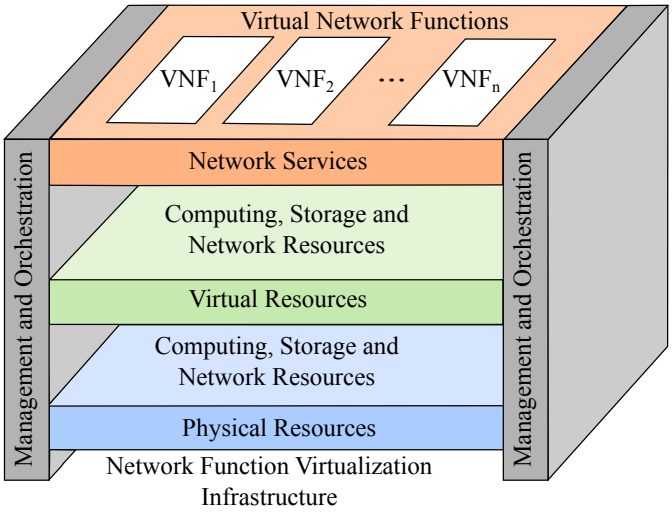
2.1 Network Function Virtualization

Network services are increasingly becoming difficult to deploy and maintain (HAN et al., 2015). Particularly in the telecommunication industry, the provisioning of network services has been based on deploying physical proprietary devices/equipments for each function being part of a given service (MIJUMBI et al., 2016). As the network demand for services is steadily increasing (CISCO, 2016) (*e.g.*, due to video streaming), network operators are required to constantly deploy and/or expand running network services. The wide adoption of hardware-based middleboxes as the core of any network service brings many drawbacks to the operation and growth of current network infrastructure. As an example, in today's infrastructure, it is almost unlikely to realize on-demand service provisioning (or scaling up/down) according to traffic fluctuation. Therefore, these drawbacks directly contribute to increase capital and operational expenditures by network operators/providers (*i.e.*, CAPEX and OPEX) as well as limiting their ability to innovate and/or optimize their operations.

Network Function Virtualization (NFV) has been proposed to address these problems by leveraging virtualization technologies as a new way to design, deploy, and manage network services (MIJUMBI et al., 2016). The main concept of NFV consists of decoupling network functions from the hardware they have been embedded into. Therefore, NFV allows software-based network functions to run over standard high-volume equipments (e.g. servers, switches and storage). On running network functions on general-purpose hardware, NFV breaks the tight dependency on specialized hardware and allows NFs to be fully virtualized. Hence, NFV has the potential to make the most of the benefits from traditional virtualization such as flexibility, dynamic resource scaling, migration, energy efficiency (to name a few). Apart from these benefits, NFV is also expected to foster the innovation of third-parties network function solutions and, therefore, encourage free network market competition. Consequently, NFV is expected to help to lower the acquisition and operation of network solutions and services.

The NFV architecture has recently been defined by the ETSI (European Telecommunications Standards Institute) and by many efforts from the IETF community (Internet Engineering Task Force). The main goal of defining a general architecture is to enable open standardized interfaces amongst the NFV components. Figure 2.1 illustrates the components and their relationships. In essence, the proposed architecture consists of three main components: *(i)* the Network Function Virtualization Infrastructure (NFVI), *(ii)* the Virtual Network Functions (VNF), and *(iii)* the NFV Management and Orchestration (MANO). The NFVI comprises all hardware

Figure 2.1: General NFV architecture proposed by the ETSI.



Source: adapted from (MIJUMBI et al., 2016).

(e.g. servers, storage and network) and software components (e.g. virtualization hypervisor and software switching) that build an NFV environment. Such NFV infrastructure is intended to be organized as a distributed set of NFV-enabled nodes – also known as N-PoPs. Each N-PoP is an NFV-enabled node located in the network infrastructure which has a limited computational power (*i.e.*, it can host only a limited amount of network functions).

In turn, VNFs are the software implementation of network functions (*e.g.*, firewall, Deep Packet Inspection – DPI, IP Multimedia Subsystem – IMS, Ran Access Network – RAN) and are usually deployed as a virtual resource – that is, using virtualization technologies such as traditional virtual machine, containers or micro-kernel. A VNF implementation can be even decomposed into multiple functional blocks which might run individually in different virtualization platforms. This decomposition enables, for instance, a particular functional block to run onto different physical devices and/or to be shared with other NFs. As an example, we can consider the packet classification component (or packet parser) that is usually present in many NFs. With that decomposition, we can have a single component being shared by different NFs.

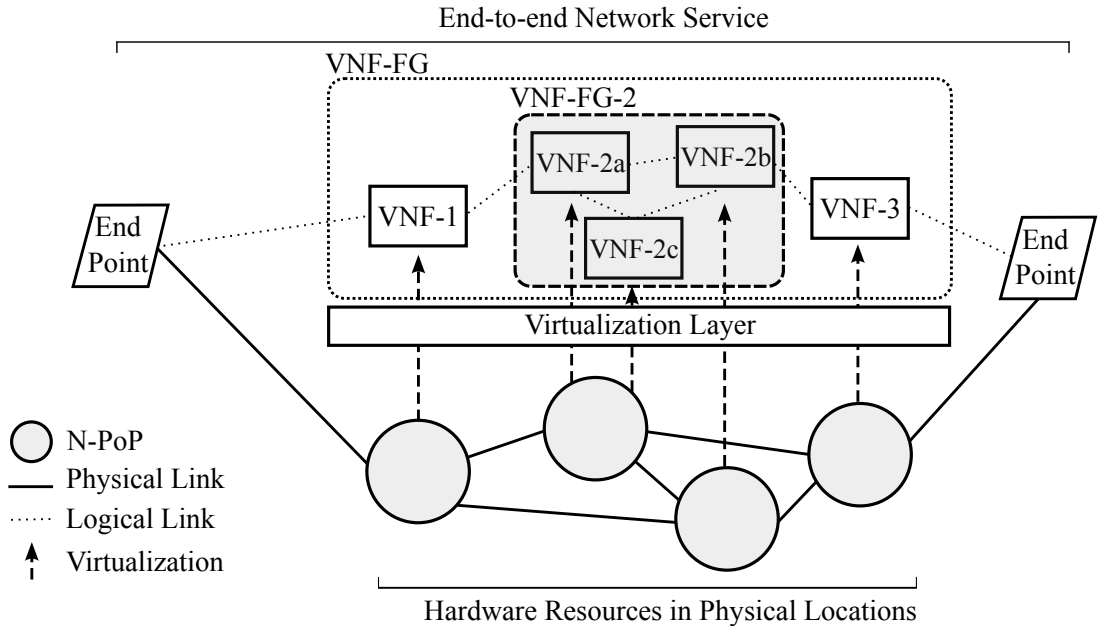
Finally, the NFV MANO defines the Management and Orchestration layer of the NFV environment. The main roles of the MANO are delegated to the NFV Orchestrator (NFVO), to the VNF Manager (VNFM) and to the Virtualized Infrastructure Manager (VIM). In short, the NFVO manages the life-cycle of network services (*e.g.*, (de-)instantiation and service repositories). The VNFM controls the life-cycle of VNFs (*e.g.*, instantiation, updates, scaling in/out). In turn, the VIM is in charge of managing and monitoring of available physical and virtual resources in the NFVI.

2.2 Service Function Chaining

Network services often requires various network functions (NF) ranging from traditional network functions (*e.g.*, firewalls and load balancers), to application-specific features (*e.g.*, HTTP header manipulation, WAN and application acceleration (HALPERN; PIGNATARO, 2015). The realization of an end-to-end network service encompasses the interconnection (or traffic steering) of required NFs in the infrastructure – which is know as Service Function Chaining (SFC).

A Service Function Chaining (SFC) is formally defined as an ordered or partially ordered set of service functions (also referred to NFs) that must be applied to packets, frames and/or flows according to a prior classification (HALPERN; PIGNATARO, 2015). It is important to emphasize that the implied order of an SFC might not be a linear progression. For instance, the current SFC architecture allows a network service to be defined as a forwarding graph (*i.e.*, Direct Acyclic Graph – DAG). In the occurrence of branches in the DAG (*i.e.*, NFs connected to two different endpoints/NFs), packets (or flows) can be forwarded to one, both, or none of the following NFs. Figure 2.2 depicts a NFV-based service being deployed through a combination of VNFs. Note that the VNF-2 is decomposed into three modular components (*i.e*, VNF-2a, VNF-2b, and VNF-2c), which allows each component to run at different locations and over different virtualization platforms.

Figure 2.2: Overview of SFC deployment.



Source: adapted from (MIJUMBI et al., 2016).

On current network service deployments, there is a tightly dependency of SFC and the physical underlying infrastructure. The process of steering the network traffic through NFs is per-

formed by a cumbersome, manual and error-prone process of inter-connecting network cables and crafting routing tables. According to (HALPERN; PIGNATARO, 2015), such topological dependency imposes many constraints (not limited to) on network services delivery such as:

1. Limited ability to (optimally) utilize (and optimize) infrastructure resources. Overtime, SFC deployments might become inefficient due to changing in traffic pattern. Therefore, there is a high operational cost associated with re-planning the chaining of existing NFs in a production environment, making it impractical to be done frequently.
2. Configuration complexity. Due to high dependency on network infrastructure, simple modifications on current deployed SFCs (*e.g.*, adding or removing a NF) require changing the logical and/or physical topology. Therefore, it hinders dynamically network service reconfiguration and slows down the provisioning of new services.
3. Constrained high availability. Redundant NFs or SFCs must be provisioned in the same topology as the primary network service. Consequently, it limits the ability to (dynamically) re-route network traffic to backup instances of NFs on the occurrence of failures or disruption.
4. Transport dependency. There is a wide variety of network transport technologies on current network infrastructure (*e.g.*, VXLAN, MPLS, Ethernet, GRE) which implies NFs to support many technologies simultaneously.
5. Elastic service delivery. There is very little room for maneuver to adjust the infrastructure to future and ongoing demands due to high configuration complexity of changing SFCs. Therefore, it is hard to realize any dynamic adjustment due to the risk and complexity.

By virtualizing NFs, NFV is expected to bring more agility and flexibility to the life cycle management of network services. To overcome technical limitation of traffic steering, Software Defined Networking can be seen as a convenient ally, due to its flexible flow handling capability, thus making placement and chaining technically easier. It is important to mention that other alternative solutions in the context of traffic steering and SFCs have recently been proposed such as NSH (Network Service Header) (QUINN; ELZUR., 2016). For additional information regarding Service Function Chaining, the interested reader is referred to Bhamare et al. (BHAMARE et al., 2016) and Mijumbi et al. (MIJUMBI et al., 2016).

2.3 NFV Enabling Technologies

On shifting the execution of network function application from specialized and dedicated hardware to commodity (and shared) hardware, many technical challenges and limitation come to light. We start discussing these technical limitations on running network functions on commodity hardware. Then, we overview how hardware acceleration technologies has overcame current limitations.

2.3.1 Performance Limitations of Commodity Hardware

Networking application requires intensive interaction between the user's application (*i.e.*, user-space VNF implementation) and the kernel's network stack (which owns and manages hardware NIC through a kernel-space module (CORBET; RUBINI; KROAH-HARTMAN, 2005)). This continuous interaction leads to several well-known performance drawbacks regarding user/kernel synchronization and memory management which ultimately affect the desired performance of virtualized NFV deployments (particularly when processing packets at line-rate).

We start by the synchronization between user- and kernel-space. One of the major performance obstacles of software-based network stack implementations is the communication sequence between the hardware (*i.e.*, Network Interface Controller – NIC) and the user application. As soon network traffic arrives into the NIC, the user-space application should be notified by the kernel. Once the application has gotten the network packets, the application has to release hardware resources as soon as possible so as the hardware can keep working. The two classical existing approaches to address this communication sequence are either using poll mode or interrupt mode. In poll mode, the user application polls for available working elements in the NIC's queue. In contrast, in the interruption mode, when the hardware adds new working elements to the queue, it then triggers an interrupt that invokes a callback function. The implementation of poll mode driver (PMD) requires allocating at least one CPU core to constantly query for available working elements in the queue. Since in commodity hardware CPU is a limited resource, interrupt mode driver has been the default kernel implementation (and up until recently the only one). On the other hand, the major drawback of interrupt mode is the intense need of CPU context switching which dramatically lowers the performance of network-intense application.

Another major drawback of using commodity hardware to run network function regards memory management in modern operating systems. Upon receiving network traffic, the kernel allocates enough memory to handle received network packets. As this procedure is done by the kernel (*i.e.*, in kernel space), it is not allowed to grant access to a user space application on that specific memory region. Consequently, it requires the kernel to copy all network traffic to a different memory region which is granted to user space applications. On perform such intense memory operations, environments naturally suffer performance degradations. Observe that in the context of a virtualized environment, specially the ones running virtual switching, this memory overhead is also observed in the communication between vNICs (*i.e.*, whenever a VNF send/receive network traffic). Specifically in NFV environments, a well-known countermeasure consists of using huge page enabled hardware devices. The usage of huge pages reduces the number of address entries in translation tables (TLB) and, therefore, makes faster to the CPU to translate memory address – which reduces the overhead involved in performing memory-intense operations.

2.3.2 Hardware Acceleration Technologies

Many hardware acceleration technologies have been developed over the past years (*e.g.*, (BONELLI et al., 2012; L. Deri, 2016; RIZZO, 2012)). Accelerating packet processing enables applications such as software-based switching (RIZZO; LETTIERI, 2012; ZHOU et al., 2013; MARTINS et al., 2014; PFAFF et al., 2015; HWANG; RAMAKRISHNAN; WOOD, 2015) and network stack implementation (BERNAL et al., 2016) to process packets fast and efficiently – *i.e.*, mitigating the aforementioned drawbacks of current operating systems.

Acceleration technologies implement new hardware interfaces that are tailored to bypass performance weaknesses in a specific domain of acceleration (*e.g.*, packet processing flow). Despite providing performance gains (*e.g.*, in terms of throughput or latency), such technologies are usually cumbersome to configure and deploy, besides being harder to program for. More importantly, these accelerations technologies usually introduce security risks. In this context, vulnerabilities are related to developing applications in error-prone environments that might be exploited by malicious parties.

In the context of NFV, Intel DPDK (Data Plane Development Kit) (INTEL, 2016a) hardware-acceleration technology has drawn attention from both academy and industry. In short, DPDK is a set of user-space libraries that enables an user space application to fully manage and own the hardware NIC– and, therefore, completely bypassing the kernel networking stack (for instance, avoiding the overhead of copying packets to user-space memory regions). Shifting the entire networking stack to the user space and implementing a poll mode driver, overcomes the two major current performance weaknesses: *(i)* zero-copy packet forwarding from the NIC to the user space application, and *(ii)* no need for context switching to handle interrupts. On the other hand, since DPDK requires the user to manage and own the physical hardware, the user must be given privilege rights to install and run a complete network stack (in the user space) to processes the entire network traffic.

2.3.3 Virtual Switching

Virtual switching is an essential building block in NFV environments, allowing the interconnection of multiple Virtual Network Functions in a flexible, isolated, and scalable manner. Early approaches of virtual switching (*e.g.*, L2 bridges) were static in nature and, therefore, not suitable to cope with the requirements of NFV environments.

Open vSwitch (OvS) is the current standard virtual switching implementation in most cloud environments (PFAFF et al., 2015). The reason for its wide adoption include: *(i)* native support to OpenFlow (MCKEOWN et al., 2008); *(ii)* integrated into most virtualization environments (*e.g.*, Xen and KVM); and *(iii)* part of successful ecosystem that develops open-source cloud solutions (*e.g.*, OpenStack (ONF, 2015)).

Due to its wide adoption, there are many initiatives to enable hardware acceleration tech-

nologies in OVS – ranging from proprietary solutions (*e.g.*, EZchip) to open-source libraries such as Intel DPDK. In particular, the implementation of OvS-DPDK enables to shift the entire packet processing pipeline to the user space, including the NIC poll mode driver and the datapath – which eliminates the overhead of context switches. In contrast to OvS-DPDK, in the kernel implementation, each arriving packet triggers an interrupt to the operating system and requires multiple copy operations (*e.g.*, between the network interface card (NIC) and the virtual switching; or, between virtual switching and VNFs) – which degrades the performance of high-speed network processing. The interested reader is referred to (PFAFF et al., 2015) for additional information.

In addition to acceleration technologies, high-speed virtual switching requires proper tuning to boost its performance in NFV environments (INTEL, 2016b). A well-known configuration for virtualization-intense environments consists of logically separate CPU-cores (disjoint sets). One set of physical cores is assigned to the hypervisor (*i.e.*, kernel and KVM) in order to manage and provision resources (and, therefore, enable networking between all virtual and physical ports). In case of OvS-DPDK, the set of cores assigned to the hypervisor also includes the cores to run DPDK poll mode driver. In turn, the second set of physical CPU-cores is used to run VNFs. Misconfiguration of these disjoint sets of CPU-cores may lead to over usage of specific cores and performance degradation of OVS. It is important to emphasize that the internal architecture of processors (*i.e.*, the way physical cores are interconnected) might lead to other limitations/implications on network performance (LEPERS; QUÉMA; FEDOROVA, 2015) (*e.g.*, some CPU-cores might be closer to PCIe that is connected to the physical NIC).

2.4 Related Work

We now review some of the most prominent research work related to network function virtualization and the network function placement and chaining problem. We start the section by discussing recent efforts aimed at evaluating the technical feasibility of deploying network functions on top of commodity hardware. Then, we review studies carried out to solve different aspects of the virtual network function placement and chaining problem.

Hwang *et al.* (HWANG; RAMAKRISHNAN; WOOD, 2014) propose the NetVM platform to allow network functions based on Intel DPDK technology to be executed at line-speed (*i.e.*, 10 Gb/s) on top of commodity hardware. According to the authors, it is possible to accelerate network processing by mapping NIC buffers to user space memory. In another investigation, Martins *et al.* (MARTINS et al., 2014) introduce a high-performance middlebox platform named ClickOS. It consists of a Xen-based middlebox software, which, by means of alterations in I/O subsystems (back-end switch, virtual net devices and back and front-end drivers), can sustain a throughput of up to 10 Gb/s. The authors show that ClickOS enables the execution of hundreds of virtual network functions concurrently without incurring significant overhead (in terms of delay) in packet processing. The results obtained by Hwang *et al.* and Martins *et*

al. are promising and definitely represent an important milestone to make the idea of virtual network functions a reality.

To the best of our knowledge, network function placement and chaining has not been investigated before the inception of NFV, *e.g.*, for planning and deployment of physical middleboxes. One of the most similar problems in the networking literature is Virtual Network Embedding (VNE): how to deploy virtual network requests on top of a physical substrate (ZHU; AMMAR, 2006; YU et al., 2008; LUIZELLI et al., 2016). In spite of the similarities, solutions to the VNE are not appropriate to the SFPC. The reason is twofold according to (HERRERA; BOTERO, 2016). First, while in VNE we observe one-level mappings (virtual network requests $\rightarrow$ physical network), in NFV environments we have two-level mappings (service function chaining requests $\rightarrow$ virtual network function instances $\rightarrow$ physical network). Second, while the VNE problem considers only one type of physical device (*i.e.*, routers), a much wider number of different network functions coexist in NFV environments.

With respect to placement and chaining of network functions, Barkai *et al.* (BARKAI et al., 2013) and Basta *et al.* (BASTA et al., 2014) have taken a first step toward modeling this problem. Barkai *et al.*, for example, propose mechanisms to program network flows to an SDN substrate taking into account virtual network functions through which packets from these flows need to pass. In short, the problem consists of mapping SDN traffic flows properly (*i.e.*, in the right sequence) to virtual network functions. To solve it in a scalable manner, the authors propose a more efficient topology awareness component, which can be used to rapidly program network flows. Note that they do not aim at providing a (sub)optimal solution to the network function placement and chaining problem as we do in this thesis. Instead, the scope of their work is more of an operational nature, *i.e.*, building an OpenFlow-based substrate that is efficient enough to allow flows – potentially hundred of millions, with specific function processing requirements – to be correct and timely mapped and programmed. Our solution could be used together with Barkai's and therefore help the decision on where to optimally place network functions and how to correctly map network flows.

The work by Basta *et al.*, in turn, proposes an ILP model for network function placement in the context of cellular networks and crowd events. More specifically, the problem addressed is the question on whether or not virtualize and migrate mobile gateway functions to datacenters. When applicable, the model also encompasses the optimal selection of datacenters that will host the virtualized functions and SDN controllers. Although the paper covers optimal virtual function placement, the proposed model is restricted, as it does not have to deal with function chaining. Our proposal is, in comparison, a broader, optimal solution. It can be applied to plan not only the placement of multiple instances of virtual network functions on demand, but also to map and chain service functions.

Moens *et al.* (MOENS; TURCK, 2014) were the first to address VNF placement and chaining, by formalizing it as an optimization problem. The authors consider a hybrid scenario, where SFCs can be instantiated using existing middlebox hardware and/or virtual functions.

Lewin-Eytan *et al.* (LEWIN-EYTAN et al., 2015) follow a similar direction, and use an optimization model along with approximation algorithms to solve the problem. The focus however is VNF placement: where to deploy VNFs, and how to assign traffic flows to them. Their work is relevant for having established a theoretical background for NFV placement, building on two classical optimization problems, namely *facility location* and *generalized assignment*. Nonetheless, network function chaining (key for NFV planning and deployment) is left out of scope. In turn, Ghaznavi et al. (GHAZNAVI et al., 2015) focus on cost-oriented placement of elastic demands. They propose a dynamic mechanism to place, migrate and/or reassign network traffic to cope with traffic fluctuations. However, their considered operational costs are not practical for real deployments (e.g., number of running VNFs). Indiscriminately migration/reallocation of VNFs (e.g., to save energy) might even increase the provider's operational costs (with respect to the internal server's switching).

Mehraghdam et al. (MEHRAGHDAM; KELLER; KARL, 2014), Luizelli et al. (LUIZELLI et al., 2015) and Bari et al. (BARI et al., 2015) introduce joint optimization problems for placement and chaining of VNFs. Mehraghdam et al. (MEHRAGHDAM; KELLER; KARL, 2014) focus on formally specify service chains and on analyzing the proposed ILP model under different objective functions. Luizelli *et al.* (LUIZELLI et al., 2015) also approach the problem from an optimization perspective. The authors introduce a general ILP model which take into account end-to-end delay and resource constraints. They analyze the effect on resource consumption when deploying different service chains. Their proposed heuristic prunes the search space, reducing the complexity of finding feasible solutions. Following, Bari et al. (BARI et al., 2015) propose a similar model focusing on reducing general operational expenditures on datacenters over time (*e.g.*, energy consumption of deployed services). However, as they do not consider insights on which VNFs to bring up/down (according to demand), their deployment solution may end up increasing operational expenditures more than the expected footprint savings. Kuo et al. (KUO et al., 2016) explore the relation between resource consumption on physical servers and links.

Rost et al. (ROST; SCHMID, 2016) and Lukovszki et al. (LUKOVSZKI; ROST; SCHMID, 2016), in turn, were the first to introduce approximation algorithms to the joint NFV optimization problem. Rost et al. (ROST; SCHMID, 2016) present the first polynomial time service chain approximation considering request admission control. The proposed solution is based on classical rounding techniques. Lukovszki et al. (LUKOVSZKI; ROST; SCHMID, 2016) propose a deterministic approximation algorithm based on submodular functions covering incremental deployment. Finally, Lukovszki and Schmid (LUKOVSZKI; SCHMID, 2015) introduce a deterministic online algorithm with logarithm competitive ratio on the length of service chains.

Information presented thus far is summarized in Table 2.1. As one can observe from the state-of-the-art, the area of network function virtualization has recently received much attention from academy and industry. Most of the effort has been focused on engineering ways of running

network functions on top of commodity hardware and, more recently, optimizing aspects in NFV-based environments. Regarding optimization, many studies have been proposed in the context of VNF placement (CLAYMAN et al., 2014; GHAZNAVI et al., 2015; LEWIN-EYTAN et al., 2015) and chaining (MEHRAGHDAM; KELLER; KARL, 2014; LUIZELLI et al., 2015; BARI et al., 2015; RANKOTHGE et al., 2015; BOUET; LEGUAY; CONAN, 2015) problems.

Table 2.1: Summary of related approaches to the VNFPC problem.

Authors	Placement and/or Chaining	Objective	Optimization Method	Infrastructure Size (Scalability)	Evaluation Scenario
(BASTA et al., 2014)	Placement	Min. network load	Exact	up to 20 nodes	Inter-datacenter
(MOENS; TURCK, 2014)	Placement (limited chaining)	Min. resource utilization	Exact	up to 7 nodes	Inter-datacenter
(LEWIN-EYTAN et al., 2015)	Placement	Min. Number of VNFs	Approximation techniques	up to 100 nodes	Inter-datacenter
(GHAZNAVI et al., 2015)	Placement (limited chaining)	Min. number of VNFs / cost of assignment	Heuristic	up to 100 nodes	Intra-datacenter
(MEHRAGHDAM; KELLER; KARL, 2014)	Placement (limited chaining)	Min. number of active nodes / latency	Exact	up to 12 nodes	Inter-datacenter
Our proposal (LUIZELLI et al., 2015)	Placement and chaining	Min. number of active VNFs	Exact / Heuristic	up to 200 nodes	Inter-datacenter
(BARI et al., 2015)	Placement and chaining	Min. general expenditures (deployment and energy costs)	Exact / Heuristic	up to 80 nodes	Inter-datacenter
(KUO et al., 2016)	Placement and chaining	Max. admitted demands	Exact / Dynamic programming	up to 200 nodes	Intra-datacenter
(ROST; SCHMID, 2016)	Placement and chaining	Max. profit	Approximation techniques	–	Theoretical analysis
(LUKOVSZKI; ROST; SCHMID, 2016)	Placement	Min. number of active VNFs	Approximation algorithms	up to 100 nodes	Inter-datacenter
Our proposal (LUIZELLI et al., 2017a)	Placement and chaining	Min. number of active VNFs	Math-Heuristic	> 1000 nodes	Inter-datanceter
Our proposal OCM	Placement and chaining	Min. software switching cost	Heuristic	> 1000 nodes	Intra-datacenter

Source: by author.

In spite of their potentialities, the investigations referred above do not properly scale to large network settings and several SFCs submitted in parallel. Moens *et al.* (MOENS; TURCK, 2014), for example, were limited to small scale scenarios. Lewin-Eytan *et al.* (LEWIN-EYTAN et al., 2015) and Luizelli *et al.* (LUIZELLI et al., 2015) have shown to be only partially effective in scaling to scenarios with a few hundreds nodes and allocating network resources wisely (and the former does not approach chaining, as mentioned earlier). Our proposal in this thesis enhances the state-of-the-art in that it outperforms the solutions of (LEWIN-EYTAN et al., 2015; LUIZELLI et al., 2015), coming up with feasible, high quality solutions for larger scenarios in a timely fashion.

Additionally, none of these studies provides optimized operational costs to NFV deployments. Therefore, most of these solutions are not practical in NFV deployments as they might lead to either infeasible or low-performance solutions. Perhaps the closest approaches to our

proposal are (GHAZNAVI et al., 2015) and (BARI et al., 2015) where the authors propose models to minimize general operational expenditures. However, as discussed later, the operational cost depends to a large extent on many factors including the way VNFs are deployed on physical servers (intra- and inter-server). Therefore, since most of these works focus on arbitrary cost functions (*e.g.*, reducing the amount of deployed VNFs) – the operational cost of maintaining deployed network services is still an open research field. In this regard, we propose a general deployment mechanism specifically tailored to provide performance-oriented deployment.

3 PIECING TOGETHER THE NFV PROVISIONING PUZZLE: EFFICIENT PLACEMENT AND CHAINING OF VIRTUAL NETWORK FUNCTIONS

In the previous chapter, we presented the fundamental concepts of NFV and discussed the most prominent solutions to the placement and chaining of VNFs. In this thesis, we first approach the Inter-datacenter Virtual Network Placement and Chaining (VNFPC) problem[1]. We start this chapter by formalizing the Inter-datacenter VNFPC problem and proposing an optimization model. Further, to cope with medium-size NFV infrastructures, we design a heuristic procedure that prunes the search space and, therefore, reduces the time of finding feasible solutions. Then, we evaluate both optimal and heuristic approaches considering different use cases and metrics, such as the number of instantiated virtual network functions, physical and virtual resource consumption, and end-to-end latencies.

The remainder of this chapter is organized as follows. In Section 3.1, we start with an overview of the VNFPC problem and a brief discussion of topological components of SFC requests. We then formalize the problem using an ILP model and prove its NP-completeness nature. In Section 3.2, we describe the design of the proposed heuristic procedure. Last, in Section 3.3, we present the performance evaluation of both solutions.

3.1 Problem Overview and Optimization Model

As briefly explained in Chapter 1, network function placement and chaining consists of interconnecting a set of network functions (*e.g.*, firewall, load balancer, etc.) through the network to ensure network flows are given the correct treatment. These flows must go through end-to-end paths traversing a specific set of functions. In essence, this problem can be decomposed into three phases: *(i)* placement, *(ii)* assignment, and *(iii)* chaining.

The placement phase consists of determining how many network function instances are necessary to meet the current/expected demand and where to place them in the infrastructure. Virtual network functions are expected to be placed on network points of presence (N-PoPs), which represent groups of (commodity) servers in specific locations of the infrastructure (with processing capacity). N-PoPs, in turn, would be potentially set up either in locations with previously installed commuting and/or routing devices or in facilities such as datacenters.

The assignment phase defines which placed virtual network function instances (in the N-

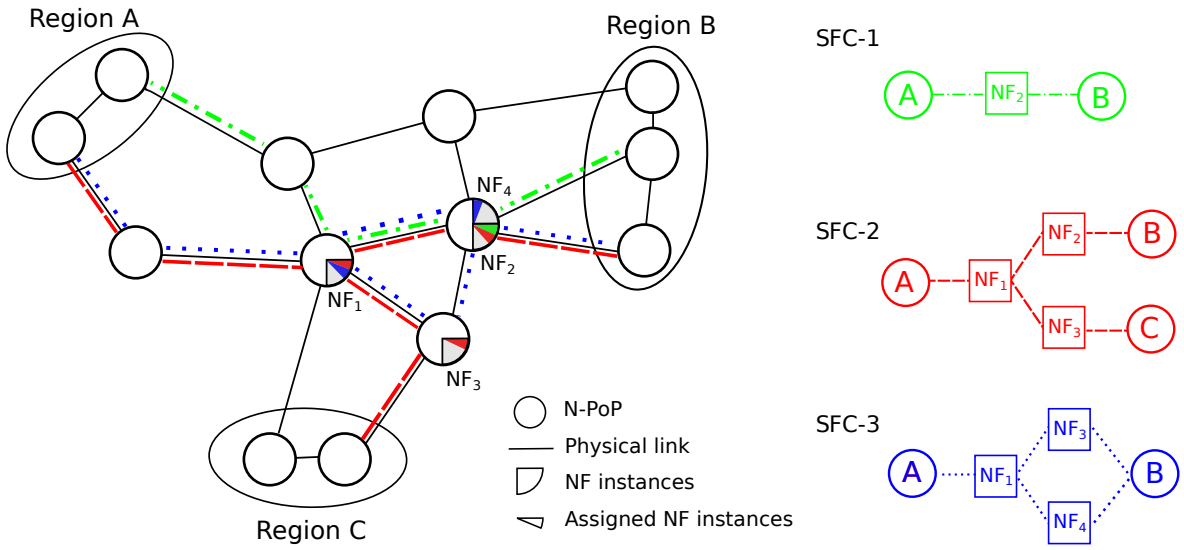
[1]This chapter is based on the following publications:

- Marcelo Caggiani Luizelli, Leonardo Richter Bays, Marinho Pilla Barcellos, Luciana Salete Buriol, Luciano Paschoal Gaspary. **Piecing Together the NFV Provisioning Puzzle: Efficient Placement and Chaining of Virtual Network Functions.** In: Proceedings of IFIP/IEEE International Symposium on Integrated Network Management, 2015.
- Marcelo Caggiani Luizelli, Weverton Luis Cordeiro, Luciana Salete Buriol, Luciano Paschoal Gaspary. **Fix-and-Optimize Approach for Efficient and Large Scale Virtual Network Function Placement and Chaining**. Elsevier Computer Communications, 2017.

PoPs) will be in charge of each flow. Based on the source and destination of a flow, instances are assigned to it in a way that prevents processing times from causing intolerable latencies. For example, it may be more efficient to assign network function requests to the nearest virtual network function instance or to simply split the requested demand between two or more virtual network functions (when possible).

In the third and final phase, the requested functions are chained. This process consists of creating paths that interconnect the network functions placed and assigned in the previous phases. This phase takes into account two crucial factors, namely end-to-end path latencies and distinct processing delays added by different virtual network functions. Figure 3.1 depicts the main elements involved in virtual network function placement and chaining. The physical network is composed of N-PoPs interconnected through physical links. There is a set of SFC requests that contain logical sequences of network functions as well as the endpoints, which implicitly define the paths. Additionally, the provider has a set of virtual network function images that it can instantiate. In the figure, larger semicircles represent instances of network functions running on top of an N-PoP, whereas the circumscribed semicircles represent network function requests assigned to the placed instances. The gray area in the larger semicircles represents processing capacity allocated to network functions that is not currently in use. Dashed lines represent paths chaining the requested endpoints and network functions.

Figure 3.1: Example SFC deployment on a physical infrastructure to fulfill a number of requests.



(a) Physical infrastructure.

(b) SFC requests.

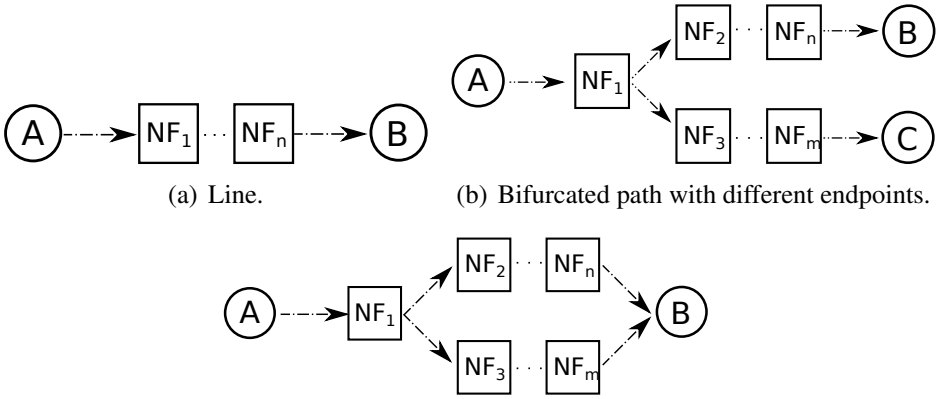
Source: by author (2015).

3.1.1 Topological Components of SFC Requests

SFC requests may exhibit different characteristics depending on the application or flow they must handle. More specifically, such requests may differ topologically and/or in size. In this paper, we consider three basic types of SFC components, which may be combined with one another to form more complex requests. These three variations – *(i)* line, *(ii)* bifurcated path with different endpoints, and *(iii)* bifurcated path with a single endpoint – are explained next.

The simplest topological component that may be part of an SFC request is a line with two endpoints and one or more network functions. This kind of component is suitable for handling flows between two endpoints that have to pass through a particular sequence of network functions, such as a firewall and a Wide Area Network (WAN) accelerator. The second and third topological components are based on bifurcated paths. Network flows passing through bifurcated paths may end up at the same endpoint or not. Considering flows with different endpoints, the most basic component contains three endpoints (one source and two destinations). Between them, there is a network function that splits the traffic into different paths according to a certain policy. A classical example that fits this topological component is a load balancer connected to two servers. As for bifurcated paths with a single end point, we consider a scenario in which different portions of traffic between two endpoints must be treated differently. For example, part of the traffic has to pass through a specific firewall, while the other part, through an encryption function. Figure 3.2 illustrates these topological components. As previously mentioned, more sophisticated SFC requests may be created by freely combining these basic topological components among themselves or in a recursive manner.

Figure 3.2: Basic topological components of SFC requests.



(a) Line.

(b) Bifurcated path with different endpoints.

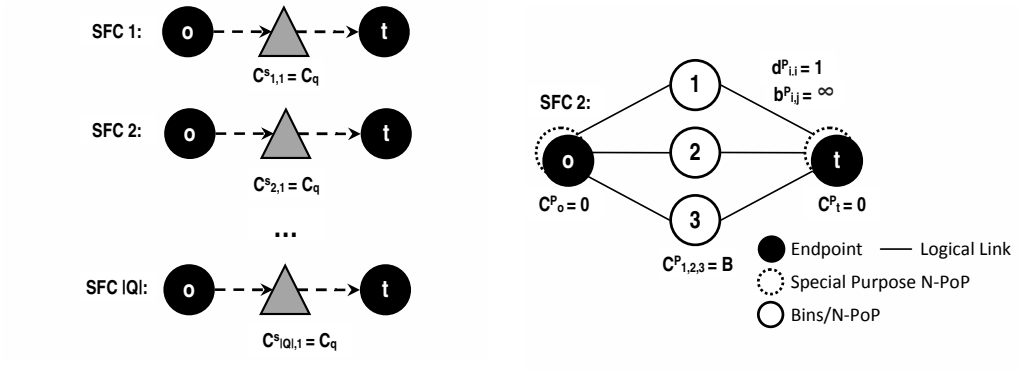
(c) Bifurcated path with a single endpoint.

Source: by author (2015).

3.1.2 Proof of NP-completeness

Next, we show that the VNF placement and chaining (VNFPC) problem belongs to class NP-Complete.

Figure 3.3: Bin Packing instance reduction to VNFPC Problem.



(a) BPP items represented as SFCs.

(b) NFV infrastructure created from Bin Packing instance.

Source: by author (2016).

Lemma 1. *The VNFPC problem belongs to the class NP.*

Proof. A solution for the problem is the sequence of links used to map the SFC, as well as the nodes where the functions were placed. All directed paths between two endpoints (at most $\binom{n/2}{2}$ pairs) of an SFC must be mapped to a valid path in the infrastructure. The solution can be guessed (by means of a nondeterministic Turing machine) and verified in polynomial time, while else accounting for the delay. Endpoints have to be mapped to pre-defined nodes, and checking this is trivial. Moreover, resources consumed by all functions installed in each N-PoP cannot surpass their capacity. While traversing the mapped paths, resource consumption of each node can be accounted. □

Lemma 2. *Any Bin Packing instance can be reduced to an instance of the VNFPC problem.*

Proof. An instance of the Bin Packing Problem (BPP), which is a classical NP-Complete problem (GAREY; JOHNSON, 1979), comprises a set Q of items, a size c_q for each item $q = 1, \ldots, |Q|$, a positive integer bin capacity B, and a positive integer n. The decision version of the BPP asks if there is a partition of Q into disjoint sets $Q_1, Q_2, \ldots, Q_n$ such that the sum of sizes of the items in each subset is at most B. We reduce any instance of the BPP to an instance of the VNFPC using the following procedure:

1. $|Q|$ SFCs are created, each with exactly three nodes and two links. Each SFC has one source endpoint, which must be mapped to N-PoP o, and one sink endpoint, which must

be mapped to t. The middle node is a VNF, which requires a computing power of c_q. Each link demands unitary bandwidth, and the maximum delay requirement of each SFC is set to two (see Figure 3.3(a)).

2. An NFV infrastructure is created with $n + 2$ N-PoPs (nodes) and $2 \cdot n$ logical links (arcs). The n central N-PoPs have a capacity B, and are linked to two other special purpose N-PoPs, called o and t. The logical links have an arbitrarily large bandwidth and an insignificant delay (see Figure 3.3(b)).

The reduction has polynomial time complexity O($|Q|$). □

Theorem 1. *VNFPC is an NP-Complete problem.*

Proof. By the instance reduction presented, if the BPP instance has a solution using n bins, then the VNFPC has a solution using n N-PoPs. Consider that each item of size c_q allocated to a bin j corresponds to place function from SFC q into N-PoP node j. Conversely, if the VNFPC has a solution using n N-PoPs, then the corresponding BPP instance has a solution using n bins. To place function from SFC q into N-PoP node j corresponds to allocate item of size c_q to a bin j. Lemmas 1 and 2 complete the proof.

□

3.1.3 Model Description and Notation

The optimization model presented here is an important building block throughout this thesis. We adopt a revised version of the model proposed by Luizelli *et al.* (LUIZELLI et al., 2015), which captures the placement and chaining aspects we are currently interested in.

We start by describing both the input and output of the model, and establishing a supporting notation. We use superscript letters ^{P} and ^{S} to indicate symbols that refer to physical resources and SFC requests, respectively. Similarly, superscript letters ^{N} and ^{L} indicate references to N-PoPs/endpoints, and the links that connect them. We also use superscript ^{H} to denote symbols that refer to a subset (sub-graph) of an SFC request.

The optimization model we use for solving the VNFPC problem considers a set of composite services Q and a physical infrastructure p, the latter a triple $p = (N^P, L^P, S^P)$. N^P is a set of network nodes (either an N-PoP or a packet forwarding/switching device), and pairs $(i, j) \in L^P$ denote unidirectional physical links. We use two pairs in opposite directions (*e.g.*, (i, j) and (j, i)) to denote bidirectional links. The set of tuples $S^P = \{\langle i, r\rangle \mid i \in N^P \wedge r \in \mathbb{N}^*\}$ contains the actual location (represented as an integer identifier) of N-PoP i. Observe that more than one N-PoP may be associated to the same location (*e.g.* N-PoPs in a specific room or datacenter). The model captures the following resource constraints: computing power for N-PoPs (c_i^P), and one-way bandwidth and delay for physical links ($b_{i,j}^P$ and $d_{i,j}^P$, respectively).

Observe that the forwarding graph of a composite service may represent any topology. Figures 3.2(a) and 3.2(b) illustrate topologies containing simple transitions and flow branches (note

Table 3.1: Glossary of symbols and functions related to the optimization model.

Symbol	Formal specification	Definition
Sets and set objects		
p	$p = (N^P, L^P, S^P)$	Physical network infrastructure, composed of nodes and links
$i \in N^P$	$N^P = \{i \mid i \text{ is a N-PoP}\}$	Network points of presence (N-PoPs) in the physical infrastructure
$(i,j) \in L^P$	$L^P = \{(i,j) \mid i,j \in N^P\}$	Unidirectional links connecting pairs of N-PoPs i and j
$\langle i,r\rangle \in S^P$	$S^P = \{\langle i,r\rangle \mid i \in N^P \wedge r \in \mathbb{N}^*\}$	Identifier r of the actual location of N-PoP i
$m \in F$	$F = \{m \mid m \text{ is a function type}\}$	Types of virtual network functions available
$j \in U_m$	$U_m = \{j \mid j \text{ is an instance of } m \in F\}$	Instances of virtual network function m available
$q \in Q$		Service function chaining (SFC) requests that must be deployed
q	$q = (N^S_q, L^S_q, S^S_q)$	A single SFC request, composed of VNFs and their chainings
$i \in N^S_q$	$N^S = \{i \mid i \text{ is a VNF instance or endpoint}\}$	SFC nodes (either a network function instance or an endpoint)
$(i,j) \in L^S_q$	$L^S_q = \{(i,j) \mid i,j \in N^S\}$	Unidirectional links connecting SFC nodes
$\langle i,r\rangle \in S^S_q$	$S^S_q = \{\langle i,r\rangle \mid i \in N^S \wedge r \in \mathbb{N}^*\}$	Required physical location r of SFC endpoint i
$H^H_{q,i} \in H^S_q$		Distinct forwarding paths (subgraphs) contained in a given SFC q
$H^H_{q,i}$	$H^H_{q,i} = (N^H_{q,i}, L^H_{q,i})$	A possible subgraph (with two endpoints only) of SFC q
$N^H_{q,i}$	$N^H_{q,i} \subseteq N^S_q$	VNFs that compose the SFC subgraph $H^H_{q,i}$
$L^H_{q,i}$	$L^H_{q,i} \subseteq L^S_q$	Links that compose the SFC subgraph $H^H_{q,i}$
Parameters		
$c^P_i \in \mathbb{R}_+$		Computing power capacity of N-PoP i
$b^P_{i,j} \in \mathbb{R}_+$		One-way link bandwidth between N-PoPs i and j
$d^P_{i,j} \in \mathbb{R}_+$		One-way link delay between N-PoPs i and j
$c^S_{q,i} \in \mathbb{R}_+$		Computing power required for network function i of SFC q
$b^S_{q,i,j} \in \mathbb{R}_+$		One-way link bandwidth required between nodes i and j of SFC q
$d^S_q \in \mathbb{R}_+$		Maximum tolerable end-to-end delay of SFC q
Functions		
$f_{type}(m)$	$f_{type} : N^P \cup N^S \to F$	Type of some given virtual network function (VNF)
$f_{cpu}(m,j)$	$f_{cpu} : (F \times U_m) \to \mathbb{R}_+$	Computing power associated to instance j of VNF type m
$f_{delay}(m)$	$f_{delay} : F \to \mathbb{R}_+$	Processing delay associated to VNF type m
Variables		
$y_{i,m,j} \in Y$	$Y = \{\, y_{i,m,j}\,, \forall\, i \in N^P, m \in F, j \in U_m\}$	VNF placement
$a^N_{i,q,j} \in A^N$	$A^N = \{\, a^N_{i,q,j}\,, \forall\, i \in N^P, q \in Q, j \in N^S_q\}$	Assignment of required network functions/endpoints
$a^L_{i,j,q,k,l} \in A^L$	$A^L = \{\, a^L_{i,j,q,k,l}\,, \forall\, (i,j) \in L^P, q \in Q, (k,l) \in L^S_q\}$	Chaining allocation

that flow joins may also be used as illustrated in Figure 3.2(c)). We assume, for simplicity, that the set of virtual paths available to carry traffic flows is known in advance, as such paths are convenient for determining end-to-end delays among pairs of endpoints in our model. In Figure 3.2(b), there are two virtual paths: one starting in A and ending in B, and another starting in A and ending in C (both traversing NFs 1 to n). It is important to emphasize here that such set of virtual paths is defined according to the network policy being implemented. For example, a network policy that allows traffic to go from and to all endpoints requires all paths to be known in advance. In this context, *path* is related to the sequence of virtual network functions and endpoints that a specific traffic should pass through in a particular composite service, instead of being related to a routing path in the physical infrastructure (this is performed by the model – as shown next). This assumption does not restrict the model, neither increase its complexity, since finding paths (*e.g.*, the shortest one) is known to have polynomial time complexity.

The set of virtual paths of a composite service q is denoted by H_q. Each element $H_{q,i} \in H_q$ is one possible sub-graph of q, and contains one source and one sink endpoint, and only one possible forward path. The subsets $N^H_{q,i} \subseteq N^S_q$ and $L^H_{q,i} \subseteq L^S_q$ contain the VNFs and links that belong to $H_{q,i}$.

A composite service $q \in Q$ is an aggregation of network functions and chaining between them. It is represented as a triple $q = (N^S_q, L^S_q, S^S_q)$. Sets N^S_q and L^S_q contain the SFC nodes and virtual links connecting them, respectively. Each SFC has at least two endpoints, denoting specific locations in the infrastructure. The required locations of SFC endpoints is determined in advance, and given by $S^S_q = \{\langle i, r\rangle \,|\, i \in N^S_q \wedge r \in \mathbb{N}^*\}$. For each composite service q, we capture the following resource requirements: computing power required by a network function i ($c^S_{q,i}$), minimum bandwidth required for traffic flows between functions i and j ($b^S_{q,i,j}$), and maximum tolerable end-to-end delay (d^S_q).

F denotes the set of types of VNFs (*e.g.*, firewall, gateway) available for deployment. Each VNF has U_m instances, and may be instantiated at most $|U_m|$ times (*e.g.* due to the number of licenses purchased/available). We denote as $f_{type} : N^P \cup N^S \rightarrow F$ the function that indicates the type of some given VNF, which can be either one instantiated in some N-PoP (N^P) or one requested in an SFC (N^S). We also use functions $f_{cpu} : (F \times U_m) \rightarrow \mathbb{R}_+$ and $f_{delay} : F \rightarrow \mathbb{R}_+$ to denote computing power requirement and processing delay of a VNF.

The model output is denoted by a 3-tuple $\chi = \{Y, A^N, A^L\}$. Variables from $Y = \{\, y_{i,m,j}\,, \forall\, i \in N^P, m \in F, j \in U_m \,\}$ indicate a VNF placement, *i.e.* whether instance j of network function m is mapped to N-PoP i. The variables from $A^N = \{\, a^N_{i,q,j}\,, \forall\, i \in N^P, q \in Q, j \in N^S_q \,\}$, in turn, represent an assignment of required network functions/endpoints. They indicate whether node j (either a network function or an endpoint), required by SFC q, is assigned to node i (either an N-PoP or another device in the network, respectively). Finally, variables from $A^L = \{\, a^L_{i,j,q,k,l}\,, \forall\, (i,j) \in L^P, q \in Q, (k,l) \in L^S_q \,\}$ indicate a chaining allocation, *i.e.* whether the virtual link (k, l) from SFC q is being hosted by physical path (i, j). Each of these variables may assume a value in $\{0, 1\}$.

3.1.4 Model Formulation

Next we describe the linear integer programming formulation for the VNFPC problem. For convenience, Table 3.1 presents the complete notation used in the formulation. The goal of the objective function is to minimize the number of VNF instances mapped on the infrastructure. That choice was based on the fact that resource allocation accounts for a significant and direct impact on operational costs. It is important to emphasize, however, that other objective functions could be adopted (either exclusively or several functions combined). Examples include number of VNF instances deployed, overall bandwidth commitment, end-to-end delays, energy-aware deployments, survivability of SFCs, VNF load balancing, just to name a few. As constraints 1-11 (detailed next) ensure a feasible solution, any objective function being considered that does not depend on any other constraints should work properly. However, there are objective functions that might require some minor modification to the model (*e.g.*, additional constraints) in order to work as expected (which is out of the scope of this thesis).

minimize $\sum_{i \in N^P} \sum_{m \in F} \sum_{j \in U_m} y_{i,m,j}$

subject to

$$\sum_{m \in F} \sum_{j \in U_m} y_{i,m,j} \cdot f_{cpu}(m,j) \leq c_i^P \qquad \forall i \in N^P \quad (3.1)$$

$$\sum_{q \in Q} \sum_{j \in N_q^S : f_{type}(j) = f_{type}(m)} c_{q,j}^S \cdot a_{i,q,j}^N \leq \sum_{j \in U_m} y_{i,m,j} \cdot f_{cpu}(m,j) \qquad \forall i \in N^P\,,\, m \in F \quad (3.2)$$

$$\sum_{q \in Q} \sum_{(k,l) \in L_q^S} b_{q,k,l}^S \cdot a_{i,j,q,k,l}^L \leq b_{i,j}^P \qquad \forall (i,j) \in L^P \quad (3.3)$$

$$\sum_{i \in N^P} a_{i,q,j}^N = 1 \qquad \forall q \in Q\,,\, j \in N_q^S \quad (3.4)$$

$$a_{i,q,k}^N \cdot l = a_{i,q,k}^N \cdot j \qquad \forall \langle i,j \rangle \in S^P\,,\, q \in Q\,,\, \langle k,l \rangle \in S_q^S \quad (3.5)$$

$$a_{i,q,k}^N \leq \sum_{m \in F} \sum_{j \in U_m : m = f_{type}(k)} y_{i,m,j} \qquad \forall i \in N^P\,,\, q \in Q\,,\, k \in N_q^S \quad (3.6)$$

$$\sum_{j \in N^P} a_{i,j,q,k,l}^L - \sum_{j \in N^P} a_{j,i,q,k,l}^L = a_{i,q,k}^N - a_{i,q,l}^N \qquad \forall q \in Q\,,\, i \in N^P\,,\, (k,l) \in L_q^S \quad (3.7)$$

$$\sum_{(i,j) \in L^P} \sum_{(k,l) \in L_{q,t}^H} a_{i,j,q,k,l}^L \cdot d_{i,j}^P + \sum_{i \in N^P} \sum_{k \in N_{q,t}^H} a_{i,q,j}^N \cdot f_{delay}(k) \leq d_q^S \qquad \forall q \in Q\,,\, (N_{q,t}^H, L_{q,t}^H) \in H_q \quad (3.8)$$

$$y_{i,m,j} \in \{0,1\} \qquad \forall\, i \in N^P, m \in F, j \in U_m \quad (3.9)$$

$$a_{i,q,j}^N \in \{0,1\} \qquad \forall\, i \in N^P, q \in Q, j \in N_q^S \quad (3.10)$$

$$a_{i,j,q,k,l}^L \in \{0,1\} \qquad \forall\, (i,j) \in L^P, q \in Q, (k,l) \in L_q^S \quad (3.11)$$

The first three constraint sets refer to limitations of physical resources. Constraint set (3.1) ensures that, for each N-PoP, the sum of computing power required by all VNF instances

mapped to it does not exceed its available capacity. Constraint set (3.2) certifies that the sum of required flow processing capacities does not exceed the amount available on a VNF instance deployed on a given N-PoP. Finally, constraint set (3.3) ensures that the physical path between the required endpoints has enough bandwidth.

Constraint sets (3.4)-(3.6) ensure the mandatory placement of all virtual resources. Constraint set (3.4) certifies that each SFC (and its respective network functions) is mapped to the infrastructure (and only once). Constraint set (3.5), in turn, seeks to guarantee that required endpoints are mapped to network devices in the requested physical locations. Constraint set (3.6) certifies that, if a VNF being requested by an SFC is assigned to a given N-PoP, then at least one VNF instance should be running (placed) on that N-PoP.

The constraints that refer to VNF chaining are the seventh and eighth ones. Constraint set (3.7) ensures that there is an end-to-end path between required endpoints. Constraint set (3.8) certifies that latency constraints on mapped SFC requests are met for each path. The first part of the equation is a sum of the delay incurred by end-to-end latencies between mapped endpoints belonging to the same path. The second part defines the delay incurred by packet processing on VNFs that are traversed by flows in the same path.

3.2 Proposed Heuristic

In this subsection we present our heuristic approach for efficiently placing, assigning, and chaining virtual network functions. We detail each specific procedure it uses to build a feasible solution, and present an overview of its algorithmic process.

In this particular problem, the search procedure performed by the integer programming solver leads to an extensive number of symmetrical feasible solutions. This is mainly because there is a considerable number of potential network function mappings/assignments that satisfy all constraints, in addition to the fact that search schemes conducted by commercial solvers are not specialized for the problem in hand.

To address the aforementioned issues, our heuristic approach dynamically and efficiently guides the search for solutions performed by solvers in order to quickly arrive at high quality, feasible ones. This is done by performing a search to find the lowest possible number of network function instances that meets the current demands. In each iteration, the heuristic employs a modified version of the proposed ILP model in which the objective function is removed and transformed into a constraint, resulting in a more bounded version of the original model. This strategy takes advantage of two facts: first, there tends to be a significant number of feasible, symmetrical solutions that meet our criteria for optimality, and once the lowest possible number of network function instances is determined, only one such solution needs to be found; and second, commercial solvers are extremely efficient in finding feasible solutions.

Algorithm 1 presents a simplified pseudocode version of our heuristic approach, and its details are explained next. The heuristic iteratively attempts to find a more constrained model by

Algorithm 1 Overview of the proposed heuristic.

Input: $Infrastructure\ G$, $set\ Q\ of\ SFCs$, $set\ VNF\ of\ network\ functions$, $timeLimit$

1: $s, s' \leftarrow \emptyset$
2: $upperBound \leftarrow |F|$
3: $lowerBound \leftarrow 1$
4: $nf \leftarrow (upperBound + lowerBound)/2$
5: **while** $nf \geq lowerBound$ and $nf \leq upppperBound$ **do**
6: $nf \leftarrow (upperBound + lowerBound)/2$
7: $Remove\ objective\ function$
8: $Add\ constraint : \sum_{i \in R^P, m \in F, j \in U_m} y_{i,m,j} \leq nf$
9: $s \leftarrow solveAlteredModel(timeLimit)$
10: **if** $s\ is\ feasible$ **then**
11: $s' \leftarrow s$
12: $upperBound \leftarrow nf$
13: **else**
14: $lowerBound \leftarrow nf$
15: **end if**
16: **end while**
17: **if** $s' = \emptyset$ **then**
18: **return** $return\ infeasible\ solution$
19: **else**
20: $return\ s'$
21: **end if**

dynamically adjusting the number of network functions that must be instantiated on the infrastructure. The upper bound of this search is initially set to the maximum number of network functions that may be instantiated on the infrastructure (line 2), while the lower bound is initialized as 1 (line 3). In each iteration, the maximum number of network function instances allowed is represented by variable nf, which is increased or decreased based on the aforementioned upper and lower bounds (line 6). After nf is updated, the algorithm transforms the original model into the bounded one by removing the objective function (line 7) and adding a new constraint (line 8), considering the computed value for nf. The added constraint is shown in Equation 10.

$$\sum_{i \in R^P, m \in F, j \in U_m} y_{i,m,j} \leq nf \tag{3.12}$$

In line 9, a commercial solver is used to obtain a solution for the bounded model within an acceptable time limit. In each iteration, the algorithm stores the best solution found so far (*i.e.*, the solution s with the lowest value for nf – line 11). Afterwards, it adjusts the upper or lower

bound depending on whether the current solution is feasible or not (lines 12 and 14). Last, it returns the best solution found (s', which represents variables y, A^N and A^L).

Although the proposed heuristic uses an exact approach to find a feasible solution for the problem, $timeLimit$ (in line 9) should be fine-tuned considering the size of the instance being handled to ensure the tightest solution will be found. In our experience, for example, a time limit in the order of minutes is sufficient for dealing with infrastructures with 200 N-PoPs.

3.3 Evaluation

In order to evaluate the provisioning of different types of SFCs, the ILP model formalized in the previous section was implemented and run in *CPLEX Optimization Studio*[2] version 12.4. The heuristic, in turn, was implemented and run in Python. All experiments were performed on a machine with four Intel Xeon E5-2670 processors and 56 GB of RAM, using the Ubuntu GNU/Linux Server 11.10 x86_64 operating system.

3.3.1 Setup

We consider four different types of SFC components. Each type uses either one of the topological components described in Subsection 3.1.1 or a combination of them. The first component is a line composed of a single firewall between the two endpoints (Figure 3.2(a)). The second component used consists of a bifurcated path with different endpoints (Figure 3.2(b)). This component is composed of a load balancer splitting the traffic between two servers. These two types of components are comparable since their end-to-end paths pass through exactly one network function. The third and fourth components use the same topologies of the previously described ones, but vary in size. The third component is a line (like Component 1) composed of two chained network functions – a firewall followed by an encryption network function (*e.g.*, VPN). The fourth component is a bifurcated path (like Component 2), but after the load balancer, traffic is forwarded to one more network function – a firewall. These particular network functions were chosen due to being commonly referenced in recent literature; however, they could be easily replaced with any other functions if so desired. All network functions requested by SFCs have the same requirements in terms of CPU and bandwidth. Each network function requires 12.5% of CPU, while the chainings between network functions require 1Gbps of bandwidth. When traffic passes through a load balancer, the required bandwidth is split between the paths. The values for CPU and bandwidth requirements were fixed after a preliminary evaluation, which revealed that they did not have a significant impact on the obtained results. Moreover, the establishment of static values for these parameters facilitates the assessment of the impact of other, more important factors.

The processing times of virtual network functions (*i.e.*, the time required by these functions

[2]http://www-01.ibm.com/software/integration/optimization/cplex- optimization-studio/

to process each incoming packet) considered in our evaluation are shown in Table 3.2. These values are based on the study conducted by Dobrescu *et al.* (DOBRESCU; ARGYRAKI; RATNASAMY, 2012), in which the authors determine the average processing time of a number of software-implemented network functions.

Table 3.2: Processing times of physical and virtual network functions used in our evaluation.

Network Function	Processing Time (physical)	Processing Time (virtual)
Load Balancer	0.2158 msec	0.6475 msec
Firewall	2.3590 msec	7.0771 msec
VPN Function	0.5462 msec	1.6385 msec

Source: by author (2015).

Networks used as physical substrates were generated with Brite[3]. The topology of these networks follows the Barabasi-Albert (BA-2) (ALBERT; BARABÁSI, 2000) model. This type of topology was chosen as an approximation of those observed in real ISP environments. Physical networks have a total of 50 N-PoPs, each with total CPU capacity of 100%, while the bandwidth of physical links is 10 Gbps. The average delay of physical links is 30ms. This value is based on the study conducted by Choi *et al.* (CHOI et al., 2007), which characterizes typical packet delays in ISP networks.

In order to provide a comparison between virtualized network functions and non-virtualized ones, we consider baseline scenarios for each type of SFC. These scenarios aim at reproducing the behavior of environments that employ physical middleboxes rather than NFV. Our baseline consists of a modified version of our model, in which the total number of network functions is exactly the number of different functions being requested. Moreover, the objective function attempts to find the minimum chaining length between endpoints and network functions. In baseline scenarios, function capacities are adjusted to meet all demands and, therefore, we do not consider capacity constraints. Further, processing times are three times lower than those in virtualized environments. This is in line with the study of Basta *et al.* (BASTA et al., 2014). These processing times, like the ones related to virtual network functions, are shown in Table 3.2.

In our experiments, we consider two different profiles of network function instances. In the first one, instances may require either 12.5% or 25% of CPU, leading to smaller instance sizes. In the second profile, instances may require 12.5% or 100%, leading to larger instances overall. We first evaluate our optimal approach considering individual types of requests. Next, we evaluate the effect of a mixed scenario with multiple types of SFCs. Last, we evaluate our proposed heuristic using large instances. Each experiment was repeated 30 times, with each

[3]http://www.cs.bu.edu/brite/

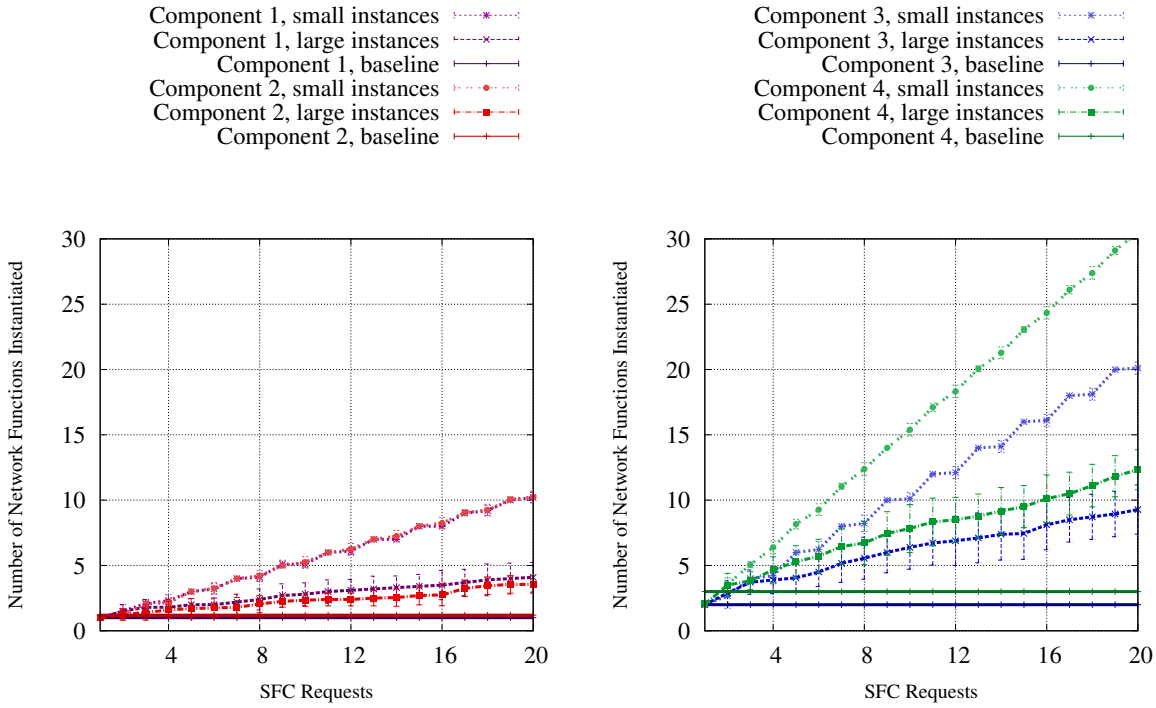
repetition using a different physical network topology. All results have a confidence level of 90% or higher.

3.3.2 Results

First, we analyze the number of network functions instances needed to cope with an increasing number of SFC requests. Figure 3.4 depicts the average number of instantiated network functions with the number of SFC requests varying from 1 to 20. At each point on the graph, all previous SFC requests are deployed together. It is clear that the number of instances is proportional to the number of SFC requests. Further, we observe that smaller instance sizes lead to a higher number of network functions being instantiated. Considering small instances, scenarios with Components 1 and 2 require, on average, 10 network function instances (Figure 3.4(a)). In contrast, scenarios with Components 3 and 4 require, on average, 20 and 30 instances (Figure 3.4(b)), respectively. For large instances, scenarios with Components 1 and 2 require, respectively, 4 and 3 network function instances, while those with Components 3 and 4 require 9 and 12 instances on average. These results demonstrate that the number of virtual network functions in an SFC request has a much more significant impact on the number of instances needed to service such requests than the chainings between network functions and endpoints. This can be observed, for example, in Figure 3.4(a), in which Components 1 and 2 only differ topologically and lead to, on average, the same number of instances. In contrast, Figure 3.4(b) shows that when handling components of type 4 (which have a higher number of network functions than those of type 3), a significantly higher number of network function instances is required.

Figure 3.5 illustrates the average CPU overhead (*i.e.*, allocated but unused CPU resources) in all experiments. Each point on the graph represents the average overhead from the beginning of the experiment until the current point. In all experiments, CPU overheads tend to be lower when small instances are used. When large instances are allocated, more resources stay idle. Considering small instances, Components 1 and 2 (Figure 3.5(a)) lead to, on average, CPU overhead of 7.80% and 7.28%, respectively. Components 3 and 4 (Figure 3.5(b)) lead to, on average, 6.58% and 3.18% CPU overhead. In turn, for large instances, Components 1 and 2 lead to average CPU overheads of 45.61% and 38.68%, respectively. Components 3 and 4 lead to, on average, 40.21% and 40.36% CPU overhead. Observed averages demonstrate that the impact of instance sizes is notably high, with smaller instances leading to significantly lower overheads. Further, we can observe that, in general, CPU overheads tend to be lower when higher numbers of SFCs are being deployed. As more requests are being serviced simultaneously, network function instances can be shared among multiple requests, increasing the efficiency of CPU allocations. In these experiments, the baseline has 0% of CPU overhead as network functions are planned in advance to support the exact demand. Since in NFV environment network function instances are hosted on top of commodity hardware (as opposed to specialized middleboxes), these overheads – especially those observed for small instances – are deemed acceptable, as

Figure 3.4: Average number of network function instances.



(a) Components 1 and 2.

(b) Components 3 and 4.

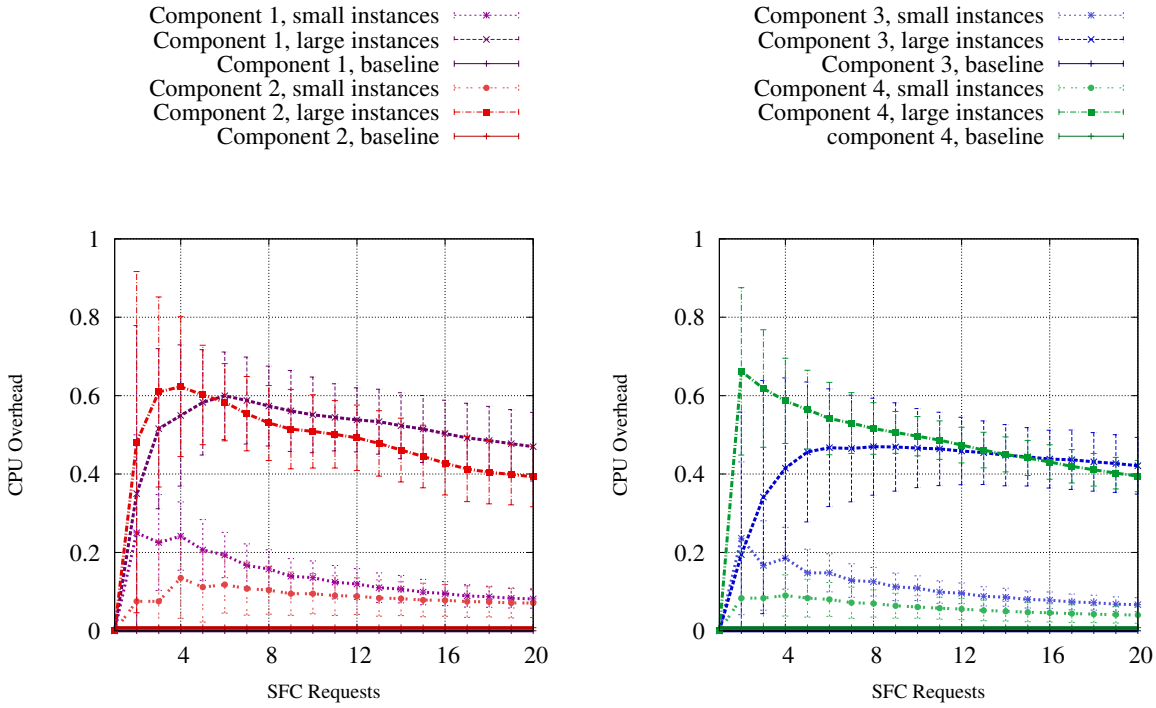
Source: by author (2015).

they do not incur high additional costs.

Next, Figure 3.6 shows the average overhead caused by chaining network functions (through virtual links) in each experiment. This overhead is measured as the ratio between the effective bandwidth consumed by SFC virtual links hosted on the physical substrate and the bandwidth requested by such links. In general, the actual bandwidth consumption is higher than the total bandwidth required by SFCs, due to the frequent need to chain network functions through paths composed of multiple physical links. The absence of overhead is observed only when each virtual link is mapped to a single physical link (ratio of 1.0), or when network functions are mapped to the same devices as the requested endpoints (ratio < 1.0). Lower overhead rates may potentially lead to lower costs and allow more SFC requests to be serviced.

Considering large instances, the observed average overhead is 50.49% and 69.87% for scenarios with Components 1 and 2, respectively. In turn, Components 3 and 4 lead to overhead ratios of 116% and 72.01%. This is due to the low number of instantiated network functions (Figure 3.4), which forces instances to be chained through long paths. Instead, when small instances are considered (*i.e.*, more instances running in a distributed way), overheads tend to be lower. Components 1 and 2 lead to, on average, 44.30% and 57.60% bandwidth overhead, while Components 3 and 4, 44.53% and 53.41%, respectively. When evaluating bandwidth overheads,

Figure 3.5: Average CPU overhead of network function instances.



(a) Components 1 and 2.

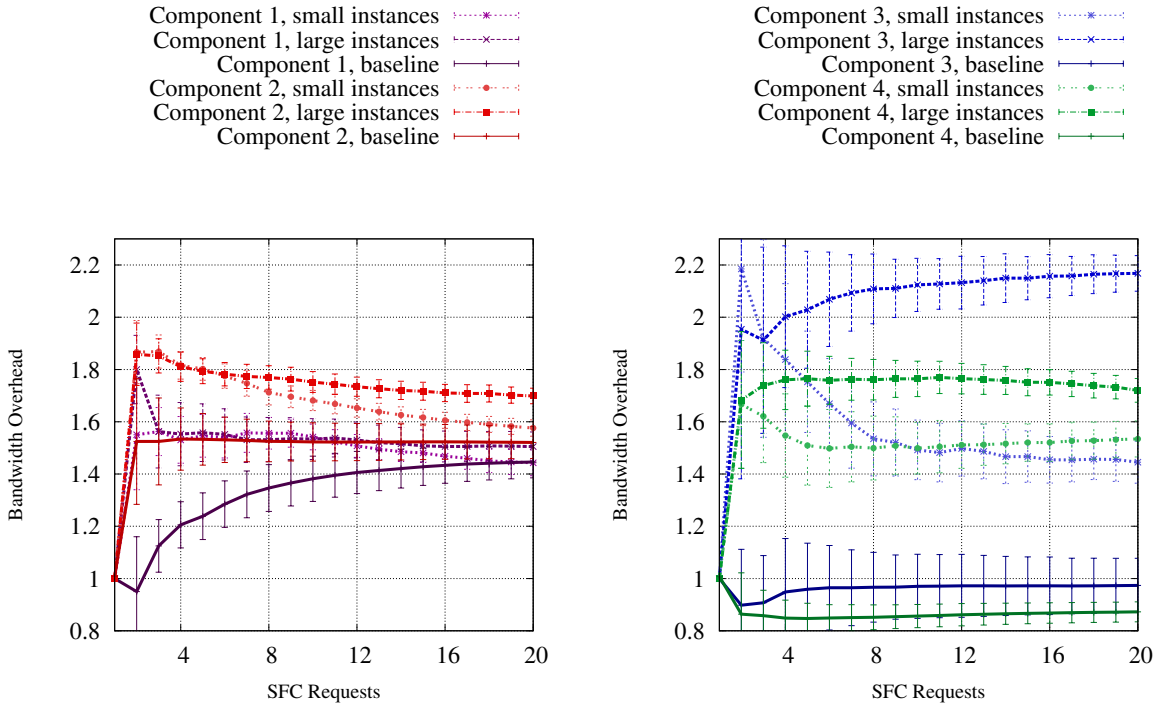
(b) Components 3 and 4.

Source: by author (2015).

we can observe that the topological structure of SFC requests has the most significant impact on the results (in contrast to previously discussed experiments). More complex chainings tend to lead to higher bandwidth overheads, although these results are also influenced by other factors such as instance sizes and the number of instantiated functions. In these experiments the baseline overhead tends to be lower than the others as the objective function prioritizes shortest paths (in terms of number of hops) between endpoints and network functions.

Figure 3.7 depicts the average end-to-end delay, in milliseconds, observed between endpoints in all experiments. The end-to-end delay is computed as a sum of the path delays and network function processing times. In this figure, results for scenarios with small and large instances are grouped together, as average delays are the same. The observed end-to-end delay for all components tends to be lower than the delay observed for the baseline scenario. This is mainly due to the better positioning of network functions and chainings between them. Furthermore, the model promotes a better utilization of the variety of existing paths in the infrastructure. Although the baseline scenario aims at building minimum chainings (in terms of hops), we observe that: *(i)* minimum chaining does not always lead to global minimum delay; *(ii)* when baseline scenarios overuse the shortest paths, other alternative paths remain unused due to the depletion of resources in specific locations (mainly in the vicinity of highly inter-

Figure 3.6: Average bandwidth overhead of SFCs deployed in the infrastructure.



(a) Components 1 and 2.

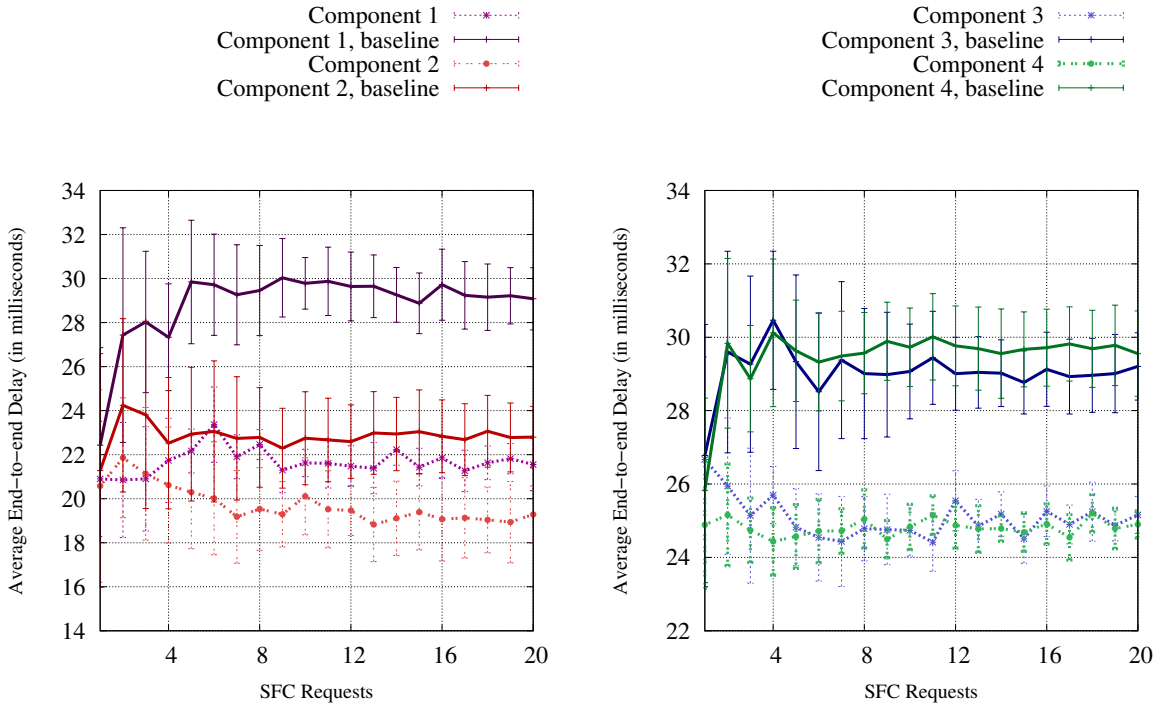
(b) Components 3 and 4.

Source: by author (2015).

connected nodes). In comparison with baseline scenarios, Component 1 leads to, on average, 25% lower delay (21.55ms compared to 29.07ms), while Component 2 leads to, on average, 15.40% lower delay (19.28ms compared to 22.79ms). In turn, Component 3 leads to, on average, 13.86% lower delay than its baseline (25.15ms compared to 29.20ms), while Component 4 leads to 15.75% lower delay (24.89ms compared to 29.55ms). In summary, even though our baseline scenarios are planned in advance to support exact demands and we consider processing times of virtual network functions to be three times those of physical ones, end-to-end delays are still lower in virtualized scenarios. This advantage may become even more significant as the estimated processing times of virtual network functions get closer in the future to those observed in physical middleboxes.

After analyzing the behavior of SFCs considering homogeneous components, we now analyze the impact of a mixed scenario. In it, Components 1, 2, and 4 are repeatedly deployed in the infrastructure sequentially. Figure 3.8 presents the results for the mixed scenario. Although there are different topological SFC components being deployed together in the same infrastructure, the results exhibit similar tendencies as those of homogeneous scenarios. In Figure 3.8(a), we observe that the average number of network functions (on average, 17 network functions when considering small instances and 9 functions considering large instances) is proportional

Figure 3.7: Average end-to-end delay of SFCs deployed in the infrastructure.



(a) Components 1 and 2.

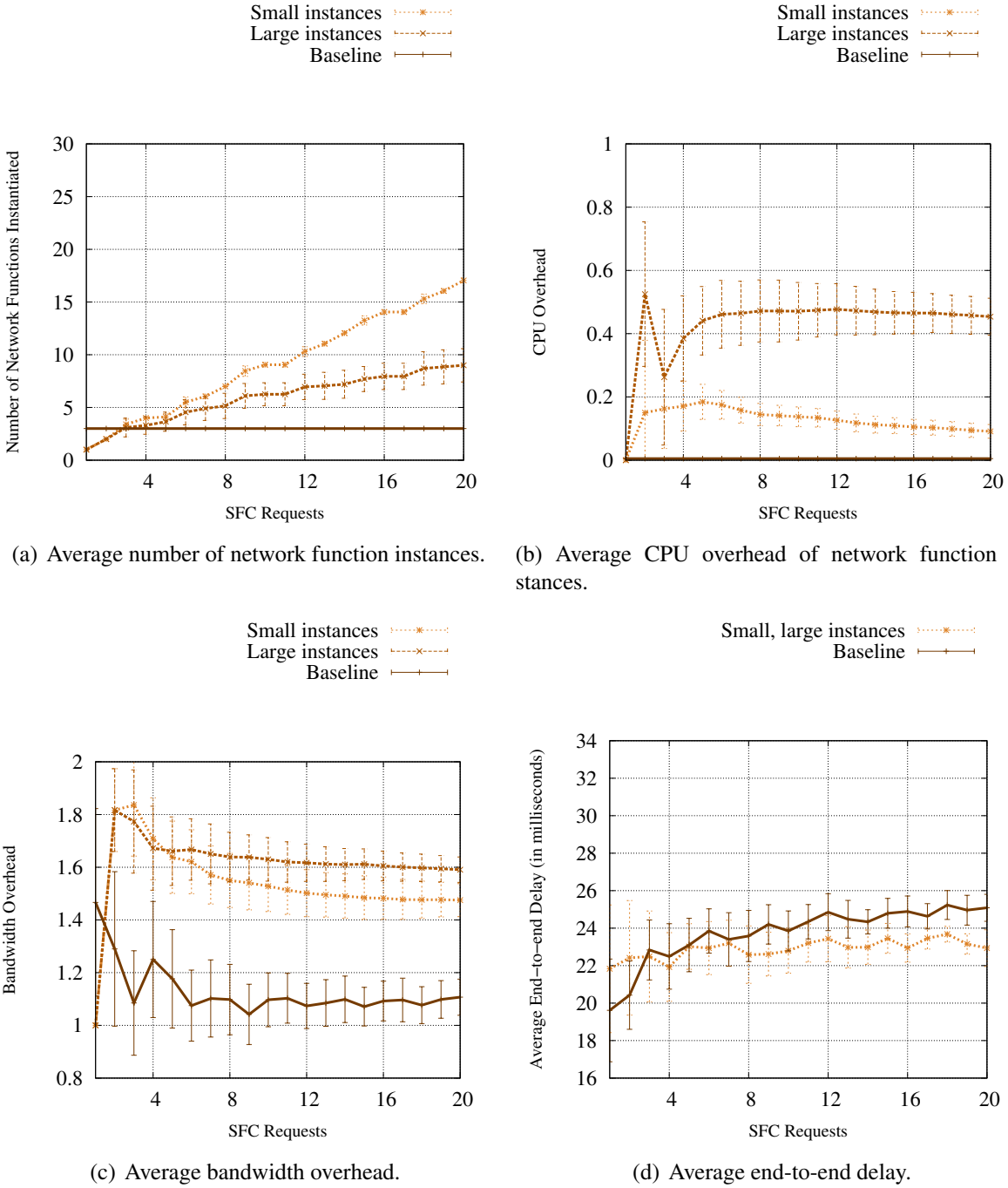
(b) Components 3 and 4.

Source: by author (2015).

to the obtained average values depicted in Figures 3.4(a) and 3.4(b). The average CPU overhead also remains similar (9.12% considering small instances and 45.37% considering large ones). In turn, the average overhead caused by chaining network functions in the mixed scenario is of 59.06% and 47.51%, for small and large instances, respectively. Despite these similarities, end-to-end delays tend to be comparatively lower than the ones observed in homogeneous scenarios. The delay observed in the proposed chaining approach is 8.57% lower than that of the baseline (22.93ms in comparison to 25.08ms). This is due to the combination of requests with different topological structures, which promotes the use of a wider variety of physical paths (which, in turn, leads to lower overutilization of paths). Similarly to homogeneous scenarios, average end-to-end delays are the same considering small and large instances.

We now proceed to the evaluation of our proposed heuristic approach. The heuristic was subjected to the same scenarios as the ILP model, in addition to the ones with a larger infrastructure. Considering the scenarios presented so far (*i.e.*, with physical infrastructures with 50 nodes and 20 SFC requests), our heuristic was able to find an optimal solution in all cases. We omit such results due to space constraints. We emphasize, however, that the heuristic approach was able to find an optimal solution in a substantially shorter time frame in comparison to the ILP model, although the solution times of both approaches remained in the order of minutes.

Figure 3.8: Mixed scenario including Components 1, 2, and 4.



(a) Average number of network function instances.

(b) Average CPU overhead of network function instances.

(c) Average bandwidth overhead.

(d) Average end-to-end delay.

Source: by author (2015).

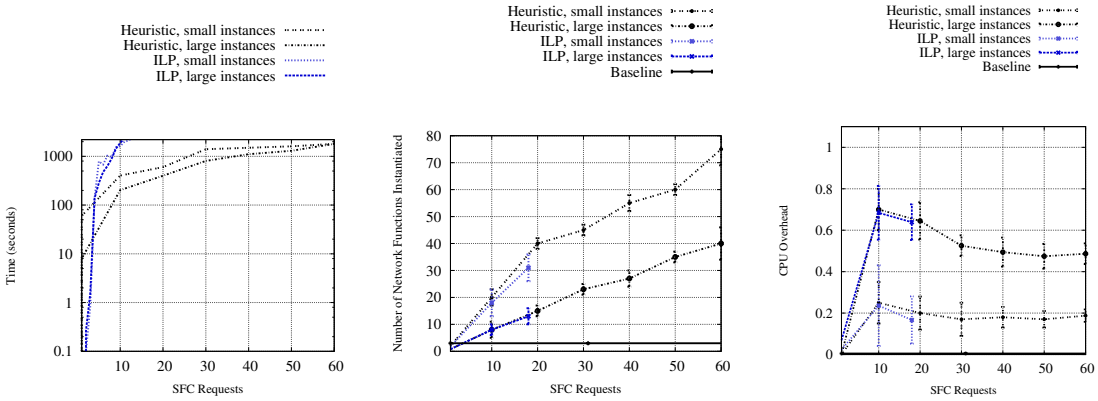
The average solution times of the ILP model and the heuristic considering all scenarios were of, respectively, 8 minutes and 41 seconds and 1 minute and 21 seconds.

Last, we evaluate our heuristic approach on a large NFV infrastructure. In this experiment, we consider a physical network with 200 N-PoPs and a maximum of 60 SFC components of type 4. The delay limit was scaled up to 90ms in order to account for the larger network size.

Figure 3.9(a) depicts the average time needed to find a solution using both the ILP model and the heuristic. The ILP model was not able to find a solution in a reasonable time in scenarios with more than 18 SFCs (the solution time was longer than 48 hours). The heuristic approach, in turn, is able to properly scale to cope with this large infrastructure, delivering feasible, high-quality solutions in a time frame of less than 30 minutes. As in previous experiments, small network function instances lead to higher solution times than large ones. This is mainly because smaller instances lead to a larger space of potential solutions to be explored.

Although the heuristic does not find the optimal solution (due to time constraints), Figures 3.9(b), 3.9(c), 3.9(d) and 3.9(e) show that the solutions obtained through this approach present a similar level of quality to the ones obtained optimally. Figure 3.9(b) depicts the average number of instantiated network functions with the number of SFC requests varying from 1 to 60. As in previous experiments, the number of instances remains proportional to the number of SFC requests. Smaller instance sizes lead to a higher number of network functions being instantiated. Considering small sizes, 75 network functions instances are required on average. In contrast, for large sizes, 40 instances are required on average. Figure 3.9(c), in turn, illustrates the average CPU overhead. For small instances, CPU overhead is limited to 18.77%, while for large instances it reaches 48.65%. Similarly to the results concerning the number of network function instances, CPU overheads in these experiments also follow the trends observed in previous ones. Next, Figure 3.9(d) presents bandwidth overheads. Small instances lead to a bandwidth overhead of 300%, while for large instances this overhead is, on average, 410%. These particularly high overheads are mainly due to the increase on the average length of end-to-end paths, as the physical network is significantly larger. Note that the bandwidth overhead observed in the baseline scenario (198%) is also significantly higher than those observed in experiments employed on the small infrastructure. Last, 3.9(e) depicts the average end-to-end delay observed in large infrastructures. In line with previous results, the end-to-end delay tends to be lower than the delay observed in the baseline scenario. The scenario considering Component 4 presents, on average, 17.72% lower delay than the baseline scenario (70.30ms compared to 82.76ms). In short, these results demonstrate that: *(i)* the heuristic is able to find solutions with a very similar level of quality as the optimization model for small infrastructures; and *(ii)* as both infrastructure sizes and the number of requests increase, the heuristic is able to maintain the expected level of quality while still finding solutions in a short time frame.

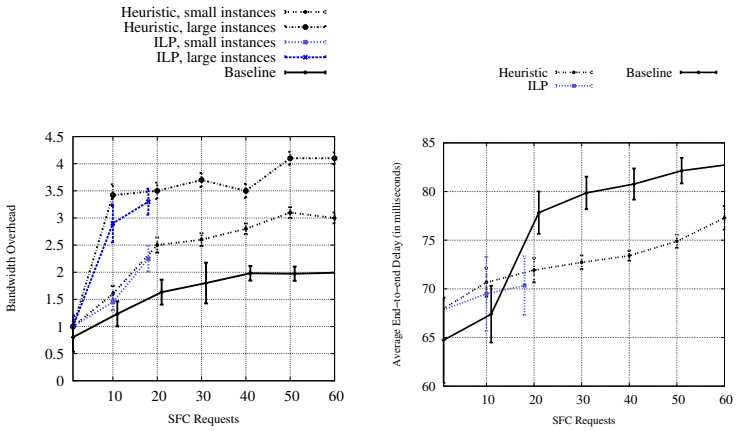
Figure 3.9: Scenario considering a medium-size NFV infrastructure and components of type 4.



(a) Average time needed to find a solution.

(b) Average number of network function instances.

(c) Average CPU Overhead of network function instances.



(d) Average bandwidth overhead of SFCs deployed in the infrastructure.

(e) Average end-to-end delay of SFCs deployed in the infrastructure.

Source: by author (2015).

4 A FIX-AND-OPTIMIZE APPROACH FOR EFFICIENT AND LARGE SCALE VIRTUAL NETWORK FUNCTION PLACEMENT AND CHAINING

In the previous chapter, we formally defined the Inter-datacenter VNFPC problem and proposed both an optimization model and a heuristic procedure to cope with medium-size infrastructures. In this chapter, we go one step further and address the scalability of the Inter-datacenter VNFPC problem in order to solve large instances (*i.e.*, thousands of NFV nodes). We propose a novel fix-and-optimize approach which combines the previously defined optimization model and Variable Neighborhood Search (VNS)[1].

The remaining of this chapter is organized as follows. In Section 4.1 we describe the design of our proposed math-heuristic based solution and, in Section 4.2, we extensively evaluate it in comparison to state-of-the-art approaches.

4.1 Fix-and-Optimize Heuristic for the VNF Placement & Chaining Problem

As previously shown in Chapter 3, VNFPC optimization problem is NP-complete. To tackle this complexity and come up with high-quality solutions efficiently, we introduce an algorithm that combines mathematical programming with heuristic search. Thus, our optimization model (defined in Section 3.1.4) is an important building block for the design of such solution. Our algorithm is based on Fix-and-Optimize and Variable Neighborhood Search (VNS), techniques that have been successfully applied to solve large/hard optimization problems (HANSEN; MLADENOVIć, 2001; HELBER; SAHLING, 2010; SAHLING et al., 2009). A comprehensive view of our proposal is shown in Algorithm 2.

4.1.1 Overview

In this subsection we provide a general view of our algorithm, using Figure 4.1 as basis. It illustrates the search for a solution to the VNFPC problem, considering the composite services and network infrastructure shown in Figure 1.1.

In our algorithm, we first compute an initial, feasible solution (or *configuration*) χ to the optimization problem (see Subsection 4.1.3). In the example shown in Figure 4.1, the initial configuration (1) has VNF 1 placed on N-PoPs 3 and 5; VNF 2 placed on N-PoP 5; and VNF 3 placed on N-PoP 3.

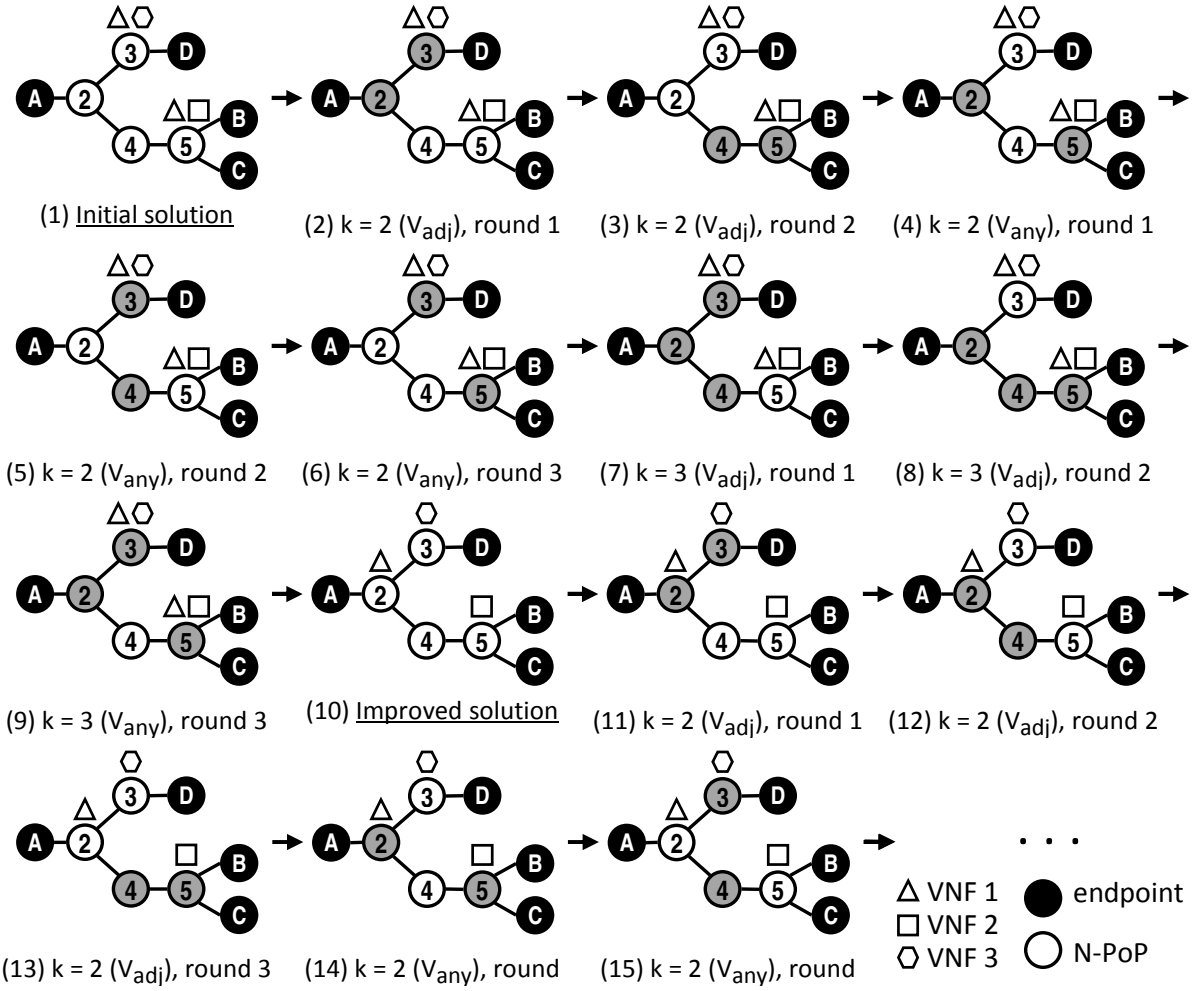
We then iteratively select a subset of N-PoPs, and enumerate the list of variables $y_{i,m,j} \in \mathcal{D} \subseteq Y$ related to them; variables listed in $\mathcal{D}$ will be subject to optimization, while others will

[1]This chapter is based on the following publication:

- Marcelo Caggiani Luizelli, Weverton Luis Cordeiro, Luciana Salete Buriol, Luciano Paschoal Gaspary. **Fix-and-Optimize Approach for Efficient and Large Scale Virtual Network Function Placement and Chaining**. Elsevier Computer Communications, 2017.

remain unchanged (or fixed, hence fix-and-optimize). We take advantage of VNS to systematically build subsets of N-PoPs (Subsection 4.1.4). We also use a prioritization scheme to give preference to those subsets with higher potential for improvement (Subsection 4.1.5).

Figure 4.1: A step-by-step run of our algorithm, for the scenario shown in Figure 1.1.



Source: by author (2016).

For each candidate subset of N-PoPs (processed in order of priority), we submit its decomposed set of variables $\mathcal{D}$ along with χ to a mathematical programming solver (Subsection 4.1.6). Here the goal is to obtain a set of values to those variables listed in $\mathcal{D}$, so that a better configuration is reached. We evaluate configurations according to the objective function of the model. In case there is no improvement, we rollback and pick the N-PoP subset that follows. We run this process iteratively until a better configuration χ' is found. Once it happens, we replace the current configuration with χ', and restart the search process (using χ' as basis). This loop continues until either we have explored the most promising combinations of N-PoP subsets, or a time limit is exceeded.

The loop explained above is illustrated in Figure. 4.1 through instances (1)-(10). Note that, for each instance, a different subset of N-PoPs (determined using VNS) is picked, and the resulting configuration (after optimized by the solver) is evaluated using the model objective function. For example, instance (6) failed as it violated delay constraints, whereas the other instances did not reduce resource commitment. When an improved configuration is found (10), the search process is restarted, now taking as basis the configuration found.

4.1.2 Inputs and Output

In addition to the optimization model input (described in the previous section), our algorithm takes five additional parameters. The first three are 1) a global execution time limit T_{global}, for the algorithm itself, 2) an initial neighborhood size $\mathcal{N}_{init}$, for the VNS meta-heuristic, and 3) an increment step $\mathcal{N}_{inc}$, for increasing the neighborhood size. These parameters are discussed in Subsection 4.1.4. The other parameters are 4) $NoImprov_{max}$, which represents the maximum number of rounds without improvement allowed per neighborhood, and 5) T_{local}, which indicates the maximum amount of time the solver may take for a single optimization run. They are discussed in Subsections 4.1.5 and 4.1.6, respectively.

Our algorithm produces as output a configuration $\chi = \{Y, A^N, A^L\}$ for the VNFPC problem. This configuration may be null, in case a feasible solution does not exist, given the network topology in place and composite services submitted.

4.1.3 Obtaining an Initial Configuration

The first step of our algorithm (line 1) is generating a configuration χ to serve as basis for the fix-and-optimize search. To this end, we remove the objective function (from model introduced in Chapter 3), therefore turning the optimization problem into a factibility one. Then we use a commercial solver (CPLEX[2]) for solving it. The resulting configuration is one that satisfies all constraints, though not necessarily a high quality one, in terms of resources required. Observe that a solution to the VNFPC problem may not exist (line 2). In this case, no solution is returned. Note that other polynomial-time heuristic procedures could be used for generating χ, instead of a solver. We argue, however, that CPLEX is very efficient at finding good initial solutions for this problem. This is because it iteratively applies a set of highly specialized heuristics, based on similarities, to well-know optimization problems.

4.1.4 Variable Neighborhood Search

One important aspect of our algorithm is how to choose which subset of variables $\mathcal{D} \in Y$ will be passed to the solver for optimization. As mentioned earlier, we approach it using VNS.

[2]http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/

It enables organizing the search space in k-neighborhoods. Each neighborhood is determined as a function of the incumbent solution, and a neighborhood size k. In each round, a neighbor (a tuple having k elements) is picked, and used to determine which subset of the incumbent solution will be subject to optimization. In case an improvement is made, VNS uses that improved solution as basis, thus moving to a potentially more promising neighborhood (for achieving further improvements).

We build a neighborhood as a combination (unordered tuple) of any k N-PoPs, with at least one having a VNF assignment (line 5). Each tuple is then decomposed, resulting in the subset of variables $\mathcal{D} \in Y$ that will be subject to optimization (variable decomposition is discussed in Subsection 4.1.6). The reason we focus only on N-PoPs to build these tuples is that determining the assignment of required network functions ($a^N_{i,q,j}$) and chaining allocations ($a^L_{i,j,q,k,l}$) can be done straightforwardly once VNF placements ($y_{i,m,j}$) are defined.

The initial value for k is given as input to the algorithm, $\mathcal{N}_{init}$ (line 3). In the example shown in Figure 4.1, we start with $k = 2$. In each iteration, we explore a neighborhood of size k (line 4) and, if no improvement is made, we increment k in $\mathcal{N}_{inc}$ units (line 21). Observe in Figure 4.1 that, after exploring the 2-neighborhood space (instances (2)-(6)), we make $k \leftarrow k + 1$ and start exploring a neighborhood composed of 3-tuples of N-PoPs (instance (7)). The highest value k may assume is limited by the number of N-PoPs.

On the event of an improvement, we reset the neighborhood size (to $\mathcal{N}_{init}$), and restart the search process. This is illustrated in Figure 4.1, in the transition from instance (10) to (11). It is important to emphasize that same size neighborhoods do not imply in same neighborhoods. For example, consider instances (2) and (11) in Figure 4.1. Although the 2-tuples are composed of the same N-PoPs, each belong to a different 2-neighborhood, as each is associated to a distinct configuration χ. The iteration over neighborhoods, and evaluation of k-tuples within a neighborhood, continues until either T_{global} is exceeded, or k exceeds the number of N-PoPs in the infrastructure (line 4).

4.1.5 Neighborhood Selection and Prioritization

The time required by the solver to process a configuration χ and a subset $\mathcal{D} \subseteq Y$ is often small. However, processing every candidate subset $\mathcal{D}$ from the entire k-neighborhood can be cumbersome. For this reason, we prioritize those neighbor tuples that might lead to a better configuration, or whose processing might be relatively less complex.

Our heuristic for prioritizing tuples in the k-neighborhood set V_k relies on three key observations. First, solving VNF allocations/relocations becomes less complex when N-PoPs are adjacent. In other words, setting values for variables $a^N_{i,q,j}$ and $a^L_{i,j,q,k,l}$ is relatively less complex when N-PoPs i and j are directly connected through a physical path.

To explore the observation above in our algorithm, we break down a k-neighborhood set into two distinct ones. The first one (line 6) is formed by tuples whose participating N-PoPs

form a connected graph. The second set (line 7) is formed by remaining tuples in V_k ($V_{k,any} = V_k \setminus V_{k,adj}$), *i.e.*, those tuples whose participating nodes form a disconnected graph. Observe in Figure 4.1 that, for each k-neighborhood, we first process the tuples of adjacent N-PoPs (V_{adj}): instances (2)-(3); (7)-(8); and (11)-(13)). Then, we process the remainder tuples (V_{any}): instances (4)-(6); (9); and (14)-(15).

The second key observation is that N-PoPs having a higher number of VNFs allocated, but fewer SFC requests assigned, are more likely candidates for optimization. Examples of such optimization are the removal of (some) VNFs allocated to those N-PoPs, or merge of those VNFs in a single N-PoP. We explore this observation by establishing a tuple priority, as a function of its residual capacity, and processing higher priority tuples first (line 11). The residual capacity of a tuple v is given by $r : V_k \rightarrow \mathbb{R}$, defined as a ratio of assigned VNFs to placed ones (according to Eq. 4.1).

$$r(v) = \exp\left(\sum_{i \in N^P} \sum_{q \in Q} \sum_{j \in N_q^S} a_{i,q,j}^N - \sum_{i \in N^P} \sum_{m \in F} \sum_{j \in U_m} y_{i,m,j}\right) \tag{4.1}$$

The third observation is a complement of the previous one. As the priority of a neighbor decreases, it becomes less likely that it will lead to any optimization at all. The reason is trivial: lower priority tuples are often composed of overcommitted N-PoPs, for which removal/relocation of allocated/assigned VNFs is more unlikely. Therefore, when processing a neighborhood, we can skip a certain fraction of low priority neighbors, with a high confidence that no optimization opportunity will be missed.

To this end, our algorithm takes as input $\textit{NoImprov}_{max}$. It indicates the maximum number of rounds without improvement that is allowed over a neighborhood subset. Before processing a given subset, we reset the counter of rounds without improvements, *NoImprov* (line 9). This counter is incremented for every round in which no improvement is made (lines 14 - 17). We stop processing the current neighborhood subset once *NoImprov* exceeds $\textit{NoImprov}_{max}$ (line 10).

4.1.6 Configuration Decomposition and Optimization

Decomposing a configuration (*i.e.*, building a subset of variables $\mathcal{D} \subseteq Y$ from χ for optimization) means enumerating all $y_{i,m,j}$ variables that can be optimized (line 12). It basically consists of listing all $y_{i,m,j}$ variables related to each N-PoP in the current neighbor tuple v, considering all network function types $m \in F$, and function instances available $j \in U_m$. Formally, we have $\mathcal{D} = \{\, y_{i,m,j} \in Y \mid i \in v \,\}$.

Once the decomposition is done, the incumbent configuration χ and set $\mathcal{D}$ are submitted to the solver (line 13). As mentioned earlier, the solver will consider variables listed in $\mathcal{D}$ as free

(for optimization), and those not listed as fixed (*i.e.*, no change in their values can be made). Observe that this restriction does not affect those variables related to the assignment of required network functions ($a^N_{i,q,j}$) and to chaining allocation ($a^L_{i,j,q,k,l}$).

We also limit each optimization run to a certain amount of time T_{local} (passed as parameter). This enables us to allocating a significant amount of the global time limit T_{global} on fewer but extremely complex χ and $\mathcal{D}$ instances.

Algorithm 2 Overview of the fix-and-optimize heuristic for the VNF placement and chaining problem.

Input: T_{global} global time limit
T_{local} time limit for each solver run
$\mathcal{N}_{init}$ initial neighborhood size
$\mathcal{N}_{inc}$ increments for neighborhood size
NoImprov$_{max}$ max. rounds without improvement
Output: χ: best solution found to the optimization model

$\chi \leftarrow$ initial feasible configuration
if a feasible configuration does not exist **then** fail **else**
  $k \leftarrow \mathcal{N}_{init}$
  while T_{global} is not exceeded **and** $k \leq |N^P|$ **do**
    $V_k \leftarrow$ current neighborhood, formed of unordered tuples of k nodes only, one of them (at least) having a VNF assignment in the incumbent configuration χ
    $V_{k,adj} \leftarrow$ neighbor tuples from V_k, whose nodes (for each tuple) form a connected graph
    $V_{k,any} \leftarrow V_k \setminus V_{k,adj}$
    for each list $\mathcal{V} \in \{V_{k,adj}, V_{k,any}\}$, while a better configuration is not found **do**
      NoImprov $\leftarrow 0$
      while $\mathcal{V} \neq \emptyset$ **and** *NoImprov* $\leq$ *NoImprov*$_{max}$ **and** better configuration is not found **do**
        $v \leftarrow$ next unvisited, highest priority neighbor tuple from $\mathcal{V}$
        $\mathcal{D} \leftarrow$ decomposed list of variables $y_{i,m,u}$ from those nodes listed in neighbor tuple v
        $\chi' \leftarrow$ configuration χ optimized by the solver, performed under time limit T_{local}, and making those variables not listed in $\mathcal{D}$ as fixed
        if χ' is a better configuration than χ **then**
          update χ to reflect configuration χ'
        else
          NoImprov $\leftarrow$ *NoImprov* + 1
        end if
      end while
    end for
    if no improvement was made **then** $k \leftarrow k + \mathcal{N}_{inc}$
    else $k \leftarrow \mathcal{N}_{init}$ **end if**
  end while
  return χ
end if

4.2 Evaluation

In this section we evaluate the effectiveness of our math-heuristic algorithm in generating feasible solutions to the VNFPC problem. We compare the solutions achieved with those of Lewin-Eytan *et al.* (LEWIN-EYTAN et al., 2015), Luizelli *et al.* (LUIZELLI et al., 2015), and with a globally optimal one obtained with CPLEX (whenever they are feasible to compute). We also establish a lower bound for the VNFPC optimization, computed by relaxing output variables of the mathematical model and making them linear (instead of discrete). Observe that such lower bound becomes a solid reference for what the optimal solution to the VNFPC problem looks like, in a given large scenario.

We used CPLEX v12.4 for solving the optimization models, and Java for implementing the algorithms. The experiments were run on Windows Azure Platform – more specifically, an Ubuntu 14.04 LTS VM instance, featuring an Intel Xeon processor E5 v3 family, with 32 cores, 448 GB of RAM, and 6 TB of SSD storage. Next we describe our experiment setup, followed by results obtained.

4.2.1 Setup

The physical network substrate was generated with Brite[3], following the Barabasi-Albert (BA-2) (ALBERT; BARABÁSI, 2000) model. The number of N-PoPs and physical paths between them varied in each scenario, ranging from 200 to 1,000 N-PoPs, and from 793 to 3,993 links. The computing power of each N-PoP is normalized and set to 1, and each link has bandwidth capacity of 10 Gbps. Average delay between any pair of N-PoPs is 90 ms, value adopted after a study from Choi *et al.* (CHOI et al., 2007).

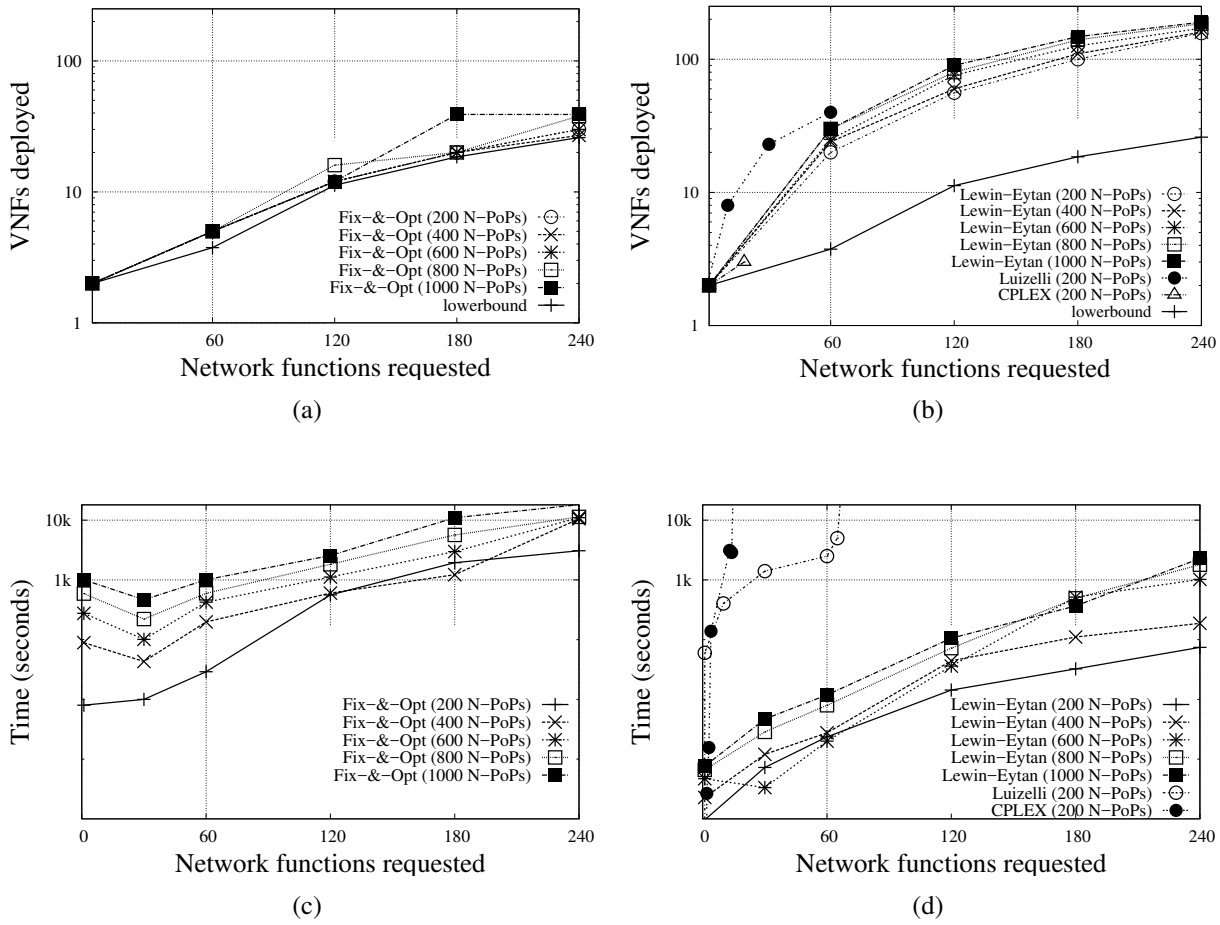
In order to fully grasp the efficiency and effectiveness of our approach, we carried out experiments considering a wide variety of scenarios and parameter settings. For the sake of space constraints, we concentrate on a subset of them. Our workload, for example, comprised from 1 to 80 SFCs. The topology of each SFC consisted of: a source endpoint and then a link (supporting 1 Gbps of traffic flow) to an instance of VNF X; a flow branch follows VNF X, each branch with 500 Mbps output flow, linking to a distinct instance of VNF Y; finally, each flow links to a distinct sink endpoint. Instances of VNF X require a normalized computing power capacity of 0.125 for small loads, and 1 for large loads, and have a processing delay of 0.6475 seconds. Instances of VNF Y require, in turn, 0.125 and 0.25 for small and large loads, and impose a processing delay of 7.0771 seconds. VNF processing delays were determined after a study from Dobrescu *et al.* (DOBRESCU; ARGYRAKI; RATNASAMY, 2012).

In the subsections that follow, we adopted the following parameter setting for our approach: $T_{global} = 5,000$ seconds, $T_{local} = 200$ seconds, $\mathcal{N}_{init} = 2$, $\mathcal{N}_{inc} = 1$, and $\mathit{NoImprov}_{max} = 15$. In Subsection 4.2.4 we present an analysis of our approach considering other parameter settings.

[3] http://www.cs.bu.edu/brite/

4.2.2 Our Heuristic Algorithm Compared to Existing Approaches

Figure 4.2: Number of deployed VNFs and required time to compute a feasible placement and chaining.



Source: by author (2016).

Figure 4.2 provides an overview of achieved results, focusing on resource commitment of computed solutions, and time required to generate them. The main takeaway here is that our approach is able to generate significantly better solutions (closer to the lower bound) in a reasonable time, compared to existing ones. Observe in Figure 4.2(a), for example, that our approach generated a solution distant at most 2.1 times from the lower bound, in a more extreme case with 1,000 N-PoPs and 180 requested functions. This is a significantly smaller gap compared to other approaches, even considering their best cases. The measured distance was 4.97 times for Lewin-Eytan *et al.* (LEWIN-EYTAN et al., 2015), in the scenario with 200 N-PoPs and 120 requested functions, whereas for Luizelli *et al.* (LUIZELLI et al., 2015) it was 6.7 times, considering 200 N-PoPs and 60 functions.

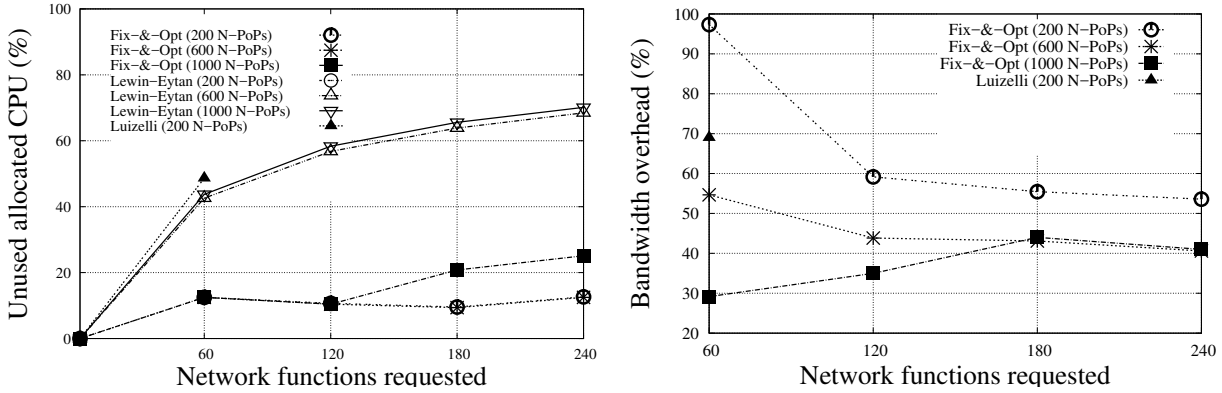
Observe that finding an optimum solution (with CPLEX) is unfeasible even for very small

instances (200 N-PoPs and less than 20 requested functions). Luizelli *et al.* (LUIZELLI et al., 2015) also failed short in scaling to such small instances, requiring more than 10k seconds for those around 60 requested functions.

An important caveat regarding the time required to obtain a solution is that, for our approach, we measured the time elapsed until it generated the highest quality solution. Note however that our algorithm is allowed to continue running, until T_{global} is passed or all neighborhoods are explored.

4.2.3 Qualitative Analysis of Generated Solutions

Figure 4.3: Analysis of resource commitment (computing power and bandwidth) of each solution generated.



Source: by author (2016).

Here we dive deeper on measuring the quality of generated solutions, analyzing their over-provisioning with regard to computing power and bandwidth. Observe from Figure 4.3 that our proposal led to comparatively higher quality solutions, minimizing the amount of allocated but unused computing power in N-PoPs. By "unused" we mean that resources were allocated to VNFs deployed on N-PoPs, but the capacity of the VNFs is not fully consumed by assigned SFCs.

The performance of Lewin-Eytan *et al.* (LEWIN-EYTAN et al., 2015) is worse in this aspect mostly because their goal is not only minimize resource commitment, but also the distance (hops) a flow must traverse to reach a required VNF. It is also important to emphasize that their theoretical model has simplifications, which result in solutions that often extrapolate resource commitment. Since the authors did not approach VNF chaining in their model, bandwidth overhead could not be measured for their case.

Table 4.1: Assessment of the quality of generated solutions, and distance to lower bound, under various parameter settings.

	$\mathcal{N}_{init}$	T_{local} 50	factor	60	factor	100	factor
T_{global} = 600 sec.	2	59	5.24	51	4.53	51	4.53
	4	59	5.24	59	5.24	42	3.73
	6	59	5.24	59	5.24	33	2.93
	$\mathcal{N}_{init}$	T_{local} 50	factor	60	factor	100	factor
T_{global} = 1,000 sec.	2	32	2.84	16	1.42	15	1.33
	4	44	3.91	12	1.06	12	1.06
	6	40	3.55	18	1.60	12	1.06
	$\mathcal{N}_{init}$	T_{local} 50	factor	60	factor	100	factor
T_{global} = 1,500 sec.	2	20	1.77	12	1.06	12	1.06
	4	36	3.20	14	1.24	12	1.06
	6	36	3.20	15	1.33	12	1.06

Source: by author (2016).

4.2.4 Sensitivity Analysis

We conclude our analysis with a glimpse on experiments varying input parameter settings, whose results are summarized in Table 4.1. The cells in gray show how far our approach is from the lower bound (in terms of resources allocated), in a scenario with 600 N-PoPs and 120 functions. We focused on the following values for T_{global}: 600, 1,000, and 1,500 secs.; for T_{local}: 50, 60, and 100 secs.; and for $\mathcal{N}_{init}$: 2, 4, and 6.

Observe in Table 4.1 that our approach was at most 5.24 times far from the lower bound, having allocated 59 VNFs (compared to a lower bound of 11.75 VNFs). More importantly, observe that such distance decreases as we provide more execution time for both the algorithm itself (T_{global}) and for each optimization instance (T_{local}). In the best case scenario, our approach was only 1.06 times distant from the lower bound, allocating the least number of VNFs (12).

Observe also a trade-off when setting T_{local}. On the one hand, larger values enable solving each sub-problem instance with higher quality, as one may note in Table 4.1. On the other hand, smaller values enable avoiding more complex sub-problem instances, for which CPLEX requires far more RAM memory to solve. It also leaves the algorithm with more global time available for exploring other promising sub-problem instances.

5 THE ACTUAL COST OF SOFTWARE SWITCHING FOR NFV CHAINING

In Chapters 3 and 4, we focused on the Inter-datacenter VNFPC problem. We mathematically formalized it and designed efficient and scalable solutions so as large NFV distributed infrastructures could be timely handled. As previously discussed, a solution to the Inter-datacenter VNFPC problem involves the placement and chaining of SFC requests into multiple locations. This is particularly important to SFC requests with stringent network requirements (*e.g*, very low end-to-end delays). In this case, SFCs are carefully broken down into sub-chains (*i.e.*, subgraphs) individually placed and chained into datacenters. Then, these partial SFC requests are deployed on top of commodity servers. Up to this point, we have focused on planning and chaining SFCs in distributed NFV environments without any insight on how VNFs are deployed (onto servers) and their incurred performance limitations.

For that reason, we take a step further in the resolution of this problem by specifically tackling the intra-datacenter VNFPC problem[1]. In this chapter, we focus on paving the way towards cost-efficiently intra-datacenter deployments. As NFV is yet a premature research area, little has been done in order to assess performance limitations of real NFV-based deployments and the incurred operational costs of commodity servers.

To fill in this gap, we conduct an extensive and in-depth evaluation, measuring the performance and analyzing the impact of deploying service chains on NFV servers with *Open vSwitch* (PFAFF et al., 2015). We analyze different intra-datacenter placement strategies and assess performance metrics such as throughput, packet processing, and resource consumption. Based on the performance evaluation, we then define a generalized abstract cost function that accurately captures the CPU cost of network switching. Our proposed model can predict the operational cost incurred by different deployment strategies given arbitrary requirements.

The remainder of this chapter is organized as follows. In Section 5.1, we start describing the intra-datacenter VNFPC problem. Then, in Section 5.2, we formalize the model used throughout this chapter. In Section 5.3, we provide an extensive evaluation of service chain deployments on a real NFV environment. Last, in Section 5.4 we define an abstract CPU-cost function that estimates operational costs of NFV deployments.

5.1 Problem Overview

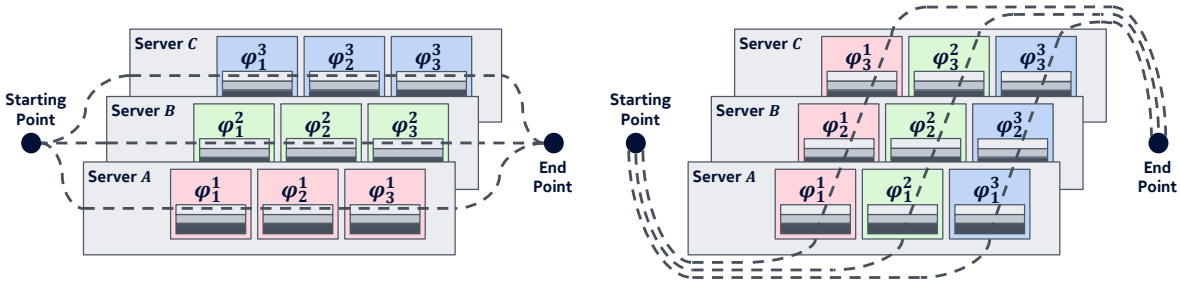
Placement of service chains in NFV-based infrastructure introduces new challenges that are not trivial and need to be addressed. A key factor in the long-term success of NFV is the optimization of the operational costs (OPEX) when deploying network services. On deploying

[1]This chapter is based on the following publication:

- Marcelo Caggiani Luizelli, Danny Raz, Yaniv Saar and Jose Yallouz. **The Actual Cost of Software Switching for NFV Chaining.** In: Proceedings of IFIP/IEEE International Symposium on Integrated Network Management, 2017.

service chains onto physical servers (*i.e.* intra-datacenter VNFPC), network services might face performance penalties and limitations on critical network metrics – such as throughput, latency, and jitter – depending on how network functions are chained and deployed. This is one of the most interesting challenges for network providers in the shift to NFV, namely, to identify *good* placement strategies that minimize the provisioning and operation cost of service chains.

Figure 5.1: Example of strategies to deploy three given Service Function Chaining.



(a) Gather placement strategy.

(b) Distribute placement strategy.

Source: by author (2017).

However, operational costs of NFV are still far from being straightforward estimated even for relatively simple deployment strategies. In the context of practical NFV deployments, *OpenStack* (ONF, 2015) is the most popular open-source orchestration system (and part of OPNFV project) and its scheduling/placement mechanisms can be used for the placement of service chains. The current available placement mechanism follows two deployment strategies (or objective functions): load balancing (*i.e.* distributing VNFs between resources) or energy conserving (*i.e.* gathering VNFs on a selected resource).

Figure 5.1 illustrates these two placement strategies. In this example, we are given three identical servers A, B and C, and three identical service chains $\{\varphi^1, \varphi^2, \varphi^3\}$. Each service chain φ^i (for $i = 1, 2, 3$) is tagged with a fixed amount of required traffic and is composed of three chained VNFs: $\varphi^i = \langle \varphi^i_1 \rightarrow \varphi^i_2 \rightarrow \varphi^i_3 \rangle$. Figure 5.1(a) depicts the first placement strategy (referred to as "gather"), where all VNFs composing a single chain are deployed on the same server. Note that in the gather case the majority of traffic steering is done by the server's internal virtual switching. Figure 5.1(b) illustrates the second placement strategy (referred to as "distribute"), where each VNF is deployed on a different server and, therefore, the majority of traffic steering is done between servers by external switches.

One can immediately see that these two placement strategies require from the VNF perspective the same amount of resources to operate. However, it is not clear the operational costs from the provider's perspective involved in each deployment – especially regarding the cost of their software switching (CPU consumption). Additionally, network metrics (*e.g.* throughput and latency) might perform differently according to each deployment. In order to decide which placement strategy is *better*, we need to identify the specific optimization criteria of interest. In

this chapter, we take the first steps towards accurately estimating the virtual switching cost – which ultimately represents the operational cost.

As previously discussed, virtual switching is an essential building block in NFV-based infrastructure which enables flexible communication between VNFs; however, it introduces costs in terms of resource utilization in the infrastructure. Therefore, measuring and modeling these costs represents an essential step to: *(i)* efficiently guide VNF placement in a physical infrastructure; *(ii)* understand how software switching impacts deployed network services regarding network metrics.

5.2 Model Definition

In this section, we define our model to estimate the cost of software switching for different placement strategies.

For a server S we denote by S^t the maximum throughput that server S can support (*i.e.*, the wire limit of the installed NIC). Following the recommended best practice for virtualization intense environment ((INTEL, 2016b)), NFV servers require to be configured with two disjoint sets of CPU-cores, one for the hypervisor and the other for the VNF to operate. We denote by S^h (and S^v) the number of CPU-cores that are allocated and reserved for the hypervisor (and guests, respectively) to operate. The total number of CPU-cores installed in server S is denoted by S^c. Throughout this chapter, unless explicitly saying otherwise, we assume that we are given a set of k servers $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$.

A *service chain* is defined to be an ordered set of VNFs $\varphi = \langle \varphi_1 \rightarrow \varphi_2 \ldots \rightarrow \varphi_n \rangle$. We denote by $\varphi|^n$ the length of the service chain, and define $\varphi|^p$ (and $\varphi|^s$) to be the number of packets per second (and average packet size, respectively) that service chain φ is required to process. Note that $\varphi|^p$ and $\varphi|^s$ are properties of service chain φ, namely given two service chains φ and ψ if $\varphi|^p \neq \psi|^p$ or $\varphi|^s \neq \psi|^s$ then $\varphi \neq \psi$. Throughout the paper, unless explicitly saying otherwise, we assume that we are given a set of m identical service chains $\Phi = \{\varphi^1, \varphi^2, \varphi^3, \ldots \varphi^m\}$, *i.e.*, $\forall_{i,j=1..m} : \varphi^i = \varphi^j$.

We define $\mathcal{P} : \Phi \rightarrow S^k$ to be a *placement function* that for every service chain $\varphi \in \Phi$, maps every VNF $\varphi_i \in \varphi$ to a server $S_j \in \mathcal{S}$. We identify two particularly placement functions:

1. $\mathcal{P}_g$ – we call *gather* placement, where for every service chain the placement function deploys all VNFs $\varphi_i \in \varphi$ on the same server, namely:

$$\forall_{\varphi \in \Phi} \forall_{\varphi_i, \varphi_j \in \varphi} : \quad \mathcal{P}_g(\varphi_i) = \mathcal{P}_g(\varphi_j)$$

2. $\mathcal{P}_d$ – we call *distribute* placement, where for every service chain the placement function deploys each VNFs $\varphi_i \in \varphi$ on a different server, namely:

$$\forall_{\varphi \in \Phi} \forall_{\varphi_i, \varphi_j \in \varphi} : \quad i \neq j \rightarrow \mathcal{P}_d(\varphi_i) \neq \mathcal{P}_d(\varphi_j)$$

As explained before, both considered placement functions follow OpenStack and the objective functions implemented by its scheduler mechanism (*i.e.* load balancing and energy conserving). Figure 5.1 illustrates these deployment strategies. Figure 5.1(a) depicts deployment of VNFs that follows the gather placement functions $\mathcal{P}_g$, and Figure 5.1(b) depicts deployment of VNFs that follows the distribute placement functions $\mathcal{P}_d$.

For a given placement function $\mathcal{P}$ that deploys all service chains in Φ on the set of servers $\mathcal{S}$, we define $\mathcal{C} : \mathcal{P} \rightarrow (\mathbb{R}^+)^k$ to be a *cpu-cost functions* that maps placement $\mathcal{P}$ to the required CPU per server. We say the cpu-cost function is *feasible* if there are enough resources to implement it. Namely, cpu-cost function $\mathcal{C}$ is feasible with respect to a given placement $\mathcal{P}$ if for every server $S_j \in \mathcal{S}$, the function does not exceed the number of CPU-cores installed in the server: $\forall_{S_j \in \mathcal{S}} : \mathcal{C}(\mathcal{P})_j \leq S_j^c$. Note that our main focus is the evaluation of the cost-functions with respect to the hypervisor performance, and therefore we evaluate deployments of VNFs that are fixed to do the minimum required processing, *i.e.* forward the traffic from one virtual interface to another.

5.3 Deployment Evaluation

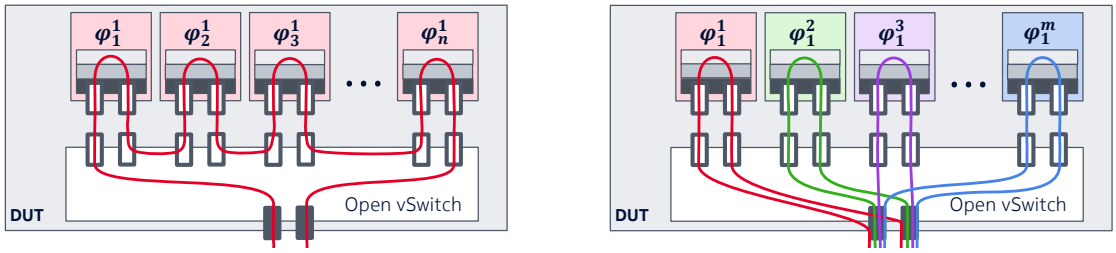
In this section, we evaluate the performance of software switching for NFV chaining considering metrics such as throughput, CPU utilization, and packet processing. We describe our environment setup, followed by a discussion on performance metrics and possible bottlenecks. We evaluate two types of OvS installations: Linux kernel and DPDK; and compare between the two types of VNF placements: gather placement function $\mathcal{P}_g$, versus distribute placement function $\mathcal{P}_d$.

5.3.1 Setup

Our environment setup consists of two high-end HP ProLiant DL380p Gen8 servers, each server has two Intel Xeon E5-2697v2 processors, and each processor is made of 12 physical cores at 2.7 Ghz. One server is our *Device Under Test* (DUT), and the other is our traffic generator. The servers have two NUMA nodes (Non-Uniform Memory Access), each has 192 GBytes RAM (total of 384 GBytes), and an Intel 82599ES 10 Gbit/s network device with two network interfaces (physical ports). We disabled HyperThreading in our servers in order to have more control over core utilization.

In both types of OvS installations (Linux kernel and DPDK), we isolate CPU-core and separate between two disjoint sets CPU-cores: $S^c = 24 = S^h + S^v$, i.e. CPU-cores used for management and for allocating resources for the VNFs to operate. This a priori separation between these two disjoint sets of CPU-cores plays an important role in the behavior of CPU consumption (and packet processing). In our experiments the size of the disjoint set of cores that are given to the hypervisor (S^h) varies between 2 to 12 physical cores, while the remainder

Figure 5.2: Given a set of m service chains Φ, illustrating deployment of VNFs on a single server (our DUT).



(a) Following gather placement function $\mathcal{P}_g$. (b) Following distribute placement function $\mathcal{P}_d$.

Source: by author(2017).

is given to the VNFs (i.e. S^v ranges between 22 to 12). The exact values depend on the type of installation (Linux kernel or DPDK).

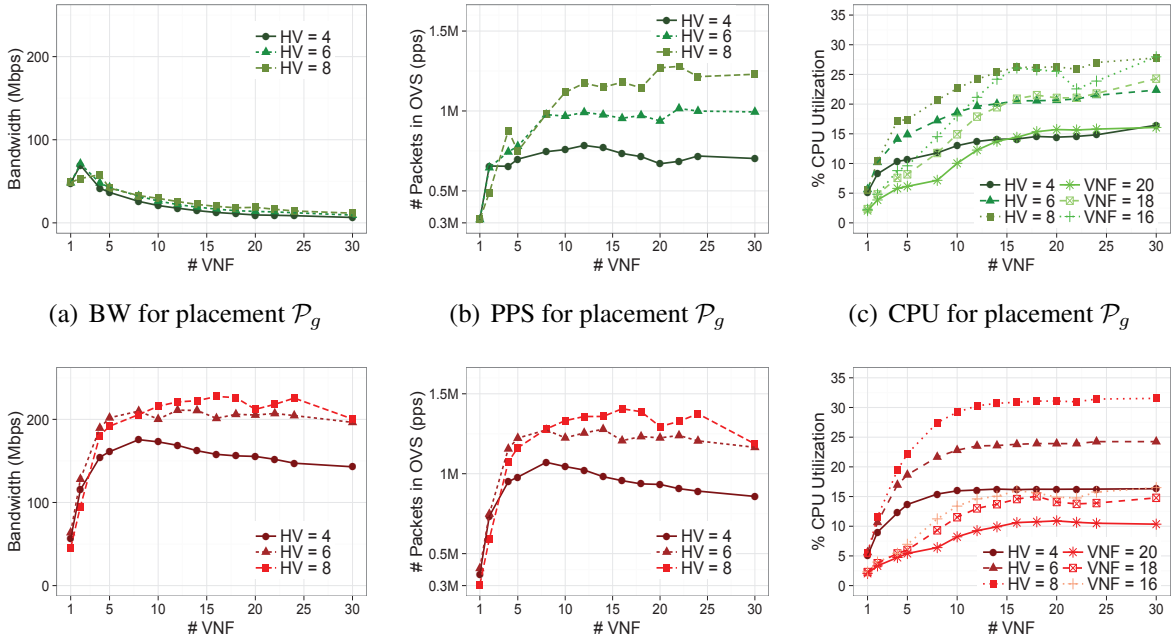
The DUT is installed with CentOS version 7.1.1503, Linux kernel version 3.10.0. All guests operating system are installed with Fedora 22, Linux kernel version 4.0.4 – running on top of qemu version 2.5.50. Each VNF is configured to have a single virtual CPU pinned to a specific physical core and 4GBytes of RAM. Network offload optimizations (e.g., `TSO`, `GSO`) in all network interfaces are disabled in order to avoid any noise in the analysis and provide a fair comparison between the two types of OvS installations. We evaluated Open vSwitch version 2.4 in both kernel mode and DPDK. For the latter, we compiled OvS against DPDK version 2.2.0 (without shared memory mechanisms – in compliance with rigid security requirements imposed by NFV). Packet generation is performed on a dedicated server that is interconnected to the DUT server with one network interface to send data, and another network interface to receive. In all experiments, we generate the traffic with Sockperf version 2.7 (CORPORATION, 2016), that invokes 50 TCP flows.

Note that our objective is to provide an analytic model that captures the cost of software switching. In order to be able to examine and provide a clear understanding of the parameters that impact the cost of software switching, we need to simplify our environment by removing optimizations such as network offloads, and fixing resource allocation. For achieving optimal performance, the reader is referred to (INTEL, 2016b).

We evaluate the placement functions defined in Section 6.1 (i.e., the gather placement function $\mathcal{P}_g$ and the distribute placement function $\mathcal{P}_d$) on our DUT as follows. We vary the number of simultaneously deployed VNFs from 1 to 30 in each of the placement functions. All VNFs forward their traffic between two virtual interfaces, each on a different sub-domains, relaying on IP route forwarding.

Figure 5.2 illustrates deployments of VNFs on a single server (our DUT). Figure 5.2(a) depicts the traffic flow of the gather placement function $\mathcal{P}_g$. Ingress traffic is arriving from the physical NIC to the OvS, which forwards it to the first vNIC of the first VNF. The VNF then

Figure 5.3: Experiment results showing throughput, packet processing and CPU consumption for traffic generated in 100Bytes packets, that is examined over increasing size of chain, on a DUT that is installed with kernel OvS.



(a) BW for placement $\mathcal{P}_g$ (b) PPS for placement $\mathcal{P}_g$ (c) CPU for placement $\mathcal{P}_g$

(d) BW for placement $\mathcal{P}_d$ (e) PPS for placement $\mathcal{P}_d$ (f) CPU for placement $\mathcal{P}_d$

Source: by author (2017).

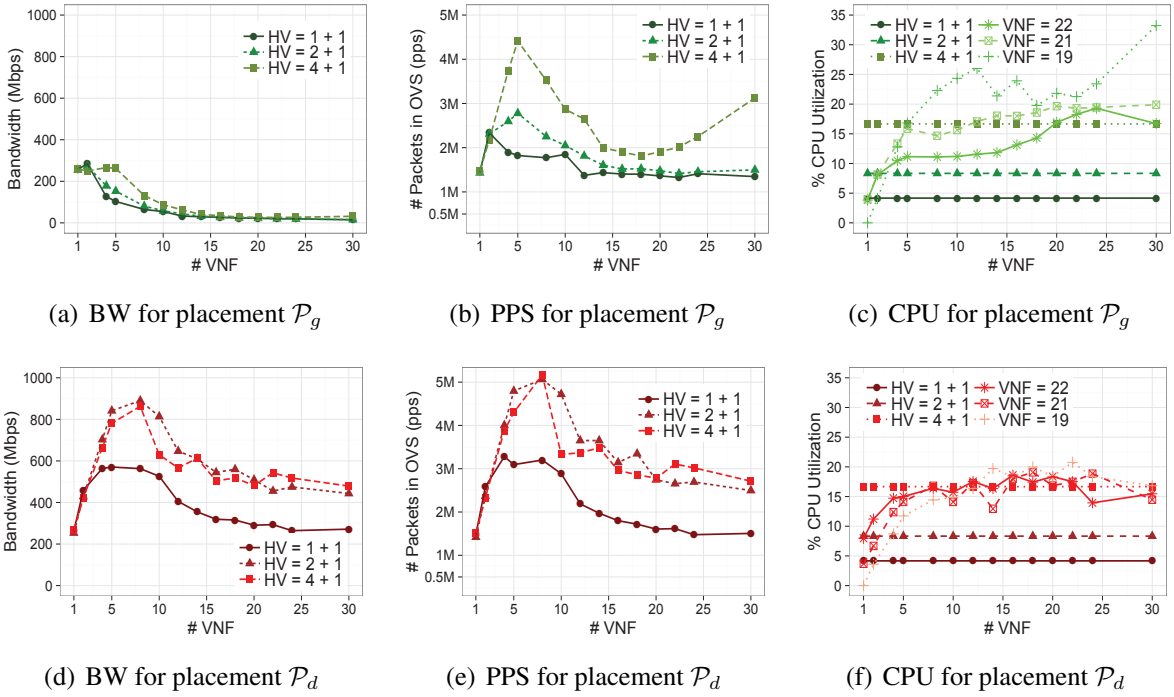
returns the traffic to the OvS through its second vNIC, and the OvS forwards the traffic to the following VNF, composing a chain that eventually egresses the traffic through the second physical NIC. On the other hand, Figure 5.2(b) depicts the traffic flow of the distribute placement function $\mathcal{P}_d$. For this placement, it is the responsibility of the traffic generator to spread the workload between the VNFs. Ingress traffic arriving from the physical NIC to the OvS that forwards it to one of the VNFs through its first vNIC. The VNF then returns the traffic to the OvS through its second vNIC, that egress the traffic through the second physical NIC.

5.3.2 Evaluating Packet Intense Traffic

In order to discuss the utilization of resources, we measure and analyze the results of our DUT for several configurations, while receiving intense network workload. To generate an intense network workload we generate traffic where each packet is composed of 100Bytes (avoid reaching the NIC's wire limitation). We examine the behavior of throughput, packet processing and CPU consumption for increasing chain size, using both placement functions, i.e. $\mathcal{P}_g$ and $\mathcal{P}_d$.

The six graphs in Figure 5.3 (and Figure 5.4) depict CPU consumption, packet processing, and throughput performance on a DUT that is installed with kernel OvS (and DPDK-OvS,

Figure 5.4: Experiment results showing throughput, packet processing and CPU consumption for traffic generated in 100Bytes packets, that is examined over increasing size of chain, on a DUT that is installed with DPDK-OvS.



(a) BW for placement $\mathcal{P}_g$ (b) PPS for placement $\mathcal{P}_g$ (c) CPU for placement $\mathcal{P}_g$

(d) BW for placement $\mathcal{P}_d$ (e) PPS for placement $\mathcal{P}_d$ (f) CPU for placement $\mathcal{P}_d$

Source: by author (2017).

respectively). The three graphs on the top (Figures 5.3(a), 5.3(b), and 5.3(c) for kernel OvS and Figures 5.4(a), 5.4(b), and 5.4(c) for DPDK-OvS) present the results following the gather placement function $\mathcal{P}_g$ and the three graphs on the bottom (Figures 5.3(d), 5.3(e), and 5.3(f) for kernel OvS and Figures 5.4(d), 5.4(e), and 5.4(f) for DPDK-OvS) present the results following the distribute placement function $\mathcal{P}_d$.

We measure throughput by aiming at the total of traffic that the traffic generator successfully sends and receives after routing through the DUT. To measure packet processing we aggregate the number of received packets (including TCP acknowledgments) in all interfaces of the OvS (including both NICs and vNICs). For CPU consumption, we present the results for both the CPU-cores allocated to the hypervisor to manage and support the VNF and CPU-cores allocated for the VNFs to operate.

5.3.2.1 Packet Processing and Throughput

A key factor in the behavior of our environment is the a priori separation between two disjoint sets of CPU-cores (S^h and S^v). Thus, in our experiments, we vary values of CPU-cores that are allocated to the OvS (hypervisor). For ease of presentation, we show only several selected values (annotated HV in the figures).

Figure 5.3(a) depicts the average throughput for gather placement function $\mathcal{P}_g$, and Figure 5.3(d) for distribute placement function $\mathcal{P}_d$. For the case of distribute placement function $\mathcal{P}_d$, the more VNFs we deploy on the server, the more the average throughput is increased, while for the gather placement function $\mathcal{P}_g$ the more the average throughput is decreased. Figures 5.3(b) and 5.3(c) depict packet per second for both our placement function $\mathcal{P}_g$ and $\mathcal{P}_d$. The results show that OvS can seamlessly scale to support 5-10 VNF, however at that point OvS reaches saturation, and we observe mild degradation when continuing to increase the number of deployed VNFs.

Figures 5.4(a) and 5.4(d) (and Figures 5.4(b) and 5.4(c)) depict average throughput (and packet per second, respectively) of our two placement function $\mathcal{P}_g$ and $\mathcal{P}_d$, on a DUT that is installed with OvS-DPDK. The behavior of OvS-DPDK is similar to that of the kernel OvS, with the exception of having better network performance.

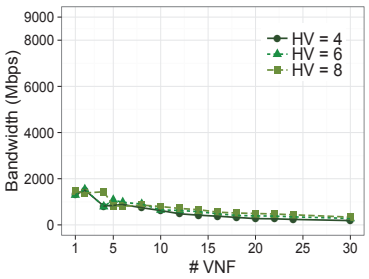
5.3.2.2 CPU Consumption

Figures 5.3(c) and 5.3(f) depict the average CPU consumption of OvS in Kernel mode. For both placement functions, the CPU consumption of the VNFs is bounded by the number of allocated cores for management S^h. A tighter bound of the CPU consumed by the VNFs (for networking) is a function of the CPU consumed by the hypervisor to manage the traffic, namely the total CPU consumed by the VNF is bounded by the total CPU consumed by the hypervisor in order to steer the traffic to the VNF.

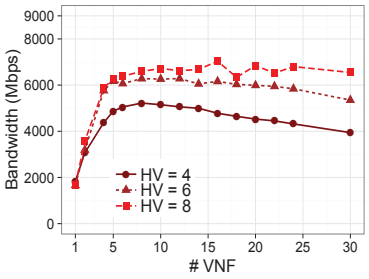
Comparing CPU utilization between the two placement functions, we observe that both placement strategies behave similarly for short service chains with little traffic. However, for longer service chains with increasing traffic requirements the behavior of the two placements differs. Namely, in the gather placement, the CPU consumed by the VNF is almost identical to the CPU consumed by the hypervisor, whereas in the distribute placement the CPU consumed by the VNFs are 70% to 50% of the CPU consumed by the hypervisor. This observation suggests that in order to achieve efficiently CPU cost placement, different traffic might require different placement strategies (as will be shown in the results of our analytical analysis in Section 5.4).

Figures 5.4(c) and 5.4(f) depict the average CPU consumption of DPDK-OvS. For both placement functions, the CPU consumed by the hypervisor is fixed and derived from the DPDK poll-mode driver (with the additional CPU-core for the management of other non-networking provision tasks). Comparing CPU utilization between the two placement functions shows again that both behave the same for small chains with little traffic, however for bigger chains with increasing traffic requirements the behaviour of the two placements is different.

Figure 5.5: Throughput for traffic generated in 1500Bytes packets, that is examined over increasing size of chain, on a DUT that is installed with kernel-OvS.

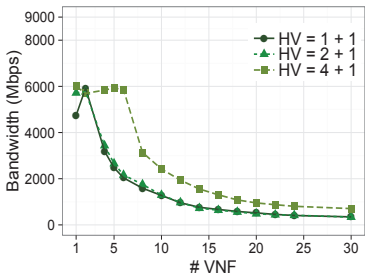


(a) BW for placement $\mathcal{P}_g$

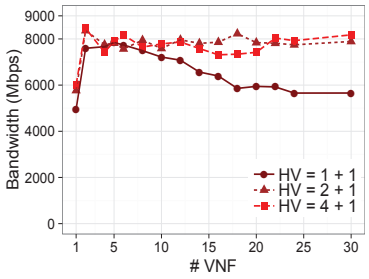


(b) BW for placement $\mathcal{P}_d$

Figure 5.6: Throughput for traffic generated in 1500Bytes packets, that is examined over increasing size of chain, on a DUT that is installed with DPDK-OvS.

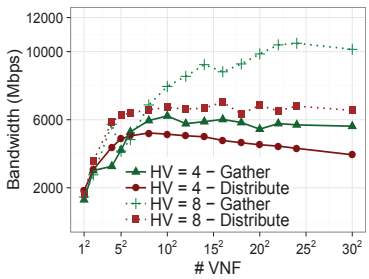


(a) BW for placement $\mathcal{P}_g$

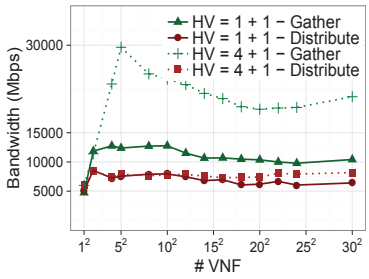


(b) BW for placement $\mathcal{P}_d$

Figure 5.7: Analysis of multiple servers, showing total throughput for traffic generated in 1500Bytes packets, that is examined over increasing size of chain.



(a) Kernel OvS



(b) DPDK-OvS

Source: by author (2017).

5.3.3 Evaluating Throughput Intense Traffic

We reiterate the experiment presented in Subsection 5.3.2 in the context of achieving maximum throughput. Note that as opposed to the previous section where we wanted to examine the CPU cost of the transferred traffic, here our goal is solely to maximize our throughput. In order to discuss maximum throughput, we measure and analyze the results of our DUT for several configurations, while receiving maximum transferable unit, i.e. we generate traffic where each packet is composed of 1500Bytes. We examine the behavior of throughput for increasing chain size, using both placement functions $\mathcal{P}_g$ and $\mathcal{P}_d$. Since the behavior of CPU consumption, and packet processing are similar to the behavior observed for 100Bytes packet (in bigger scale), we omit their presentation.

Figure 5.5 (and Figure 5.6) depicts throughput performance on a DUT that is installed with kernel OvS (DPDK-OvS, respectively). Both Figures 5.5(b) for kernel-OvS, and 5.6(b) for DPDK-OvS show that the average throughput can scale up as long as the server is not over-

provisioning resources (when deploying 1-5 VNFs). For the case of kernel OvS the bottleneck is the packet processing limit, while the NIC's wire limit is the bottleneck for the case of DPDK-OvS. The same effect can also be seen in Figures 5.5(a) and 5.6(a) where again as long as the server is not over-provisioning resources, the chain of VNFs is able to forward ∼0.8-1.5 Gbit/s in the case of kernel OvS, and ∼6 Gbit/s in the case of DPDK-OvS. Note that DPDK-OvS does not reach its packet processing limit (as it was seen in the case of 100-Byte packets – Subsection 5.3.2). Instead, the observed limit is induced by the amount of packet processing that a single CPU core (allocated for the VNF) can perform (∼6 Gbit/s). This bottleneck can be mitigated if VNFs are set to have more than a single vCPU – that are configured to enable Receive Side Scaling (RSS).

So far, we presented the average throughput of a single server (our DUT). A naive straightforward analysis might lead to the wrong conclusion that the distribute placement function $\mathcal{P}_d$ outperforms the gather placement function $\mathcal{P}_g$. As can be seen in Figures 5.7, this is not the case. Figures 5.7(a) and 5.7(b) present the average overall throughput estimated by the model defined in Section 5.2 on a set of many servers that are installed with kernel OvS (DPDK-OvS, respectively). The gather placement function $\mathcal{P}_g$ has the advantage of being able to break the traffic between several different independent servers, whereas the distribute placement function $\mathcal{P}_d$ is bound by the wire capacity.

5.4 Monolithic Cost Function

Section 5.3 focuses on exhaustive evaluations, analyzing throughput, OvS packet processing and CPU consumption for different placement functions and software switching technologies (kernel OvS and DPDK-OvS). In the following, we describe how we build the generalized abstract cpu-cost functions, and then present and discuss the results.

5.4.1 Building an Abstract Cost Function

Based on the measured results on a single server (presented in Figures 5.3 and 5.4), we are now ready to craft an abstract generalized cost function that accurately captures the CPU cost of network switching.

We iterate the following process for both kernel OvS and DPDK-OvS. For each placement function ($\mathcal{P}_g$ or $\mathcal{P}_d$) and for each packet size (100Bytes or 1500Bytes), we split the construction and build a set of sub-functions that compose the cpu-cost function, namely $\mathcal{C} = \{\mathcal{C}_{gs}, \mathcal{C}_{ds}, \mathcal{C}_{gb}, \mathcal{C}_{db}\}$ where $\mathcal{C}_{gs}$ and $\mathcal{C}_{gb}$ are sub-functions for placement $\mathcal{P}_g$ and packet size 100Bytes and respectively 1500Bytes, and $\mathcal{C}_{ds}$ and $\mathcal{C}_{db}$ are sub-functions for placement $\mathcal{P}_d$ and packet size 100Bytes and respectively 1500Bytes.

Per each sub-function and for all measured service chain length, we sample the throughput (Figures 5.3(a) - 5.3(d), 5.4(a) - 5.4(d), 5.5(a) - 5.5(b), and 5.6(a) - 5.6(b)) to estimate the amount

Table 5.1: Coefficients α and β, and the constant factor γ, per each cpu-cost sub-function.

	Kernel OvS			DPDK-OvS		
	α	β	γ	α	β	γ
$\mathcal{C}_{gs}$	0.586	0.858	-1.789	0.370	0.467	1.543
$\mathcal{C}_{ds}$	0.660	0.243	-2.661	0.217	0.091	3.795
$\mathcal{C}_{gb}$	0.752	0.979	-3.856	0.478	0.578	0.194
$\mathcal{C}_{db}$	1.009	0.268	-7.176	0.157	0.109	4.718

Source: by author (2017).

of packets that a single server can process, and correlate between the service chain length and the amount of packets. Next, we correlate the resulting packet processing, with the measured CPU consumption (Figures 5.3(c) - 5.3(f), 5.4(c) - 5.4(f)).

Finally, after extracting a 3-dimensional correlation between *(i)* service chains length; *(ii)* packet processing; and *(iii)* CPU consumption, we use the results to extract a set of cpu-cost functions using logarithmic regression, as follows:

$$log(\mathcal{C}_X) = \alpha \cdot log(\varphi|^n) + \beta \cdot log(\varphi|^p) + \gamma \tag{5.1}$$

Where $X \in \{gs, ds, bd, db\}$, namely gather or distribute placements for 100Bytes or 1500Bytes size of packets. Table 5.1 lists per each sub-function the coefficients α and β, and the constant factor γ. Given a service chain φ, the set of sub-functions $\mathcal{C}$ estimates the total required CPU consumption on all servers.

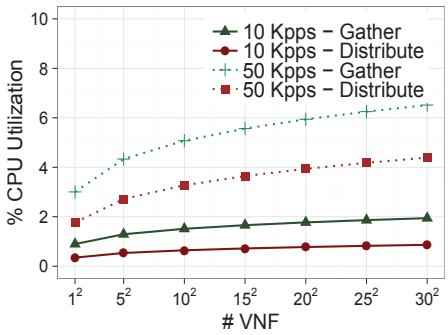
5.4.2 Insights

The values presented in Table 5.1 reflect the real CPU cost of the various deployments, but they provide very little insight regarding our motivation question (see Figure 5.1). In order to get a real understanding of this cost we provide graphs that depict the CPU cost for various service chains characterized by the length of the chain $\varphi|^n$, and the amount of packets to process $\varphi|^p$.

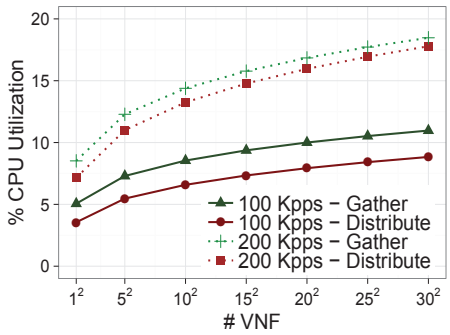
Figures 5.8(a), 5.8(b), and 5.8(c) depict the CPU cost for both placement functions when increasing the number of VNFs (and also the number of service chains) on servers that are installed with kernel-OvS, and traffic is received in large packets (1500Bytes per packet).

In all graphs, the CPU consumption is the total amount of CPU required on all physical machines to support the service chain (where 100% represents all CPU-core on all servers). For

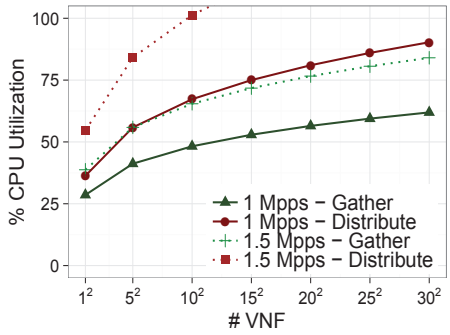
Figure 5.8: Cpu-cost ranging over different service chain length ($\varphi|^n$), while receiving traffic generated in 1500Bytes packets, for both placement functions on servers that are installed with kernel OvS.



(a) 10Kpps and 50Kpps.



(b) 100Kpps and 200Kpps



(c) 1Mpps and 1.5Mpps

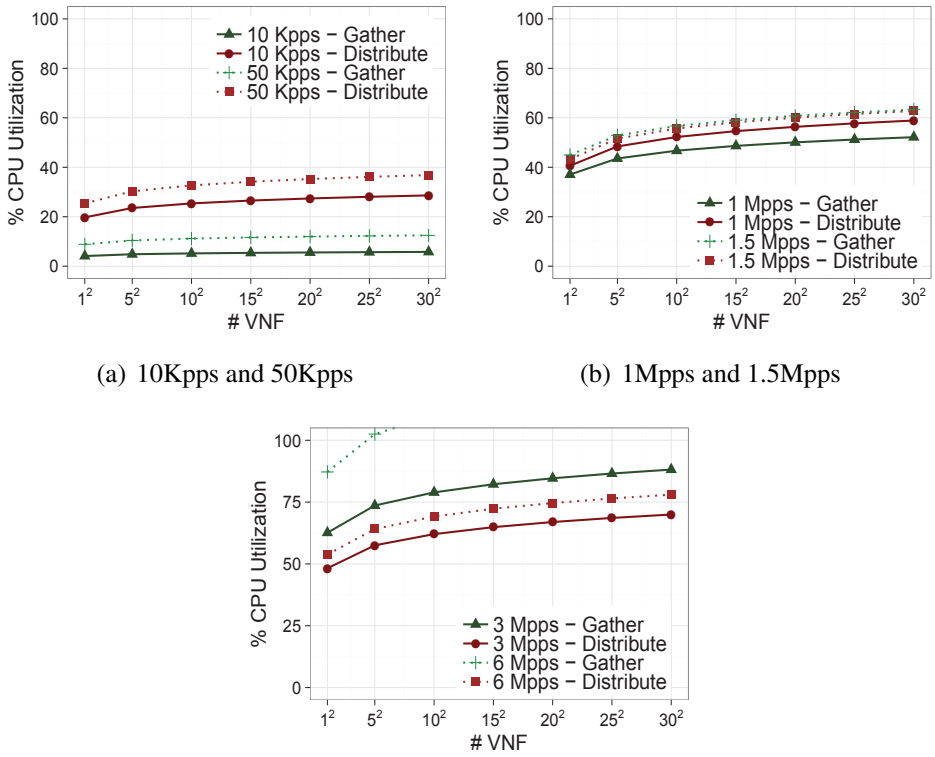
Source: by author (2017).

service chains with low packet processing requirements (10 Kpps - 50 Kpps), the cpu-cost function of the distribute placement $\mathcal{C}_{db}$, outperforms its gather placement counterpart $\mathcal{C}_{gb}$. However, as the requirement for packet processing increases (100 Kpps to 1.5 Mpps), the behaviour turns over and favors the gather placement cpu-cost function $\mathcal{C}_{gb}$.

In turn, Figures 5.9(a), 5.9(b), and 5.9(c) depict the CPU cost for both placement functions when increasing the number of VNFs (and also the number of service chains) on servers that are installed with DPDK-OvS, and traffic is received in 1500Bytes per packet. In this case the behavior changes. For service chains with low packet processing requirements (10Kpps - 50Kpps), the cpu-cost function of the gather placement $\mathcal{C}_{gb}$ outperforms its distribute placement counterpart $\mathcal{C}_{db}$. However, as the requirement for packet processing increases (1Mpps to 6Mpps), the behavior turns over and favors the distribute placement cpu-cost function $\mathcal{C}_{db}$. Both results presented above show that deciding which of the placement strategy is better, depends on the required demand of packets to process, where the exact dependency varies according to the technology used.

Next we examine the cpu-cost function by varying the demand (i.e. required packets to

Figure 5.9: Cpu-cost ranging over different service chain length ($\varphi|^n$), while receiving traffic generated in 1500Bytes packets, for both placement functions on servers that are installed with DPDK-OvS.



(a) 10Kpps and 50Kpps

(b) 1Mpps and 1.5Mpps

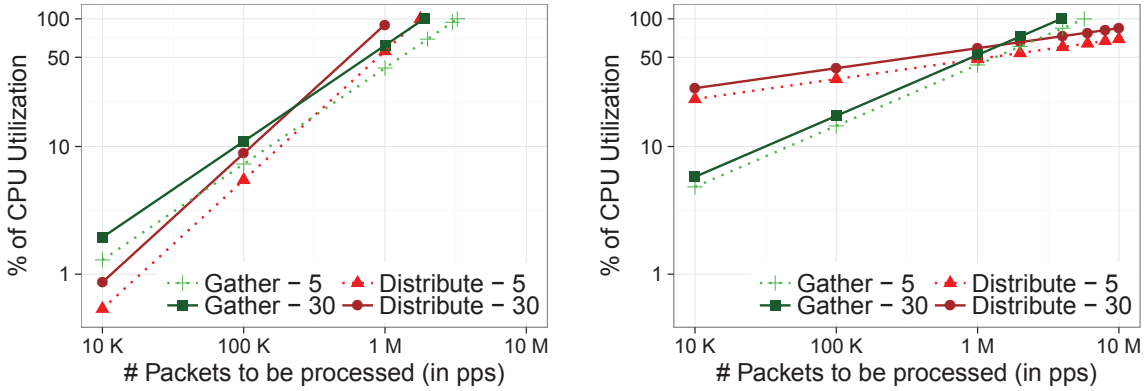
(c) 3Mpps and 6Mpps

Source: by author (2017).

process) of a service chain, for arbitrarily selected few service chain lengths. Figures 5.10 depicts the CPU cost for both placement functions when increasing the required number of packets to process (1500Bytes per packet) on servers that are installed with kernel OvS (Figure 5.10(a)) and DPDK-OvS (Figure 5.10(b)). In these graphs we focus on the definition of a feasible cpu-cost function. Again, we normalize the cpu-cost values (i.e., value 100% represent all CPU-core on all servers), and scale the amount of required packets to process, in order to illustrate the bounds. All values presented in the graphs are in log scale. The graphs reaffirm the results discussed in Figures 5.8 and 5.9, that is, deciding the appropriate placement strategy depends on the required network traffic demand to process.

Recall the definition of feasible cpu-cost function from Section 6.1: a cpu-cost function is feasible if there are enough resources to implement it. In the results presented in Figures 5.8, 5.9, and in Figure 5.10, the value 100% indicates that we have reached the processing bound and we cannot process more packets. We can observe that the infeasibility point is reached differently in each deployment strategy. For instance, the cpu-cost function of the distribute placement $\mathcal{C}_{db}$ reaches its infeasibility point faster than the gather placement $\mathcal{C}_{gb}$ when using

Figure 5.10: Cpu-cost ranging over different packet processing requirements ($\varphi|^p$) 1500Bytes per packet, for both placement functions



(a) Servers installed with kernel OvS

(b) Servers installed with DPDK-OvS

Source: by author (2017).

OvS in Kernel model (Figure 5.10(a)).

6 OPTIMIZING OPERATIONAL COSTS OF NFV SERVICE CHAINING

In Chapter 5, we paved the way towards cost-efficient intra-datacenter deployment of service chains. We performed an extensive experimental evaluation and analyzed the impact on the operational cost. Then, we developed a cost function that accurately predicts the CPU required to maintain the network traffic (with a guaranteed level of performance) for two baseline placement strategies (*i.e.*, *gather* and *distribute*). In essence, results show that the operational cost of deploying service chains depends on many factors including the installed OvS (*i.e.*, whether it supports or not hardware acceleration), the required amount of packets to process, the length of the service chain, and (more importantly) the placement strategy. Further, we show that there is no single deployment strategy that is always *better* – regarding minimum operational costs – being the best strategy dependent on the system parameters and the characteristics of the deployed service chains.

In this chapter, we develop a general service chain deployment mechanism that considers both the actual performance of the service chains as well as the required extra internal switching resource[1]. This is done by decomposing service chains into sub-chains and deploying each one on a (possibly different) physical server, in a way that minimizes the total switching overhead cost. We then introduce a novel algorithm based on an extension of the well-known reduction from weighted matching to min-cost flow problem and show that it gives an almost optimal solution. Additionally, we extend the switching cost formulation defined in Chapter 5 from considering only the gather and distribute placements to a general deployment scheme that can compute the switching related CPU cost for arbitrary deployments.

The remainder of this chapter is organized as follows. In Section 6.1 we define our model and then in Section 6.2 we describe our proposed intra-datacenter algorithm. Last, in Section 6.3, we evaluate the performance of our proposal.

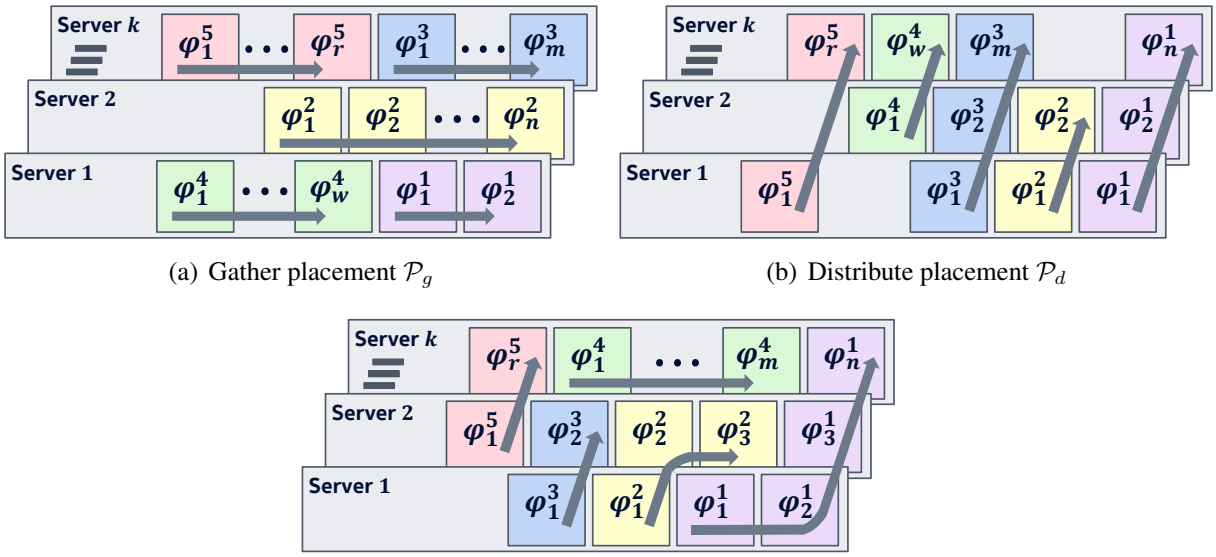
6.1 Model and Problem Definition

In this section, we extend the model and notations previously defined in Chapter 5. As mentioned, the recommended best practice for virtualization intense environment requires each server to allocate two disjoint sets of CPU cores. One specifically to the hypervisor – which manages the VNF provisioning –, and the other to properly operate VNFs. Given a server S we denote by S^h the number of CPU cores that are allocated and reserved solely for the hypervisor to operate, and by S^v the number of CPU cores required by the VNFs to operate. Throughout this chapter, unless explicitly saying otherwise, we assume that we are given a set of k servers $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$.

[1]This chapter is based on the following publication:

- Marcelo Caggiani Luizelli, Danny Raz, Yaniv Saar. **Optimizing NFV Chain Deployment Through Minimizing the Cost of Virtual Switching.** In: Submitted to IEEE INFOCOM 2018.

Figure 6.1: Given a set of five service chains Φ and a set of k servers $\mathcal{S}$, illustrating deployment strategies.



(a) Gather placement $\mathcal{P}_g$

(b) Distribute placement $\mathcal{P}_d$

(c) Arbitrary placement $\mathcal{P}$

Source: by author (2017).

We define a *service chain* to be an ordered set of VNFs $\varphi = \langle \varphi_1 \rightarrow \varphi_2 \ldots \rightarrow \varphi_n \rangle$, and denote by $\varphi|^n$ its length. We denote by $\varphi|^p$ (and $\varphi|^s$) the number of packets per second (and average packet size, respectively) that service chain φ is required to process. For a VNF $\varphi_i \in \varphi$ we denote by $\varphi_i|^c$ the CPU required for its operation. Throughout the chapter, unless explicitly saying otherwise, we assume that we are given a sequence of service chains $\Phi = \{\varphi^1, \varphi^2, \ldots \varphi^m\}$.

A *sub-chain* of VNFs is sub-sequence $\varphi[i..j] = \langle \varphi_i \ldots \rightarrow \varphi_j \rangle$, where $1 \leq i \leq j \leq \varphi|^n$. A set of sub-chains $\{\varphi^1[1..i_1], \ldots, \varphi^r[1..i_r]\}$ is a *decomposition* of φ if the concatenation of all sub-chains recomposes φ, i.e. $\varphi = \langle \varphi^1[1..i_1] \rightarrow \ldots \rightarrow \varphi^r[1..i_r] \rangle$. We say that a set of VNFs is an *affinity group* (and *anti-affinity group*) if all VNFs are mapped to the same server (and respectively mapped to a different servers). We say that a set of sub-chains is a *strict decomposition* if all VNFs in each sub-chain is an affinity group, and exactly one (arbitrary) VNF from each sub-chain is in the same anti-affinity group. Intuitively, strict decomposition of φ is a constraint that ties VNFs in order to implement a placement where each server is visited at most once. We adopt the placement functions already defined in Chapter 5. For that, we use the same definition of a general placement function, that is, $\mathcal{P} : \Phi \rightarrow S^k$ to be a *placement function* that for every service chain $\varphi \in \Phi$, maps every VNF $\varphi_i \in \varphi$ to a server $S_j \in \mathcal{S}$. In this context, we use the placement functions gather and distribute (respectively, $\mathcal{P}_g$ and $\mathcal{P}_d$).

Figure 6.1 illustrates possible deployment strategies. Figure 6.1(a) depicts deployment of VNFs that follows the gather placement functions $\mathcal{P}_g$, while Figure 6.1(b) illustrates the deployment of VNFs following the distribute placement functions $\mathcal{P}_d$. Last, Figure 6.1(c) show

a possible deployment of VNFs that follows arbitrary mixed placement functions $\mathcal{P}$. Note that given a service chain φ, the distribute placement function requires to have at least $\varphi|^n$ servers in order for the deployment to be feasible. Also note that the gather placement function requires to have at least one server that can host all VNFs in φ in order for the deployment to be feasible. In Section 6.2 we develop a new algorithm that allows any arbitrary segmentation of the service chain, while minimizing the operational cost of network traffic.

For a given placement function $\mathcal{P}$ that deploys all service chains in Φ on the set of servers $\mathcal{S}$, we define two cpu-cost functions as follows:

1. $\mathcal{C}^v : \mathcal{P} \rightarrow (\mathbb{R}^+)^k$ is the *guest cpu-cost function* that per server outputs the CPU required to operate the VNFs allocated on the server. For simplicity, we use $\mathcal{C}^v_i$ to refer to the i'th value in the output, and $\mathcal{C}^v$ to denote the sum of all k servers, i.e. $\mathcal{C}^v = \sum_{i=1}^{k} \mathcal{C}^v_i$.
2. $\mathcal{C}^h : \mathcal{P} \rightarrow (\mathbb{R}^+)^k$ is the *hypervisor cpu-cost function* that per server outputs the CPU required to manage its network traffic. For simplicity, we use $\mathcal{C}^h_i$ to refer to the i'th value in the output, and $\mathcal{C}^h$ to denote the sum of all k servers, i.e. $\mathcal{C}^h = \sum_{i=1}^{k} \mathcal{C}^h_i$.

We say that placement function $\mathcal{P}$ is *feasible* with respect to the set of servers $\mathcal{S}$, if for every server in $\mathcal{S}$, the sum of the hypervisor cpu-cost function and the guest cpu-cost function does not exceed the number of CPU cores that are available on this server, i.e.,

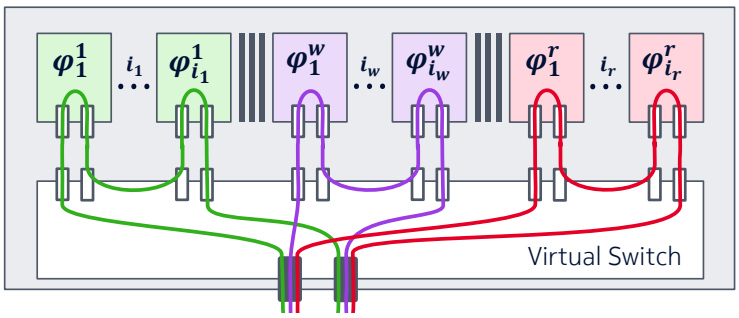
$$\forall_{S_j \in \mathcal{S}} : \mathcal{C}^h_j(\mathcal{P}) + \mathcal{C}^v_j(\mathcal{P}) \leq S^h_j + S^v_j \tag{6.1}$$

We can now formally define the algorithmic problem of interest. We are given a set of k servers, and an online sequence of service chains. For each service chain in the sequence we need to find a feasible placement to all the VNFs in the chain such that the total cpu-cost increase is minimized. In some sense finding a feasible placement for a given service chain is a zero-one acceptance problem since we are not allowed to place partial service chain (*i.e.* either deploying the entire chain or nothing). Minimizing the total cpu-cost (hypervisor) releases additional resources that can be potentially used to allocated additional network functions, which directly improves the acceptance rate of service chains overtime. Our evaluation of acceptance rate (comparing our algorithm to OpenStack scheduler) shows that minimizing the cpu-cost locally brings us to a near-optimal global point at the end of the sequence.

6.2 Operational Cost Optimized Placement

In this section, we introduce our Operational Cost Minimization algorithm (OCM) for the problem described above. Given a sequence of service chains and a set of physical servers, our algorithm entails a strategy to deploy the service chains onto the set of physical servers. Usually, resource scheduling algorithms decide their deployment strategy based on an analysis of the current state of the infrastructure, and the computation resources required for the VNFs

Figure 6.2: Server S_j deployed with r sub-chains from different network services.



Source: by author (2017).

to operate. Our approach extends the typical resource scheduling algorithms by reasoning about the operational cost involved in managing the network traffic to the VNFs. As explained above the idea is that minimizing this cost will release additional resources that can be potentially used to allocated additional network functions. The first step, is then, computing the operational related cost, and as explained in the previous sections, we focus on the internal server switching cpu-cost.

6.2.1 The Operational Cost of Switching

Recall that the cost function is composed of two parts: *(i)* the guest cpu-cost function $\mathcal{C}^v$ that describes the VNF computation cost and is simply the sum of the VNFs CPU requirements, and *(ii)* the more interesting hypervisor cpu-cost function $\mathcal{C}^h$ that describes the CPU cost that is required to manage the network traffic. Given a placement function $\mathcal{P}$, our goal in this subsection is to evaluate the CPU cost of $\mathcal{C}_j^h(\mathcal{P})$.

In Chapter 5, we studied and evaluated the feasibility and the hypervisor cpu-cost function of gather and distribute placements. For a service chain φ, composed of n VNFs (*i.e.*, $\varphi|^n$), they showed how to calculate the values of $\mathcal{F}_g(\varphi)$ which is the cpu-cost of a gather deployment on a server (if the deployment is feasible, otherwise $\mathcal{F}_g(\varphi)$ is infinity), and of $\mathcal{F}_d(\varphi)$ which is the cpu-cost of each server when l copies of φ are deployed in a distributed way on l identical servers (if such a deployment is feasible otherwise $\mathcal{F}_d(\varphi)$ is infinity). This is done separately for both types of OvS installation, kernel OvS and DPDK-OvS.

Using these functions as building blocks, we are now ready to define the $\mathcal{C}_j^h$ function that captures the cpu-cost per server j. Let $\{\varphi^1[1..i_1], \ldots, \varphi^w[1..i_w], \ldots, \varphi^r[1..i_r]\}$ be a set of sub-chains that are deployed on server S_j, as illustrated in Figure 6.2. Note that each sub-chain may be part of a different service chain decomposition, with different properties of a traffic.

The function $\mathcal{C}_j^h$ should take into account both the switching work needed to support the gathering of each sub-chain, as well as the effort given for the parallel deployment of all r sub-chains together. As explained above, the cpu-cost associated with the gathering of each sub-

chain $\varphi^w[1..i_w]$ can be obtained directly from the function $\mathcal{F}_g(\varphi^w[1..i_w])$ (defined in Chapter 5). On the other hand, there is an extra effort because there are several (r in our case) parallel deployments of sub-chains that is not covered by previous definitions (Chapter 5), and thus we approximate this value, using $r \cdot \mathcal{F}_d(\varphi^r_{i_r}) - \mathcal{F}_d(\varphi^r[1..i_r])$. Note that this value is the difference between r times the case that one VNF is deployed on the server and the case where r VNFs are deployed. However, when the VNFs that compose the chain have different characterizations we have to select one of them for this computation and we selected the last one.

Figure 6.3 depicts the overhead cost as a function of the number of sub-chains that exists in the server. The overhead (in CPUs) represents the amount of resources each physical server has to dedicate to properly operate each service chains (which includes hypervisor and software switching layer). We vary the amount of network traffic each sub-chain is processing from 10Kpps to 500Kpps. This is done separately for environment installed with kernel OvS (Figure 6.3(b) for big packets and Figure 6.3(c) for small packets), and DPDK-OvS (for big packets)[2]. The overall computation of $\mathcal{C}^h_j$ is then done as follows:

$$\mathcal{C}^h_j = r \cdot \mathcal{F}_d(\varphi^r_{i_r}) - \mathcal{F}_d(\varphi^r[1..i_r]) + \sum_{w=1}^{r} \mathcal{F}_g(\varphi^w[1..i_w]) \tag{6.2}$$

6.2.2 Chain Partitioning

When receiving a new service chain, we need to find servers for each of the VNFs such that the overall placement is feasible, and has the minimal switching cost (as described above).

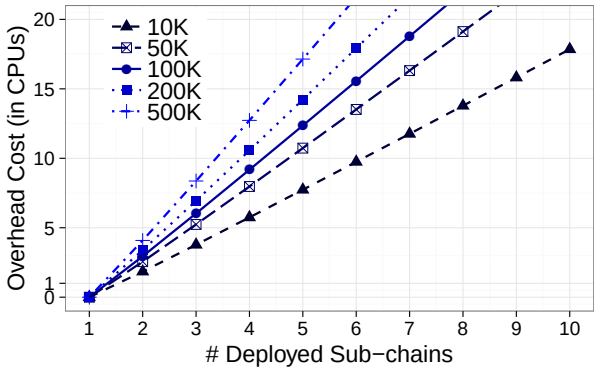
A naive approach would be to go over all VNFs in the service chain, for each one filter the feasible servers and then compute the cost of the placement. This is basically what is done in OpenStack `nova-scheduler` for a single VNF. When we have k servers and N VNFs in the service chain the complexity of this method is $O(k^N)$. Even for small chains (*i.e.*, length four or five), this can easily become impractical even when we have a thousand of servers. Note that the main reason for this is that we need to check the cost of the placement for the entire chain and not for one VNF at a time.

To address this deployment problem we take a different approach. We first consider all possible set partitions of the service chain into subsets of VNFs (not necessarily a decomposition of the service chain) that are co-located on the same server, find the optimal cost for each such partition (we show how to do this in polynomial time), and select the one with the minimal cost.

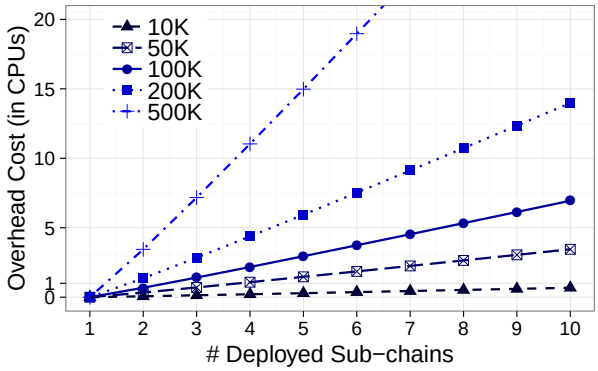
Identifying all possible subsets of VNFs in a service chain φ is equivalent to solving the combinatorial problem of *partition of a set*. Namely given a set of VNFs $\{\varphi_1, \ldots \varphi_n\}$ (all member of service chain φ), enumerate all subsets of the set into non-empty subsets, in such a way that every element is included in one and only one of the subsets. Given an input $N = \varphi|^n$,

[2]The behavior for small packets is similar to big packets.

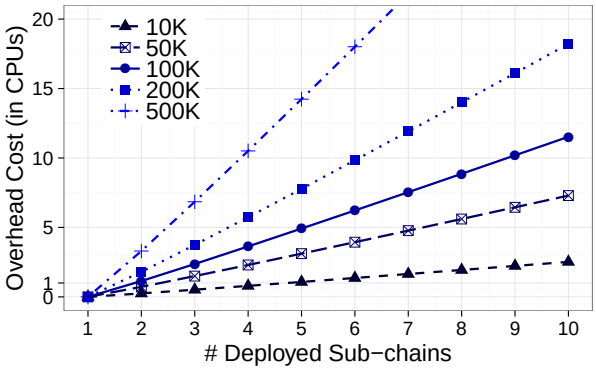
Figure 6.3: The extra effort overhead cost as a function of the number of sub-chains that exists in the server



(a) DPDK-OvS with big packets



(b) Kernel OvS with big packets



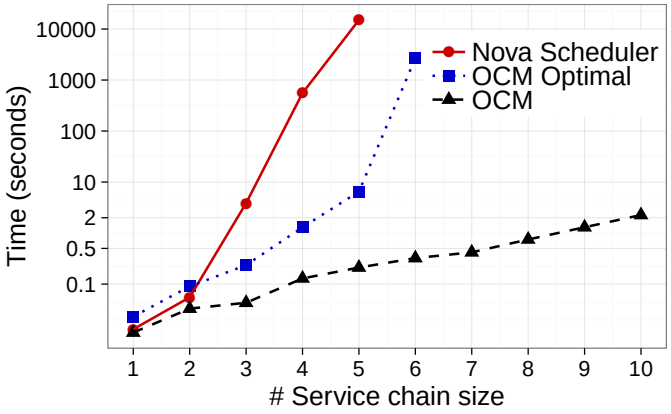
(c) Kernel OvS with small packets

Source: by author (2017).

the number of all possible partitions of a set is the Bell number $B(N)$. Recent study ((BEREND; TASSA, 2010)) established an upper bound of $\left(\frac{0.792N}{ln(N+1)}\right)^N$, i.e. requires $O(N^N)$ iterations. This is already much better than S^N as the typical chain length is less than five and the typical number of servers can be 1K or more. Note, however, that the complete enumeration suggested above also includes instances of placements that allow traffic to: enter a server, do some processing in a certain VNF, then leave the server, and enter it again for the second time in order to do some other processing in a different VNF. Since in practice this is not a reasonable placement, in the following we suggest a relaxed version of the problem.

Assuming service chain φ goes through a server at most once, we can scan all possible decompositions of φ. All possible decompositions of φ is equivalent to solving the combinatorial problem of *integer composition*. Namely, given a positive integer N, enumerate all ordered lists of positive integers $n_1, \ldots, n_r$, such that their sum is equal to N, i.e. $N = \sum_{i=1}^{r} n_i$. The problem of integer composition has been widely studied in the literature, including fast algorithms for its solution (e.g., (MERCA, 2012; KELLEHER; O'SULLIVAN, 2009)). Given an input N the number of all possible integer compositions is exactly 2^{N-1} (i.e., requires $O(2^N)$ iterations).

Figure 6.4: Run time analysis of OCM compared to optimal solution.



Source: by author (2017).

Since in our case the input number to the problem is the length of the service chain $\varphi|^n$ that is practically bound, the enumeration of all integer composition is practically bound as well. Our evaluations show that the enumeration of all integer composition can scale to service chains of length $\sim$20 (executing for few seconds), whereas the enumeration of all set partitions can scale to chains of length $\sim$6 (up to a minute of execution).

Figure 6.4 compares the running time (in log scale) of our approach with `nova-scheduler` implementation that performs an exhaustive search over all possibilities of assigning servers to the NFVs. We ran both methods on a reduced infrastructure setting with 100 servers, since the exhaustive search algorithm cannot handle bigger sizes. On this input, our algorithm does not take more than 2 seconds (in the worst case – long service chains) and 1 second on average to compute the solution, while the optimum surpasses 10K seconds. It is important to emphasize that we significantly improved the running time while delivering quality-wise solutions. For all evaluated solutions, our solution that is based on integer composition was able to find the optimal solution. Furthermore, when scaling our infrastructure to more than 1K servers, it is still able to find solutions in a reasonable time (in the order of few seconds in the worst case).

6.2.3 Using Matching to Find Optimal Cost Placement

To simplify the presentation, in the following we refer to decomposition of a service chain, but the algorithmic steps also apply to the case of partition of a set. For every decomposition $\{\varphi^1[1..i_1], \ldots, \varphi^r[1..i_r]\}$ of the service chain φ, we need to find an optimal placement over the set servers $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ with the current load already taken into account. Again, we could try all possible assignments and select the one with minimal cost but this will bring us back to the exhaustive search complexity. Instead, we use matching to find in polynomial time the optimal servers to assign to each sub-chain of $\varphi^w[1..i_w]$. The idea is to construct a bipartite

graph where the nodes in one side are the sub-chains $\varphi^1, \ldots, \varphi^r$, and the nodes in the other side are all the servers $S_1, S_2, \ldots, S_k$. Sub-chain φ^w is connected to server S_j, with an edge associated with the extra cost induced by placing sub-chain φ^w on server S_j on top the existing sub-chains (i.e., $\mathcal{C}_j^h$ with φ^w minus $\mathcal{C}_j^h$ without it) if placing φ^w on server S_j is feasible. This cost is infinity if it is not feasible to place sub-chain φ^w on server S_j. It is not too difficult to verify that a feasible minimal cost placement of this service chain decomposition corresponds exactly to a minimum-weight perfect matching in the bipartite graph.

Solving the minimum-weight perfect matching in the bipartite graph problem can be reduced to the problem of finding a minimum cost flow in a graph ((AHUJA; MAGNANTI; ORLIN, 1993)). The complexity is polynomial and, therefore, for a given service chain φ of length $N = \varphi|^n$ and set of k servers in $\mathcal{S}$, can be solved in time $O((N + k) \cdot N^2 \cdot k^2)$.

6.2.4 Operational Cost Minimization Algorithm

We are now ready to present our placement algorithm. Roughly, our algorithm has three building blocks: *(i)* list all decompositions of φ or all partitions of the set of all VNFs in φ in extended version of the algorithm (subsection 6.2.2); *(ii)* for a given decomposition build the objective function as suggested above, a cost function that measures the operational cost of network traffic management per each server (subsection 6.2.1); *(iii)* for a given decomposition and objective function build a reduction to minimum-weight matching in bipartite graphs between sub-chains and servers, where the weights are given by the objective function (subsection 6.2.3).

The *operation cost minimization* algorithm receives as an input a set of servers $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ and a sequence of service chains $\Phi = \{\varphi^1, \varphi^2, \ldots \varphi^m\}$. For each service chain $\varphi \in \Phi$, the algorithm invokes the optimal service chain placement step (presented in Algorithm 3) over the input φ and $\mathcal{S}$.

6.3 Evaluation

In order to assess the expected performance of the OCM algorithm and the ability to increase utilization compared to commonly used placement methods, we implemented the OCM algorithm and performed an extensive set of simulation-based evaluations. All experiments were performed on an Intel i7 machine with 8 cores at 2.6Ghz and 16GB of RAM, installed with MacOS 10.11.

6.3.1 Setup

The physical infrastructure representing a typical NFV-node (*i.e.*, small datacenter) compose of 100 physical servers, each having 24 physical cores (again representing typical servers in

Algorithm 3 Optimal service chain placement

Input: $S \leftarrow \{S_1, S_2, \ldots, S_k\}$: set of servers
$\varphi \leftarrow \langle \varphi_1 \rightarrow \varphi_2 \ldots \rightarrow \varphi_n \rangle$: service chain

1: $min\text{-}cost \leftarrow \infty$: minimum cost found so far
2: $deploy\text{-}map \leftarrow$ NIL: maps all VNFs to servers in S
3: $\mathcal{A} \leftarrow$ all partition of a set (or decompositions) of φ
4: **for** every specific partition of a set $a \in \mathcal{A}$ **do**
5: **for** specific partition $p \in a$, and server $S_j \in \mathcal{S}$ **do**
6: $\mathcal{C}_j^h, \mathcal{C}_j^v \leftarrow$ build cost function deploying p on S_j
7: **end for**
8: $G \leftarrow$ build reduction to minimum cost flow in a graph
9: **if** $min(G) \leq min\text{-}cost$ **then**
10: $min\text{-}cost \leftarrow min(G)$
11: $deploy\text{-}map \leftarrow$ extract solution from G
12: **end if**
13: **end for**
14: **return** $deploy\text{-}map$

this environment). Service chains are composed of sequences of VNFs, where the number of VNFs was chosen uniformly at random between 2 and 10 (which is the expected length of service chains in real deployments). We set the VNF computation requirement to be a fixed amount of CPU (1 vCPU). This was done since our main goal is to assess operational costs on deployments. Note that increasing the required amount of CPU per VNF does not change the operational costs, it only impacts the acceptance ratio and overall resource utilization. We vary the amount of packets to be processed for each service in the range 10K to 1.5M packets. We also considered both kernel mode as well as DPDK packet acceleration for evaluating the operational costs when deploying service chains using Open vSwitch.

For the purpose of this evaluation, we developed a generator of service chains requests. Service chain requests are handled one by one as described in Section 6.1. Each experiment was performed 30 times, considering different service chain requests.

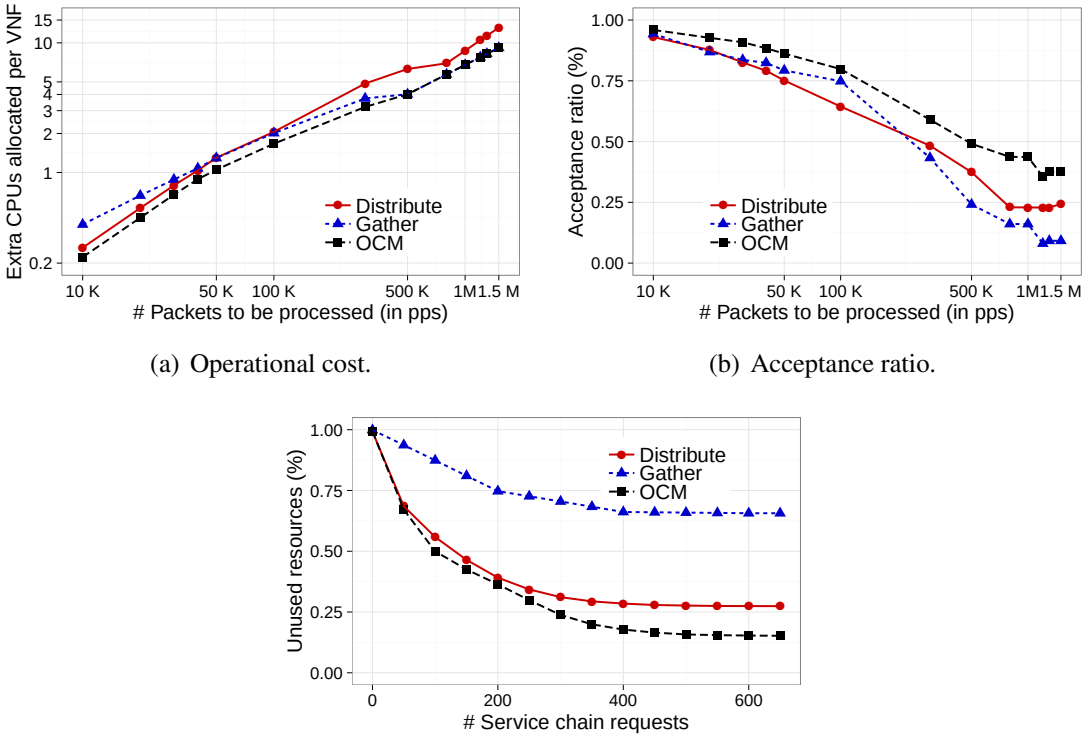
6.3.2 Comparison to OpenStack

We compare the OCM algorithm to two possible deploying strategies in *OpenStack/Nova*: *(i)* load balanced policy (distribute placement) *(ii)* energy saving policy (gather placement). Observe that we mimic the `nova-scheduler` engine in our simulator by using the gather and distribute policies, which assume hard requirements (instead of soft constraints) and, therefore, requests are rejected whenever not possible to meet such requirements.

6.3.3 Results

NFV operational cost. Figures 6.5(a) and 6.6(a) depict the average additional CPU required to deploy VNFs. This additional cost represents resources specifically allocated to keep up the network performance (e.g., throughput) of software switching. As we observe, the additional number of physical cores needed for the virtualized switching functionality is not negligible, and as expected the cost tends to be higher as the traffic requirement (in packet per seconds) increases. For high network requirements (1M to 1.5M pps), the operational cost is up to half of the available resources ($\approx 10 - 12$ cores) of a whole physical server (see Figure 6.5(a)). Observe that the operational cost of `nova-scheduler/distribute` is substantially higher in DPDK-enable OvS server for low requirement network traffic. In OvS-DPDK, there is at least a subset of poll-mode threads (doing busy-waiting) in the hypervisor (which consumes resources independently of the network traffic). As the `nova-scheduler/distribute` deployment tries to utilize available servers evenly, for low-requirement network traffic there is a low level of sharing of those resources (which leads to higher operational costs). Observe that as the network traffic requirement increases, such cost tends to lower in comparison to kernel-base implementations.

Figure 6.5: Analysis of service chains deployment on NFV servers with Open vSwitch in kernel mode.



(a) Operational cost.

(b) Acceptance ratio.

(c) Resource utilization.

Source: by author (2017).

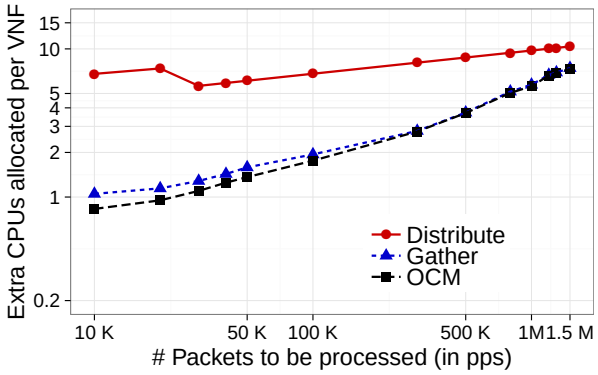
When comparing the OCM algorithm to the other two methods, one can see that the amount of the additional resources required when placement is done by the OCM algorithm is always below the other two methods. The amount of improvement depends on the specific parameters and in extreme cases can be as high as a factor of 4. However, in most cases the improvement is between 20% and 40%. Note that reducing the switching cost is not the ultimate goal, the goal is to free resources in order to be able to support additional functionality, so the real benefit is the ability to accept additional services as shown next.

Service chain acceptance ratio. Figures 6.5(b) and 6.6(b) depict the average acceptance ratio (this is the ratio between the number of deployed chains and the number of the requested chains) for different network traffic requirements. It is clear that, on average, OCM is able to deploy a higher number of requests. In the worst case, it behaves similarly to `nova-scheduler/gather` deployment strategy. Due to higher operational cost of deploying services in a distributed way (e.g., in DPDK mode), the acceptance ratio is substantially affected in those cases (see Figure 6.6(b), the acceptance is $\approx 30\%$ in this case). However, note that for higher network traffic requirements, there is a slight improvement of `nova-scheduler/distribute` over `nova-scheduler/gather`. OCM, in turn, even for high requirements has shown to better take advantage of the available resources. This is because OCM provides flexible solutions that better fits available physical resources and, therefore, it ensures a higher acceptance rate. The improvement of OCM over standard deployment policies is much more noticeable if we look only at large requests.

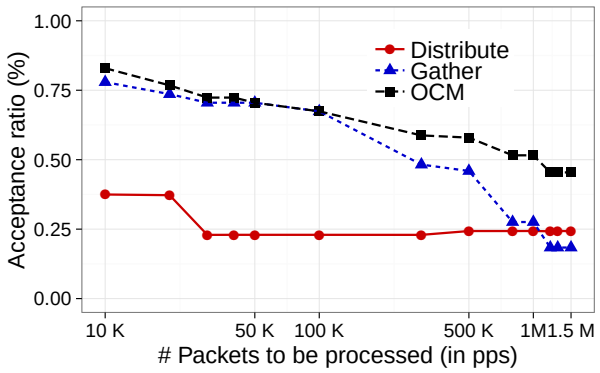
Physical infrastructure resource utilization. Figures 6.5(c) and 6.6(c) illustrate the average unused resource over time (i.e., the lower the average values the better). Observe that OCM can utilize more resources in the physical infrastructure (up to 18% of unused resources) with comparatively lower operational cost (see Figures 6.5(a) and 6.6(a)). The average of unused resources may be lower depending on the workload (for longer service chains, OCM can better take advantage of chaining decomposition). In the OvS-DPDK, the average percentage of unused resources is slightly higher in comparison to kernel mode. That happens because DPDK requires a fixed amount of resources to operate (threads required by poll-mode drivers) which reflects in high operational costs and less flexibility on placing service chains.

Overall one can see that trying to reduce the switching CPU cost locally at each deployment of a new request, proves itself to be a very good strategy as it improves the acceptance rate and allows a much better utilization of the resources.

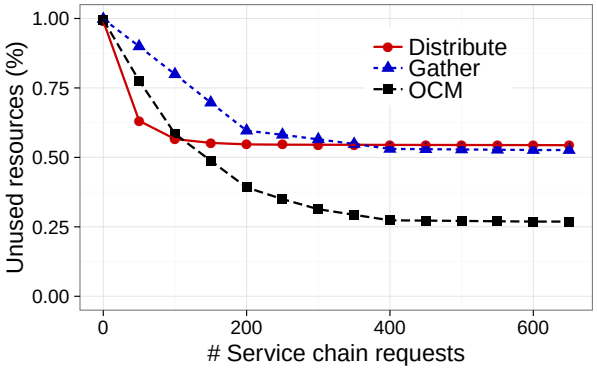
Figure 6.6: Analysis of service chains deployment on NFV servers with Open vSwitch in DPDK mode.



(a) Operational cost.



(b) Acceptance ratio.



(c) Resource utilization.

Source: by author (2017).

7 OPTIMIZING DISTRIBUTED NETWORK MONITORING IN NFV

In the previous chapters, we approached the Inter- and Intra-datacenter VNFPC problem. The interplay between these two optimization problems ensures an efficient and scalable provisioning of service chains. In this chapter, we tackle the problem of monitoring network traffic in service chains – another important building block for the properly *operation* of NFV-based network services[1]. We propose an optimization model that is able to effectively coordinate the monitoring of service chains in the context of NFV deployments. The model is aware of SFC topological components, which allows to independently monitor elements within a network service.

The remainder of this chapter is organized as follows. In Sections 7.1 and 7.2, we motivate the Distributed Network Monitoring (DNM) problem and assess the impact of forwarding network traffic (to monitoring services) on network performance. We then formalize the DNM problem using an ILP model in Section 7.3. Last, in Section 7.4, we present the performance evaluation of our proposed model.

7.1 Problem Motivation

Collecting and analyzing flow measurement data are essential tasks to properly operate a network infrastructure. By continuously analyzing this data, network operators can diagnose performance problems (*e.g.*, congested links (ZHU et al., 2015)), detect network attacks (*e.g.*, superspread attack (YU; JOSE; MIAO, 2013)), or even perform traffic engineering (*e.g.*, detect large traffic aggregation (BEN-BASAT et al., 2017) or traffic changes) – to name just a few possible monitoring operations. Considering a software-based network infrastructure (*i.e.*, NFV- and SDN-based one), network operators have increased flexibility to dynamically collect network traffic and, consequently, monitor network services thoroughly.

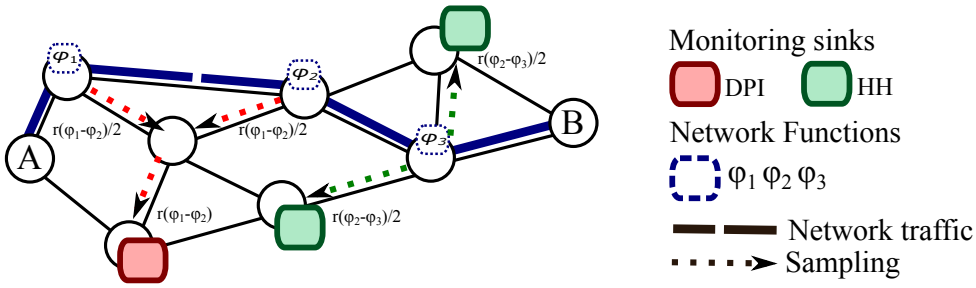
The flexibility offered by software-based environments allows monitoring services to be dynamically and efficiently deployed. In the context of NFV, monitoring services are orthogonal to the deployment of SFCs – as the network monitoring involves to cope with highly dynamic events. As previously defined in Chapters 3 and 5, service chains can be defined as an ordered set of network functions that the network traffic is steered through. As an example, we can consider the following service chains φ^1 composed of three VNFs – $\varphi^1 = \langle \varphi^1_1 \rightarrow \varphi^1_2 \rightarrow \varphi^1_3 \rangle$. In this particular service chain, there might exist independent monitoring requirements for each segment. In other words, segments $\varphi^1_1 \rightarrow \varphi^1_2$ and $\varphi^1_2 \rightarrow \varphi^1_3$ might be monitored according to multiple policies so as to fulfill different network monitoring demands at packet- or flow-level.

[1]This chapter is based on the following publication:

- Marcelo Caggiani Luizelli, Luciana Salete Buriol, Luciano Paschoal Gaspary. **In the Right Place at the Right Time: Optimizing Distributed Network Monitoring for NFV Service Chains** In: To be submitted to IEEE NOMS 2018.

For instance, the egress network traffic of VNF φ_1^1 could be forwarded to a specific monitoring system with a predefined traffic rate $r(.)$, while the egress of VNF φ_3^1 to a different one. Figure 7.1 illustrates an example where the SFC φ^1 is deployed along multiple locations in an NFV-based infrastructure. Consider that SFC φ^1 is composed of a firewall (VNF φ_1^1), an encryption function (VNF φ_2^1), and a user accounting function (VNF φ_3^1) – in this respective order. On monitoring the SFC φ^1, the network traffic should be carefully collected so as to ensure the correct analysis of network data. In the example, the SFC φ^1 is constantly monitored by a DPI and a HH (heavy hitter) solution. As the DPI requires the raw network traffic to properly operate (*i.e.*, non-encrypted), the network traffic should be sent to it right after being processed by VNF φ_1^1 and right before VNF φ_2^1. In turn, the HH can receive the network traffic any point after φ_3^1 – assuming that the operator wants to know the top-k flows authorized by the accounting system. Figure 7.1 illustrates a possible solution for the problem of steering the network monitoring traffic to a set of monitoring sinks. From this point on, we use the term *monitoring sink* as a reference to monitoring function.

Figure 7.1: Coordinated network monitoring of SFC-based network services.



Source: by author (2017).

While some SFC segments are required to collect/aggregate flow-level statistics (*e.g.*, number of packets/volume of a given flow), other segments are required to collect specific packet-level information to be sent to complex monitoring sinks. Packet-level monitoring enables network operators to deepen the understanding of a significant number of network behaviors – *e.g.*, understand flow properties (such as entropy), measure delay or throughput per flow (YU, 2014), and perform complex network traffic inferences (such as identify network attacks). Despite these benefits, packet level monitoring incurs in major network overheads due to the collection, storage, transmission, and analysis of an enormous amount of packets (particularly, at line rate). As illustrated, examples of packet level monitoring applications include DPI and HH solutions. While DPI requires the inspection of all packet contents (headers and payload), HH solutions (BEN-BASAT et al., 2016) demand header checking (packet size). To reduce the burden of packet-level monitoring, complex monitoring systems rely on network sampling mechanisms (*e.g.*, (BEN-BASAT et al., 2017)). Consequently, network devices can keep up with network performance requirements while the monitoring system can still provide accurate results.

To fulfill monitoring requirements (at flow- and packet-level), network devices rely extensively on flow export protocols such as sFlow (PHAAL; PANCHEN; PANCHEN, 2001) and NetFlow (CLAISE, 2004). Despite their wide adoption in traditional infrastructures, these protocols are rigidly designed to either mirror or uniformly sample network traffic to monitor applications. As a consequence, it has led to a few drawbacks on monitoring NFV service chains. These drawbacks include the following.

1. Monitoring applications are bounded to run at predefined locations in the network infrastructure.
2. Network flows are treated equally, being sampled at the same rate.
3. Network flows are potentially sampled multiple times along the path a service chain is deployed – in case many routers are enabled to perform sFlow/Netflow.

As one can observe, traditional monitoring mechanisms are not flexible enough to cope with monitoring requirements imposed by NFV-based service chains – namely, independent and coordinated monitoring of service chain segments. Fortunately, NFV infrastructures have largely adopted software switching (*e.g.,* Open vSwitch or P4-based dataplanes (BOSSHART et al., 2014; SHAHBAZ et al., 2016)) as a building block to interconnect VNFs. In addition to interconnect VNFs, software switching allows network traffic to be monitored (by sampling and/or forwarding network traffic to monitoring sinks) at any granularity (by filtering network flows) and at any position of a given deployed service chain. The benefit, in summary, is to provide a fully customized monitoring system that can be orchestrated orthogonally to SFC deployments in a flexible way.

7.2 The Cost of Packet-Level Monitoring in Software Switching

The drawbacks discussed above of traditional monitoring mechanisms are overcome in NFV environments by taking advantage of existing capabilities of (programmable) software dataplanes. These capabilities include mainly *(i)* fine-grained traffic matching and *(ii)* flexible mechanisms to sample network traffic. Therefore, such mechanisms enable service chains to be monitored at any given point (*e.g.,* egress/ingress ports) to as many as necessary monitoring sinks.

However, providing monitoring flexibility directly on the dataplane also incurs in network performance overheads. In order to simultaneously forward network traffic (coming from different service chains) and sample to monitoring sinks, the software switching layer requires an additional amount of computing resources to keep up with performance requirements. Despite this resource overhead, sampling mechanisms are much more efficient than sending all packets out to monitoring sinks for two main reasons: *(i)* network links are not overwhelmed by monitoring packets and *(ii)* dataplane resources are saved as it reduces substantially the number of packet copies performed between network interfaces (physical and virtual ones). This is par-

ticularly important as the dataplane is upper-bounded by the number of packets it can handle (LUIZELLI et al., 2017b).

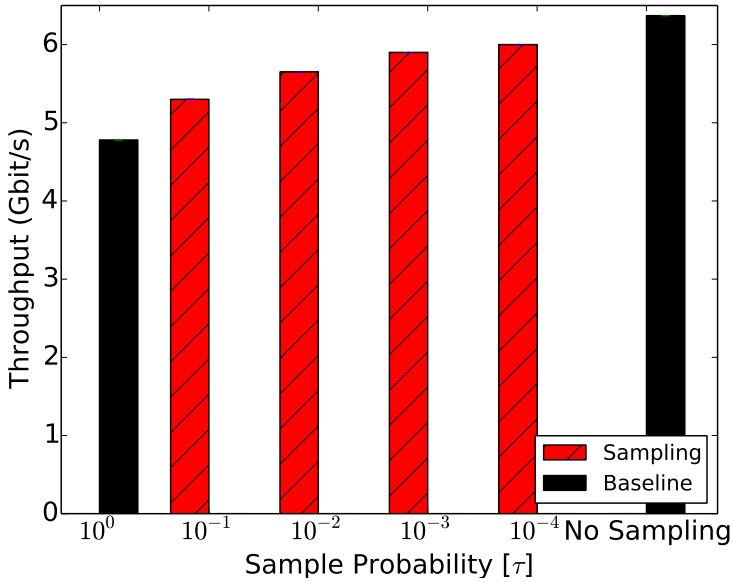
Performing per-packet sampling directly into the dataplane introduces greater flexibility to monitor service chains. However, it comes with a cost. In order to quantify network performance degradation when performing network sampling, we conduct a set of experiments in a typical NFV environment. For that, we deployed one service chain composed of a single VNF (*i.e.*, $\varphi^1 = \langle \varphi^1_1 \rangle$). All the traffic processed by VNF φ^1_1 is then sampled to a monitoring sink. We vary the sampling rate from 10^{-4} to 10^{-1} and compare it against two baselines cases: *(i)* mirroring network traffic (*i.e.*, copy all packets to the monitoring sink), and *(ii)* without any sampling. We highlight that our main goal is not to provide the best performance on sampling packets, but rather motivate about the potential benefits and incurred costs of sampling.

Our NFV evaluation environment consists of two HP ProLiant servers. Each server is equipped with an Intel Xeon E3-1220v2 processor, with four physical cores running at 3.1 Ghz. Each server has 8 GB RAM and a DPDK-enabled Intel 82599ES 10 Gbit/s NIC with two physical ports – which are used to interconnect both servers. The operating system running on each server is CentOS version 7.2.1511, Linux kernel version 3.10.0. One of the servers is used as *Design Under Test (DUT)*, and the other is used as a traffic generator. In our DUT, we use Open vSwitch 2.6[2] – compiled with Intel's DPDK 16.07. The VNF and the monitoring sink are configured with Fedora 22 operating system, Linux kernel version 4.0.4 – running on top of qemu version 2.6. Further, they are configured to operate with a single virtual CPU (pinned to a specific physical core) and 512MB of RAM. In turn, our traffic generator server runs MoonGen (EMMERICH et al., 2015) to send UDP network traffic at line speed through our deployed service chain φ^1.

Figure 7.2 illustrates the maximum achieved throughput on the evaluated scenario. In case there is sampling, the Open vSwitch dataplane first copies the packets to the egress port (in compliance with service chain specification) and then to the monitoring sink. As one can observe, the more packets are copied from VNF φ^1_1 to the monitoring sink (*e.g*, 1 out of 10 – sampling rate of 10^{-1}), the lower is the network performance observed on the dataplane. Further, observe that full mirroring (*i.e.*, sampling rate of 10^0) incurs in considerably high throughput overhead (~40% lower) in comparison to the case in which there is no sampling (*i.e*, VNF φ^1_1 just sends the network traffic to the egress port). Note, however, that sampling leads to an acceptable overhead – being as low as 5%. The reason for that performance degradation is due to *(i)* more packet copies (according to the sampling rate) and *(ii)* more CPU cycles spent for each packet (for each packet, the dataplane checks whether or not to sample).

[2]We extended the Open vSwitch dataplane capabilities in order to support sampling. Our code is publicly available at https://github.com/ovssample

Figure 7.2: Performance degradation when sampling in software switching.



Source: by author (2017).

7.3 Distributed Network Monitoring: Problem Overview and Optimization Model

In this section, we first introduce the Distributed Network Monitoring (DNM) problem, and then we propose an optimization model to solve it.

7.3.1 Problem Overview

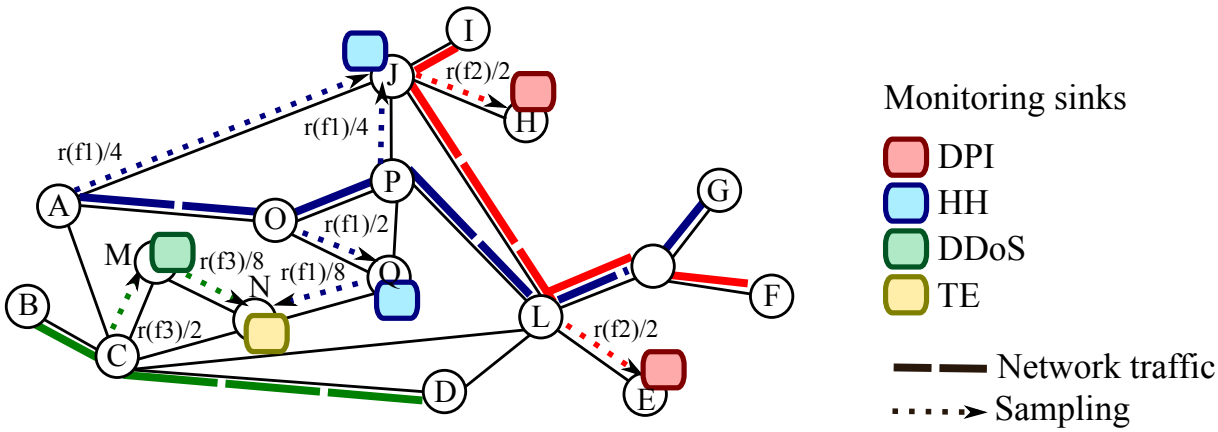
We start by introducing the DNM problem with an example. The problem in hand relies extensively on sampling mechanisms by NFV infrastructures to efficiently monitor service chains.

Consider an NFV infrastructure composed of N-PoPs interconnected by physical links. Each N-PoP has a limited CPU capacity to support the operation of VNFs, while links have limited bandwidth capacity. Attached to each N-PoP, there is a forwarding device with a limited capacity to sample network traffic. On top of this NFV infrastructure, there are many deployed service chains. For the sake of simplicity, consider three identical service chains φ^1, φ^2 and φ^3 (in terms of VNF composition) – that is, $\varphi^{1,2,3} = \langle \varphi_1 \rightarrow \varphi_2 \rightarrow \varphi_3 \rangle$. For example, each service chain is specialized on handling different network traffic. Consider that HTTP network traffic is steered through service chain φ^1, video streaming traffic through service chain φ^2 and all non-classified network traffic through service chain φ^3.

As previously mentioned, there exist complex network monitoring sinks that might be applied individually (or simultaneously) to each network flow. In this example, we consider that the HTTP network traffic (being steered through service chain φ^1) is monitored by a monitor sink that implements DDoS detection/prevention. Similarly, the network traffic steered through

service chain φ^2 (*i.e.*, video streaming) is monitored by HH. Last, all non-classified network traffic is monitored by a DPI. Yet, we assume that service chains φ^1 an φ^2 are monitored simultaneously by a Traffic Engineering (TE) solution.

Figure 7.3: Example of a solution for the Distributed Network Monitoring (DNM) problem.



Source: by author (2017).

In our example, we have a set of monitoring applications, represented by M. Set M is composed of DDoS, HH, DPI and TE monitoring functions. Each monitoring application $m \in M$ requires a specific portion of the network traffic so as to accurately process the input. Let's consider that the network sampling rate each network monitor function requires is represented by function $r(\cdot)$ and, therefore, for all $m \in M$, there is a sampling rate $r(m)$ known in advance. Note that the sampling rate required by monitoring sinks $m \in M$ might change overtime – for that cases, we say that a sampling rate is known in advance at each time slot.

As mentioned, the DNM problem relies on the ability to sample network traffic along the path each service chain is deployed. Then, a valid solution for the DNM admits distributing the required sampling rate for a given monitoring service through arbitrarily network devices. Hence, a solution for the DNM problem determines *(i)* how to split and forward sampled network traffic efficiently and *(ii)* where to place network monitoring sinks according to the demand. Note that the NFV infrastructure is constrained by the number of packets/s each forwarding device can handle and, therefore, sampling all network traffic at one location might incur in high-performance degradation. Figure 7.3 illustrates a possible solution for the DNM problem. In the figure, we have the three service chains deployed (represented by the bold dashed lines) on top of the NFV infrastructure. For simplicity, we omit where network functions are deployed (*i.e.*, service chains φ_1, φ_2 and φ_3) and focus on showing the position of monitoring network functions (*i.e.*, $m \in M$). Note that the network traffic being steered through service chains are sampled according to the required sampling rate $r(m)$ of each monitoring application. Further, each network monitoring sink $m \in M$ can be deployed multiple times (as the case of DPI and HH in the example).

7.3.2 Model description and notation

We start by describing both the input and output of the model, and establishing a supporting notation. We use superscript letters ^{P} and ^{S} to indicate symbols that refer to physical resources and SFC requests, respectively. Similarly, superscript letters ^{N} and ^{L} indicate references to N-PoPs/endpoints and the links that connect them.

The optimization model we use for solving the DNM problem considers a set of already deployed service chains Q and a physical infrastructure p, the latter a tuple $p = (N^P, L^P)$. N^P is a set of network nodes, and pairs $(i, j) \in L^P$ denote unidirectional physical links. We consider each node $i \in N^P$ as a computing node directly connected to a forwarding device. We use two pairs in opposite directions (*e.g.*, (i, j) and (j, i)) to denote bidirectional links. The model captures the following resource constraints: computing power for N-PoPs (c_i^P), sampling capacity of forwarding device (s_i^P) and one-way bandwidth for physical links ($b_{i,j}^P$).

A service chain $q \in Q$ is an aggregation of network functions and chaining between them. It is represented as a tuple $q = (N_q^S, L_q^S)$. Sets N_q^S and L_q^S contain the SFC nodes and virtual links connecting them, respectively. Each virtual link $(i, j) \in L_q^S$ is potentially chained through a path in the physical infrastructure p. We denote the deployed path of a virtual link (i, j) by a set of physical links $P_{q,i,j}^S \subseteq L^P$.

M denotes the set of monitoring sinks (*e.g.*, HH, DPI, DDoS) available for deployment in the infrastructure. Each VNF has U_m instances, and may be instantiated at most $|U_m|$ times. We denote as $f_{type} : N^P \cup N^S \to M$ the function that indicates the type of a given VNF, which can be either one instantiated in an N-PoP (N^P) or one requested in an SFC (N^S). We also use functions $f_{cpu} : (F \times U_m) \to \mathbb{R}_+$ and $f_{pkt} : (F \times U_m) \to \mathbb{R}_+$ to denote computing power requirement and packet processing capabilities of a given VNF.

As previously discussed, service chains $q \in Q$ might have multiple independent monitoring requirements. We define monitoring requirements to individual virtual links $(i, j) \in L_q^S$. In other words, each link (i, j) is associated with a set of monitoring sinks, denoted by set $M_{q,i,j}^S \subseteq M$. On monitoring virtual links independently, the model allows to cope with topological dependencies of already deployed network services. This is particularly important for monitoring applications that require network traffic right after/before being handled by specific network functions – *e.g.*, packets sent to a DPI must not be encrypted/compressed. Last, for each monitoring sink $m \in M_{q,i,j}^S$, we denote the sampling rate it requires to properly operate as $r_m^{q,i,j}$.

The model output is denoted by a 4-tuple $\chi = \{Y, F, D, S\}$. Variables from $Y = \{\, y_{i,m,j} \,, \forall\, i \in N^P, m \in F, j \in U_m \,\}$ indicate a monitoring sink placement, *i.e.* whether instance j of monitoring sink m is mapped to N-PoP i. The variables from $F = \{\, f_{i,j}^{q,k,l,m} \,, \forall\, q \in Q, (k, l) \in L_q^S, m \in F, (i, j) \in L^P \,\}$, in turn, represent how the network traffic is split (and forwarded) to the required set of monitoring services. Specifically, they indicate how much network traffic, required by monitoring service m on virtual link (k, l) belonging to service chains q, is for-

warded through physical link (i, j). The variables from $D = \{\, d_i^{q,k,l}\,, \forall\, q \in Q, (k,l) \in L_q^S, m \in F, i \in R^P \,\}$ represent how much traffic is sent by forwarding device i for a given virtual link $(k,l) \in L_q^S$. Similarly, the variables from $S = \{\, d_i^{q,k,l}\,, \forall\, q \in Q, (k,l) \in L_q^S, m \in F, i \in R^P \,\}$ represent the amount of network traffic that is received by forwarding device i for a given virtual link $(k,l) \in L_q^S$. Variables Y may assume a value in $\{0,1\}$, while variable sets $\{F, D, S\} \in \mathbb{R}_+$.

7.3.3 Model Formulation

Next, we describe the linear integer programming formulation for the DNM problem. The goal of the objective function is to minimize simultaneously the number of deployed monitoring sinks and the cost of forwarding monitoring flows. The objective is illustrated in Equation 7.1.

Objective:

$$\textbf{Minimize.} \quad \alpha \cdot \sum_{i \in N^P} \sum_{m \in M} \sum_{j \in U_m} y_{i,m,j} + \beta \cdot \sum_{\forall q \in Q} \sum_{\forall (k,l) \in L_q^S} \sum_{\forall m \in M} \sum_{\forall (i,j) \in L^P} f_{i,j}^{q,k,l,m} \tag{7.1}$$

Subject to:

$$\sum_{j \in R^P} f_{i,j}^{q,k,l,m} - \sum_{j \in R^P} f_{j,i}^{q,k,l,m} = s_{i,m}^{q,k,l} - d_{i,m}^{q,k,l} \qquad \forall q \in Q, (k,l) \in L_q^S, i \in R^P, m \in M_{q,k,l}^S \tag{7.2}$$

$$\sum_{i \in P_{q,k,l}^S} s_{i,m}^{q,k,l} = r_m^{q,k,l} \qquad \forall q \in Q, (k,l) \in L_q^S, m \in M_{q,k,l}^S \tag{7.3}$$

$$\sum_{q \in Q} \sum_{(k,l) \in L_q^S} \sum_{m \in M_{q,k,l}^S} s_{i,m}^{q,k,l} \leq s_i^P \qquad \forall i \in R^P \tag{7.4}$$

$$\sum_{q \in Q} \sum_{(k,l) \in L_q^S} \sum_{m \in M_{q,k,l}^S} f_{i,j}^{q,k,l,m} \leq b_{i,j}^P \qquad \forall (i,j) \in L^P \tag{7.5}$$

$$\sum_{m \in M} \sum_{j \in U_m} y^{i,m,j} \cdot f_{cpu}(m,j) \leq c_i^P \qquad \forall i \in N^P \tag{7.6}$$

$$\sum_{q \in Q} \sum_{(k,l) \in L_q^S} d_{i,m}^{q,k,l} \leq y_{i,m,j} \cdot f_{pkt}(m,j) \qquad \forall i \in R^P, m \in M_{q,k,l}^S, j \in U_m \in \quad (7.7)$$

$$f_{i,j}^{q,k,l,m} \in \mathbb{R}_+ \qquad \forall (i,j) \in L^P, q \in Q, (k,l) \in L_q^S, m \in M_{q,k,l}^S \quad (7.8)$$

$$s_{i,m}^{q,k,l}, d_{i,m}^{q,k,l} \in \mathbb{R}_+ \qquad \forall i \in R^P, q \in Q, (k,l) \in L_q^S, m \in M_{q,k,l}^S \quad (7.9)$$

$$y^{i,m,j} \in \{0,1\} \qquad \forall i \in R^P, m \in M, j \in j \in U_m \quad (7.10)$$

The first two constraint sets refer to the forwarding of network monitoring traffic. Constraint set (7.2) ensures *(i)* network traffic is sampled by forwarding devices; *(ii)* the flow conservation along the path that monitoring traffic is routed through; and *(iii)* that N-PoPs (or network nodes) receive the sampled network traffic. In turn, constraint set (7.3) guarantees that just a subset of forwarding devices can sample network traffic for a given SFC, that is, the ones belonging to the path the SFC is currently deployed. In addition, it ensures that the sampled network traffic meets a sampling rate for a given monitoring sink. Constraint sets (7.4), (7.5), (7.6), and (7.7) are related to capacity aspects. Constraint set (7.4) ensures an upper-bounded on the number of network traffic being sampled. Constraint sets (7.5) ensures that the monitoring traffic does not exceed the available bandwidth on the links used to forward packets to monitoring sinks. Constraint set (7.6) ensures that the monitoring sinks does not exceed the available computing capacity of a particular N-PoP. Last, constraint set (7.7) ensures that the monitoring sinks have available capacity to process incoming monitoring traffic. Constraint sets (7.8), (7.9), and (7.10) define the domain of output variables F, S, D and Y.

7.4 Evaluation

In this section, we evaluate the effectiveness of our mathematical model in generating feasible solutions to the DNM problem. The ILP model formalized in the previous section was implemented and run in *CPLEX Optimization Studio*[3] version 12.7.1. All experiments were performed on a machine with four Intel Xeon E5-2670 processors and 56 GB of RAM, using the Ubuntu GNU/Linux Server 11.10 x86_64 operating system. Next, we describe our experiment workload, followed by results obtained.

[3]http://www-01.ibm.com/software/integration/optimization/cplex- optimization-studio/

7.4.1 Setup

To perform the experiments, we adopted a similar workload to recent literature, such as the ones conducted by (BARI et al., 2015; LUIZELLI et al., 2017a). The physical network substrate was generated with Brite[4], following the Barabasi-Albert (BA-2) (ALBERT; BARABÁSI, 2000) model. We consider physical infrastructures consisting of 100 N-PoPs and 393 physical links. The computing power of each N-PoP is normalized and set to 1, and each link has a bandwidth capacity of 10 Gbps. Attached to each N-PoP, there is a forwarding device with limited sampling capability. We set the sampling capability (c_i^P) of each forwarding device to 5% of its line rate.

Our workload is comprised of a set of already deployed SFCs. For simplicity, we consider that all deployed SFCs are uniform – *i.e.*, they have the same requirements in terms of VNFs and bandwidth. The topology of each SFC consisted of: a source endpoint, a sequence of three VNFs interconnected by links (supporting 1 Gbps of traffic flow), and a destination endpoint. Source and destination endpoints are placed randomly at predefined locations in the infrastructure (*i.e.*, location constraints adopted by VNFPC (LUIZELLI et al., 2017a)). All VNF instances require a normalized computing power capacity of 0.25. Our model takes as input the mechanism for placement & chaining of SFCs (VNFPC model), previously defined in Chapter 3. The output of VNFPC determines the set $P_{q,i,j}^S : (\forall q \in Q, (i,j) \in L_q^S)$ that specifies the set of physical links used to steer network traffic from i to j in SFC q.

We consider a set of five types of arbitrary monitoring sinks (defined in our model as the set M). Monitoring sinks require a normalized computing power capacity of 0.25 to properly analyze incoming network traffic. Each monitor sink is set to support an incoming traffic rate of up to 1 Gbps. As previously defined, SFC segments are monitored independently – that is, each segment (*i.e.*, $(i,j) \in L_q^S : \forall q \in Q$) is potentially monitored considering different requirements. We vary the network sampling requirements ($r_m^{q,i,j}$) of each segment (i,j) from 10^{-5} to 10^{-1} uniformly amongst all SFCs $q \in Q$. Each experiment is repeated 30 times considering different deployed service chains and physical infrastructures to ensure a confidence level of 90% or higher.

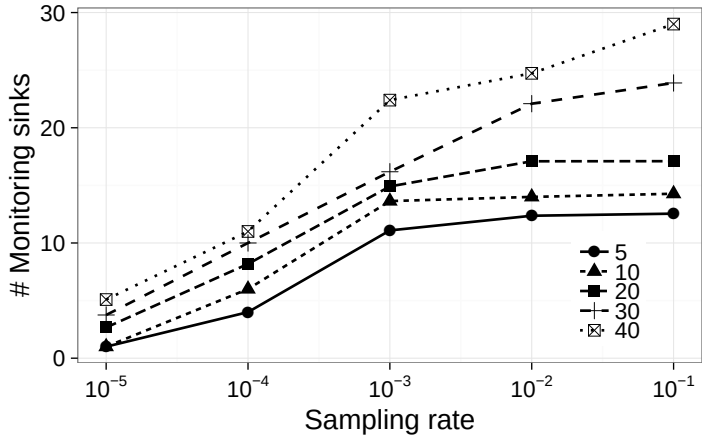
7.4.2 Results

First, we analyze the number of monitoring sinks needed to cope with an increasing sampling rate required by deployed SFCs. Figure 7.4 depicts the average number of instantiated monitoring sinks with sampling rates varying from 10^{-5} to 10^{-1} (in log scale). Each dashed line represents a scenario with an increasing number of simultaneously deployed SFCs (from 5 to 40). We observe that the higher the sampling rate (*i.e.*, towards rate 10^{-1}), the more monitoring sinks are deployed in the infrastructure – up to 30 in the worst case. In contrast, as

[4]http://www.cs.bu.edu/brite/

the sampling rate tends to be lower (*i.e.*, towards rate 10^{-5}), the number of sinks reduces dramatically – up to a factor of 6. The reason for this behavior is because as the sampling rate increases, a larger number of network packets is expected to be forwarded to monitoring sinks to further analysis. Observe that the baseline case (for the conducted evaluation) consists of forwarding all network traffic (or none of it) to monitoring sinks. By forwarding all network traffic to monitoring sinks, the DNM solution is infeasible for most of the considered scenarios. In such infeasible scenarios, the amount of demanded resources is far from meeting the available physical resources in the infrastructure. For instance, if we consider 40 deployed SFCs, each SFC having five potential segments to be monitored (of 1 Gbps), it would lead to a theoretical lowerbound of (at least) 200 instances of deployed sinks.

Figure 7.4: Number of monitoring sinks required for different sampling rates. For this evaluation, we consider $\alpha = 1$ and $\beta = 1$.

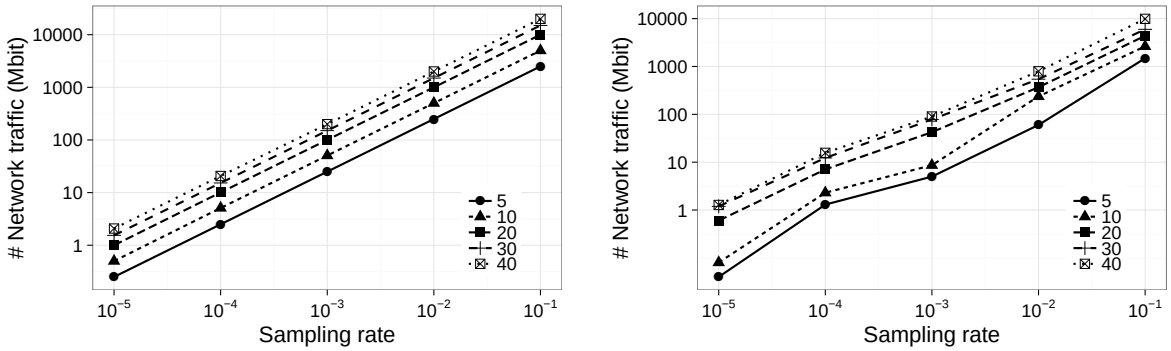


Source: by author (2017).

Next, we evaluate the volume of network monitoring traffic that is routed in the network infrastructure. Figure 7.5(a) depicts the aggregate amount of network monitoring traffic demanded by all SFC segments being monitored. Observe that higher sampling rates lead to a higher volume of network monitoring traffic – as it is expected. The observed behavior is linear since we are evaluating a set of homogeneous scenarios. In turn, Figure 7.5(b) illustrates the aggregated amount of network monitoring traffic that is routed through physical links in the network infrastructure in comparison to the demanded network monitoring traffic (illustrated in Figure 7.5(a)). The volume of network monitoring traffic steered in the infrastructure depends primarily *(i)* on the required sampling rate and *(ii)* on how distant monitoring sinks are placed from SFC segments and sampling devices. When a monitoring sink is instantiated on top of an N-PoP attached to the network device that is sampling the network traffic, there is no need to route network monitoring traffic through the physical infrastructure. In this case, the DNM solution leads to a substantial gain regarding the usage of physical resources. However, in most cases, due to the lack of available resources at specific network devices (*i.e.*, forwarding devices, physical links and/or monitoring sinks), the sampling of network traffic is often performed at

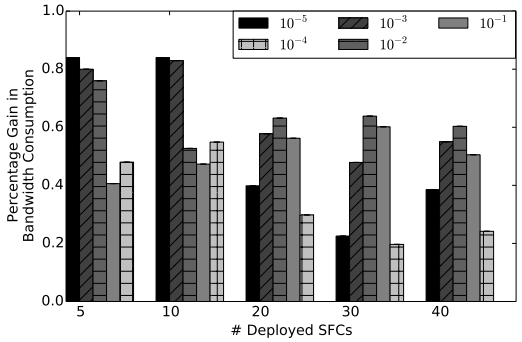
multiple forwarding devices (implying in network overheads). As an example, for the scenario with 10 SFCs using a sampling rate of 10^{-3}, it requires around 50Mbit of network monitoring traffic to be steered in the network infrastructure (illustrated in Figure 7.5(a)). For the same setting, using DNM, we observe a network consumption of 9Mbit. In order to clearly illustrate the benefits of DNM solutions, Figure 7.5(c) depicts the percentage gain in terms of network monitoring traffic that is saved from being routed in the physical infrastructure. In the figure, higher percentage gains are interpreted as lower overheads, as it leads to a reduced consumption of physical resources. The average observed gain is around 54% in relation to the total amount of demanded network monitoring traffic (illustrated in Figure 7.5(a)). Note that these gains might reach as much as 80% for lower sampling rates (10^{-5} and 10^{-4}), and from 20% to 50% for higher ones (10^{-1}). That happens mainly because the DNM aims at placing monitoring sinks (whenever possible) as close as possible to sampling devices. As mentioned, the closer they are placed, the fewer resources are consumed to steer network monitoring traffic to them.

Figure 7.5: Network monitoring traffic demanded, consumed and saved when applying DNM solutions.



(a) Aggregate amount of demanded network monitoring traffic.

(b) Average amount of consumed physical network resources.



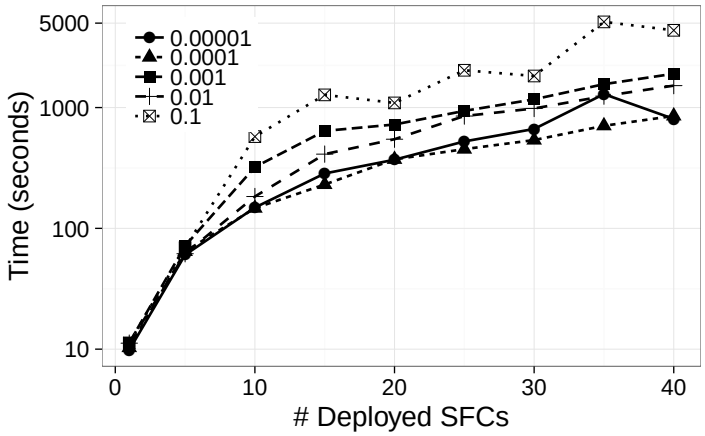
(c) Average amount of network resources that are saved when using DNM model.

Source: by author (2017).

Last, Figure 7.6 depicts the average time needed to find the optimal solution for different

DNM problem instances. We vary the sampling rate and the number of simultaneously deployed SFCs. We can observe that, even for very small instances, the required time to find a solution is non-negligible. For example, DNM instances having up to 20 SFCs, the required time spent by CPLEX is in the order of minutes (up to 1,000 sec.). As the number of SFCs considered increases (*i.e.* scenarios from 20 SFCs), the average required time to find the optimal solution reaches the order of hours (up to 5,200 sec.). Further, we also observe that there is a perceptible difference for the required computing time when we vary the sampling rates. For higher sampling rates, the required time tends to be lower in comparison to lower sampling rates. The reason for that is due to higher/lower volume of monitoring traffic to be steered in each scenario – which directly impacts on the hardness of finding the optimal solution to the DNM instance in hand.

Figure 7.6: Required time to compute the optimal solution for the DNM problem.



Source: by author (2017).

8 FINAL CONSIDERATIONS

In this chapter, we present the conclusions and contributions obtained with the work developed in the context of this thesis. We briefly present the publications and other achievements of this thesis.

8.1 Conclusions

While Network Function Virtualization (NFV) is increasingly gaining momentum, with promising benefits of flexible service function deployment and reduced operations and management costs, there are several challenges that remain to be properly tackled, so that it can realize its full potential. One of these challenges, which has a significant impact on the NFV production chain, is effectively and (cost) efficiently deploying service functions, while ensuring that service level agreements are satisfied and making wise allocations of network resources.

Amid various other aspects involved, Virtual Network Function Placement & Chaining (VNFPC) is key to fulfilling this challenge. VNFPC poses, however, an important trade-off between quality, efficiency, and scalability that previously proposed solutions (MOENS; TURCK, 2014; LEWIN-EYTAN et al., 2015; LUIZELLI et al., 2015) have failed to satisfy simultaneously. This is no surprise, though, given that the complexity of solving this problem is NP-complete, as we have demonstrated in Chapter 3.

In this thesis, we tackled the VNFPC problem in the context of Inter- and Intra-datacenter. As a major contribution to the state-of-the-art, we first formalized the Inter-datacenter VNFPC and proposed an optimization model to solve it (LUIZELLI et al., 2015). Our mathematical model has established one of the first baseline comparison in the field of resource management in NFV and has been widely used in the recent related literature. Further, we devised a novel Fix-and-Optimize based approach for efficiently solving the VNFPC problem for realistic and large instances of the problem (LUIZELLI et al., 2017a). Our proposed approach combines mathematical programming and meta-heuristic algorithms for systematically exploring the VNFPC solution space.

Then, we tackled the VNFPC problem in the context of Intra-datacenter deployments. First, we conducted an extensive set of experiments to better comprehend the effects of empirical environments on the performance of deployed network services in a typical NFV environment (LUIZELLI et al., 2017b). From this set of experiments, we developed a generalized cost function that accurately captures and estimates the CPU cost of software switching for arbitrary NFV setups. Then, we designed the Operational Cost Minimization (OCM) algorithm for the efficient and cost-oriented Intra-datacenter placement of service chains. OCM mechanism heavily relies on the usage of our previously defined CPU cost function to guide the search for optimal solutions with minimum operational cost.

Finally, after diving specifically into the provisioning of NFV services in both Inter- and

Intra-datacenter contexts, we tackled the problem of monitoring network traffic of service chains – another essential building block for the proper operation of NFV-based network infrastructures. For that, we formalized the Distributed Network Monitoring (DNM) problem and proposed a mathematical optimization model to effectively coordinate the monitoring of service chains in the context of NFV/SDN deployments.

In Chapter 1, we presented the following hypothesis based on the limitations of state-of-the-art proposals in the context of NFV resource management:

Hypothesis: *In order to take full advantage of the benefits provided by NFV, efficient and scalable strategies should be used to orchestrate the deployment of Service Function Chaining.*

Based on the work presented in this thesis, it is possible to identify evidence to answer the research questions associated with the hypothesis that has been posed to guide this study. The answer to each question is detailed as follows.

Research Question 1. *When performing the deployment of NFV-based SFCs, what are the gains regarding network metrics and resource consumption that can be attained in comparison to traditional network service deployments (i.e., based on physical network functions)?*

Answer. In this thesis, we formalized the Inter-datacenter VNFPC problem taking into account the main constraints involved in the process of deploying SFCs requests on top of NFV-based environments. Our proposed optimization model for the aforementioned problem minimizes the number of VNFs instantiated in the infrastructure while ensuring quality related metrics such as end-to-end delay. We evaluated our approach to the VNFPC problem considering realistic workloads and different use cases. Results show that solutions for the problem lead to a reduction of up to 25% in end-to-end delays and an acceptable resource over-provisioning limited to 4% in comparison to traditional middlebox-based deployments.

Research Question 2. *Since the SFCP problem is NP-complete, how to provide efficient and near-optimal solutions to SFPC instances on large-scale NFV infrastructures?*

Answer. We proposed a novel fix-and-optimized based heuristic which is able to timely solve large instances of the VNFPC problem – in the order of a couple of hours in the worst case. The results achieved not only evidenced the potentialities of fix-and-optimize and Variable Neighborhood Search as building blocks for systematically exploring the VNFPC solution space. More importantly, they have shown the significant improvement of our approach over the state-of-the-art – considering the quality (500% better solutions on average), efficiency (in the order of a couple of hours in the worst case), and scalability (to the order of thousands NFV nodes).

Research Question 3. *On deploying SFCs onto physical servers, there is a non-negligible CPU cost associated with the service operation. Which are these costs and how to properly estimate them in an NFV environment?*

Answer. In NFV-based environments (and any virtualization-intense one), virtual switching is an essential functionality that provides isolation, scalability, and mainly flexibility to the environment. Such functionalities provided by virtual switching introduces (in general) a non-negligible operational cost, which makes much harder to guarantee a reasonable level of network performance to network services – considered a key requirement for the success operation of NFV. To fully grasp the costs and limitation of typical NFV deployments, we conducted an extensive experimental evaluation measuring the performance and analyzing the impact of virtual switching (*i.e.*, Open vSwitch). Our results indicate that the operational cost of deploying service chains depends on the installed Open vSwitch (either kernel OvS or DPDK-OvS), the required amount of traffic to process, the length of the service chain, and the placement strategy. Based on such evaluation, we developed a generalized cost function that accurately captures the CPU cost of software switching in this setting. Our cost function is based on logarithm regression and is able to estimate the operational cost to any arbitrary input parameters. By fully understanding incurred costs and potential limitations in NFV environments, we were able to design tailored solutions for the management and orchestration of VNFs.

Research Question 4. *How to minimize operational costs in SFC deployments? Is it possible to efficiently guide NFV orchestrators on deploying VNFs Intra- and Inter-server?*

Answer. Understanding the costs and potential limitations of software switching in NFV environments is a key ingredient in the ability to design efficient solutions for VNF management and orchestration – possibly leading to lower operational costs. We proposed OCM algorithm which relies on the usage of the previously defined CPU cost function. OCM design relies on the well-known *minimum cost matching* problem in order to efficiently (*i.e.*, in polynomial time) place a service chain into a set of physical servers with minimum cost to the NFV provider. The obtained results show that the proposed algorithm can reduce the operational costs (*i.e.*, additional resources consumption of the hypervisor) significantly (up to a factor of 4 in the extreme case and 20% – 40% in typical cases) when compared to traditional deployment policies implemented in OpenStack. Moreover, OCM makes a better usage of the available resources – which contributes to a higher acceptance ratio of new network services overtime.

Research Question 5. *The proper operation of network services requires to constantly monitor them. Is it possible to take advantage of NFV/SDN technologies to efficiently deploy virtualized monitoring services?*

Answer. We formalized the Distributed Network Monitoring (DNM) problem in order to monitor network traffic of service chains. DNM is able to effectively coordinate the monitoring of service chains in the context of NFV/SDN deployments. By coordinating the placement of monitoring sinks (also referred as monitoring functions) and the efficient forwarding of network traffic to them, DNM can outperform traditional monitoring mechanism. Results for our proposed optimal model indicate a reduction of up to 80% in network monitoring traffic that is routed through the network infrastructure, while placing a reasonable number of monitoring sinks.

8.2 Future Research Directions

As NFV is a recent network paradigm and yet in a maturing phase, there are many open research problems that were out of the scope of this thesis. Throughout this thesis, we have focused on the VNFPC problem in the context of Inter- and -Intra-datacenter. In this realm, we consider extending this thesis in the following manner.

First, we plan to evaluate the impact of complex SFC request (*e.g.*, non-linear) on deploying them both in Inter- and Intra-datacenter scenarios. This is particularly important since VNFs can be decomposed into independent subcomponents – potentially leading to even more complex requests. *Second*, we plan to study other operational costs that were not covered in this thesis – such as energy consumption – and its correlation to current performance limitations. The interplay between different operational costs can open up opportunities to design even more accurate pricing model by NFV providers. *Third*, we intend to integrate our devised models/algorithms in OpentStack scheduler and measure its performance in a real NFV-based deployment. For that purpose, there are many existing technological limitations that should be tackled in advance. For instance, it is well known the current limitation of OpenStack regarding its computing (responsible for managing VNFs) and networking (responsible for managing the routing) modules. In order to instantiate an SFC in an NFV environment, these two modules should be further integrated. *Fourth*, we plan to extend our proposed solutions in order to cope with online scheduling of SFC requests. The scheduling of SFCs is another cornerstone to the effective success of NFV. We envision that, in a near future, NFV orchestrators will be able to instantiate a network service within a few milliseconds – which would allow, for instance, a network service to be provisioned per customer or yet (even more challenging) per network flow. *Fifth*, we intend to extend our approach to deal with constantly evolving network conditions. In such cases, assigned composite services need to be reorganized (reassigned) in response to fluctuations in traffic demand (for example), so that service level agreements are not violated. *Lastly*, we plan to extend our conducted evaluation in order to fully understand the impact of

different parameters in our models, as well as to design efficient and online strategies to cope with highly dynamic scenarios.

8.3 Achievements

The development of this thesis has led to the publication of the following peer-reviewed/journal papers:

- **Piecing Together the NFV Provisioning Puzzle: Efficient Placement and Chaining of Virtual Network Functions. (Best Student Paper Award)**
  Marcelo Caggiani Luizelli, Leonardo Richter Bays, Marinho Pilla Barcellos, Luciana Salete Buriol e Luciano Paschoal Gaspary.
  IFIP/IEEE International Symposium on Integrated Network Management 2015 (IM 2015).
- **A Fix-and-Optimize Approach for Efficient and Large Scale Virtual Network Function Placement and Chaining.**
  Marcelo Caggiani Luizelli, Weverton Luis da Costa Cordeiro, Luciana Salete Buriol e Luciano Paschoal Gaspary.
  Elsevier Computer Communications, 2016.
- **The Actual Cost of Software Switching for NFV Chaining**
  Marcelo Caggiani Luizelli, Danny Raz, Yaniv Saar e Jose Yallouz.
  IFIP/IEEE International Symposium on Integrated Network Management 2017 (IM 2017).

In addition to the aforementioned main outcomes of this thesis, we further authored/coauthored some others studies on correlated optimization problems in the context of NFV, SDN, and Network Virtualization. These publications are listed next.

- **Constant Time Updates in Hierarchical Heavy Hitters**
  Ran Basat, Gil Enziger, Roy Friedman, Marcelo Caggiani Luizelli, Erez Waisbard.
  ACM SIGCOMM, 2017.
- **Constant Time Weighted Frequency Estimation for Virtual Network Functionalities**
  Gil Enziger, Marcelo Caggiani Luizelli, Erez Waisbard.
  IEEE International Conference on Computer Communications and Networks (ICCCN), 2017.
- **NFV-PEAR: Posicionamento e Encadeamento Adaptativo de Funções Virtuais de Rede**
  Gustavo Miotto, Marcelo Caggiani Luizelli, Weverton Luis da Costa Cordeiro e Luciano Paschoal Gaspary.
  Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC), 2017.
- **How physical network topologies affect virtual network embedding quality: A characterization study based on ISP and datacenter networks**

Marcelo Caggiani Luizelli, Leonardo Richter Bays, Marinho Pilla Barcellos, Luciana Salete Buriol e Luciano Paschoal Gaspary.
Journal of Network and Computer Applications, 2016.

- **QoS Aware Schedulers for Multi-users on OFDMA Downlink: Optimal and Heuristic.**
  Matheus Cadori Nogueira, Marcelo Caggiani Luizelli, Samuel Marini, Cristiano Both, Juergen Rochol; Armando Ordonez e Oscar Caicedo.
  IEEE Latin-American Conference on Communications, 2016.
- **HIPER: Heuristic-based Infrastructure Expansion through Partition Reconnection for Efficient Virtual Network Embedding.**
  Marcelo Caggiani Luizelli, Leonardo Richter Bays, Marinho Pilla Barcellos e Luciano Paschoal Gaspary.
  IFIP/IEEE/ACM International Conference on Network and Service Management (CNSM), 2014.
- **Survivor: an Enhanced Controller Placement Strategy for Improving SDN Survivability**
  Lucas Fernando Muller, Rodrigo Ruas de Oliveira, Marcelo Caggiani Luizelli, Luciano Paschoal Gaspary e Marinho Pilla Barcellos.
  IEEE Global Communications Conference (GLOBECOM), 2014.
- **Reconectando Partições de Infraestruturas Físicas: Rumo a uma Estratégia de Expansão para o Mapeamento Eficiente de Redes Virtuais**
  Marcelo Caggiani Luizelli, Leonardo Richter Bays, Marinho Pilla Barcellos, Luciana Salete Buriol e Luciano Paschoal Gaspary.
  Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC), 2014.

REFERENCES

AHUJA, R. K.; MAGNANTI, T. L.; ORLIN, J. B. **Network flows - theory, algorithms and applications**. New Jersey, USA: Prentice Hall, 1993. ISBN 978-0-13-617549-0.

ALBERT, R.; BARABÁSI, A.-L. Topology of evolving networks: Local events and universality. **Physical Review Letters**, American Physical Society, v. 85, p. 5234 – 5237, Dec 2000.

BARI, M. F.; CHOWDHURY, S. R.; AHMED, R.; BOUTABA, R. On orchestrating virtual network functions. In: INTERNATIONAL CONFERENCE ON NETWORK AND SERVICE MANAGEMENT. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2015. (CNSM '15), p. 50–56.

BARKAI, S.; KATZ, R.; FARINACCI, D.; MEYER, D. Software defined flow-mapping for scaling virtualized network functions. In: ACM SIGCOMM WORKSHOP ON HOT TOPICS IN SOFTWARE DEFINED NETWORKING. **Proceedings...** New York, NY, USA: ACM, 2013. (ACM SIGCOMM HotSDN'13), p. 149–150.

BASTA, A.; KELLERER, W.; HOFFMANN, M.; MORPER, H. J.; HOFFMANN, K. Applying nfv and sdn to lte mobile core gateways, the functions placement problem. In: WORKSHOP ON ALL THINGS CELLULAR: OPERATIONS, APPLICATIONS AND CHALLENGES. **Proceedings...** New York, NY, USA: ACM, 2014. (AllThingsCellular '14), p. 33–38.

BEN-BASAT, R.; EINZIGER, G.; FRIEDMAN, R.; KASSNER, Y. Heavy hitters in streams and sliding windows. In: IEEE INFOCOM. **Proceedings...** Piscataway, NJ, USA: IEEE Press, 2016. (INFOCOM '16), p. 1–9.

BEN-BASAT, R.; EINZIGER, G.; FRIEDMAN, R.; LUIZELLI, M. C.; WAISBARD, E. Constant time updates in hierarchical heavy hitters. In: ACM SIGCOMM CONFERENCE ON DATA COMMUNICATION. **Proceedings...** New York, NY, USA: ACM, 2017. (SIGCOMM '17), p. 1–14.

BENSON, T.; AKELLA, A.; SHAIKH, A. Demystifying configuration challenges and trade-offs in network-based isp services. In: ACM SIGCOMM CONFERENCE ON DATA COMMUNICATION. **Proceedings...** New York, NY, USA: ACM, 2011. (SIGCOMM '11), p. 302–313.

BEREND, D.; TASSA, T. Improved bounds on bell numbers and on moments of sums of random variables. **Probability and Mathematical Statistics**, v. 30, n. 2, p. 185–205, 2010.

BERNAL, M. V.; CERRATO, I.; RISSO, F.; VERBEIREN, D. Transparent optimization of inter-virtual network function communication in open vswitch. In: INTERNATIONAL CONFERENCE ON CLOUD NETWORKING. **Proceedings...** Piscataway, NJ, USA: IEEE Press, 2016. (Cloudnet '16), p. 76–82.

BHAMARE, D.; JAIN, R.; SAMAKA, M.; ERBAD, A. A survey on service function chaining. **Journal of Network and Computer Applications**, v. 75, p. 138–155, 2016.

BONELLI, N.; PIETRO, A. D.; GIORDANO, S.; PROCISSI, G. On multi gigabit packet capturing with multi core commodity hardware. In: INTERNATIONAL CONFERENCE ON PASSIVE AND ACTIVE MEASUREMENT. **Proceedings...** Berlin, Heidelberg: Springer-Verlag, 2012. (PAM'12), p. 64–73.

BOSSHART, P.; DALY, D.; GIBB, G.; IZZARD, M.; MCKEOWN, N.; REXFORD, J.; SCHLESINGER, C.; TALAYCO, D.; VAHDAT, A.; VARGHESE, G.; WALKER, D. P4: Programming protocol-independent packet processors. **SIGCOMM Computer Communication Review**, ACM, New York, NY, USA, v. 44, n. 3, p. 87–95, jul 2014.

BOUET, M.; LEGUAY, J.; CONAN, V. Cost-based placement of vdpi functions in nfv infrastructures. In: CONFERENCE ON NETWORK SOFTWARIZATION. **Proceedings...** Piscataway, NJ, USA: IEEE Press, 2015. (NetSoft'15), p. 1–9.

CHOI, B.-Y.; MOON, S.; ZHANG, Z.-L.; PAPAGIANNAKI, K.; DIOT, C. Analysis of point-to-point packet delay in an operational network. **Computer Networks**, Elsevier North-Holland, Inc., New York, NY, USA, v. 51, p. 3812–3827, 2007.

CISCO. **Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2015-2020**. 2016. Available at: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html. Visited on: Jan. 30, 2016.

CLAISE, B. **Cisco Systems NetFlow Services Export Version 9**. 2004. Available at: https://www.ietf.org/rfc/rfc3954.txt. Accessed on: Jun. 1, 2016.

CLAYMAN, S.; MAINI, E.; GALIS, A.; MANZALINI, A.; MAZZOCCA, N. The dynamic placement of virtual network functions. In: IEEE NETWORK OPERATIONS AND MANAGEMENT SYMPOSIUM. **Proceedings...** Piscataway, NJ, USA: IEEE Press, 2014. (NOMS'14), p. 1–9.

CORBET, J.; RUBINI, A.; KROAH-HARTMAN, G. **Linux device drivers – where the Kernel meets the hardware**. Sebastopol, EUA: O'Reilly, 2005.

CORPORATION, M. **Sockperf Traffic Generator**. 2016. Available at: https://github.com/Mellanox/sockperf/. Accessed on: Oct. 20, 2015.

DOBRESCU, M.; ARGYRAKI, K.; RATNASAMY, S. Toward predictable performance in software packet-processing platforms. In: USENIX CONFERENCE ON NETWORKED SYSTEMS DESIGN AND IMPLEMENTATION. **Proceedings...** San Jose, CA: USENIX, 2012. (NSDI'12), p. 141–154.

EMMERICH, P.; GALLENMüLLER, S.; RAUMER, D.; WOHLFART, F.; CARLE, G. Moongen: A scriptable high-speed packet generator. In: ACM INTERNET MEASUREMENT CONFERENCE. **Proceedings...** New York, NY, USA: ACM, 2015. (IMC '15), p. 275–287.

GAREY, M. R.; JOHNSON, D. S. **Computers and Intractability: A Guide to the Theory of NP-Completeness**. New York, NY, USA: W. H. Freeman, 1979.

GEMBER-JACOBSON, A.; VISWANATHAN, R.; PRAKASH, C.; GRANDL, R.; KHALID, J.; DAS, S.; AKELLA, A. Opennf: Enabling innovation in network function control. In: ACM CONFERENCE ON COMPUTER COMMUNICATIONS. **Proceedings...** New York, NY, USA: ACM, 2014. (ACM SIGCOMM'14), p. 163–174.

GHAZNAVI, M.; KHAN, A.; SHAHRIAR, N.; ALSUBHI, K.; AHMED, R.; BOUTABA, R. Elastic virtual network function placement. In: IEEE INTERNATIONAL CONFERENCE ON CLOUD NETWORKING. **Proceedings...** Piscataway, NJ, USA: IEEE Press, 2015. (CloudNet'15), p. 255–260.

GROUP, N. F. I. S. **Network Function Virtualisation (NFV): An Introduction, Benefits, Enablers, Challenges and Call for Action**. 2012. 1–16 p. Available at: https://portal.etsi.org/nfv/. Accessed on: Jan. 20, 2015.

HALPERN, J.; PIGNATARO, C. **Service Function Chaining (SFC) Architecture**. 2015. Available at: https://datatracker.ietf.org/doc/draft-ietf-sfc-architecture/. Accessed on: Dez. 1, 2015.

HAN, B.; GOPALAKRISHNAN, V.; JI, L.; LEE, S. Network function virtualization: Challenges and opportunities for innovations. **IEEE Communications Magazine**, v. 53, n. 2, p. 90–97, Feb 2015.

HANSEN, P.; MLADENOVIć, N. Variable neighborhood search: Principles and applications. **European Journal of Operational Research**, v. 130, n. 3, p. 449–467, 2001.

HELBER, S.; SAHLING, F. A fix-and-optimize approach for the multi-level capacitated lot sizing problem. **International Journal of Production Economics**, v. 123, n. 2, p. 247–256, 2010.

HERRERA, J. G.; BOTERO, J. F. Resource allocation in nfv: A comprehensive survey. **IEEE Transactions on Network and Service Management**, v. 13, n. 3, p. 518–532, Sept 2016.

HWANG, J.; RAMAKRISHNAN, K. K.; WOOD, T. Netvm: High performance and flexible networking using virtualization on commodity platforms. In: USENIX CONFERENCE ON NETWORKED SYSTEMS DESIGN AND IMPLEMENTATION. **Proceedings...** Oakland, CA: USENIX Association, 2014. (NSDI'14), p. 34–47.

HWANG, J.; RAMAKRISHNAN, K. K.; WOOD, T. Netvm: High performance and flexible networking using virtualization on commodity platforms. **IEEE Transactions on Network and Service Management**, v. 12, n. 1, p. 34–47, March 2015.

INTEL. **Intel DPDK API**. 2016. Available at: http://dpdk.org/. Visited on: May 19, 2016.

INTEL. **Intel Open Network Platform Release 2.1: Performance Test Report**. 2016. Available at: https://01.org/packet-processing/intel/textregistered-onp. Visited on: Jun. 15, 2016.

KELLEHER, J.; O'SULLIVAN, B. **Generating All Partitions: A Comparison Of Two Encodings**. 2009. Available at: https://arxiv.org/abs/0909.2331. Visited on: Jul. 15, 2016.

KUO, T.; LIOU, B.; LIN, J.; TSAI, M. Deploying chains of virtual network functions: On the relation between link and server usage. In: IEEE INTERNATIONAL CONFERENCE ON COMPUTER COMMUNICATIONS. **Proceedings...** San Francisco, USA: IEEE Press, 2016. (INFOCOM'16), p. 1–9.

L. Deri. **Direct NIC Access**. 2016. Available at: http://www.ntop.org/products/packet-capture/pf_ring. Accessed on: Jun. 1, 2016.

LEPERS, B.; QUÉMA, V.; FEDOROVA, A. Thread and memory placement on numa systems: Asymmetry matters. In: USENIX CONFERENCE ON USENIX ANNUAL TECHNICAL CONFERENCE. **Proceedings...** Berkeley, CA, USA: USENIX Association, 2015. (USENIX ATC'15), p. 277–289.

LEWIN-EYTAN, L.; NAOR, J.; COHEN, R.; RAZ, D. Near optimal placement of virtual network functions. In: IEEE INFOCOM. **Proceedings...** New York, NY, USA: IEEE, 2015. (INFOCOM'15), p. 1346–1354.

LUIZELLI, M. C.; BAYS, L. R.; BURIOL, L. S.; BARCELLOS, M. P.; GASPARY, L. P. Piecing together the nfv provisioning puzzle: Efficient placement and chaining of virtual network functions. In: IFIP/IEEE INTERNATIONAL SYMPOSIUM ON INTEGRATED NETWORK MANAGEMENT. **Proceedings...** New York, NY, USA: IEEE, 2015. (IM'15), p. 98–106.

LUIZELLI, M. C.; BAYS, L. R.; BURIOL, L. S.; BARCELLOS, M. P.; GASPARY, L. P. How physical network topologies affect virtual network embedding quality: A characterization study based on {ISP} and datacenter networks. **Journal of Network and Computer Applications**, v. 70, p. 1 – 16, 2016.

LUIZELLI, M. C.; CORDEIRO, W. L. da C.; BURIOL, L. S.; GASPARY, L. P. A fix-and-optimize approach for efficient and large scale virtual network function placement and chaining. **Computer Communications**, v. 102, p. 67 – 77, 2017.

LUIZELLI, M. C.; RAZ, D.; SAAR, Y.; YALLOUZ, J. The actual cost of software switching for nfv chaining. In: IFIP/IEEE INTERNATIONAL SYMPOSIUM ON INTEGRATED NETWORK MANAGEMENT. **Proceedings...** New York, NY, USA: IEEE, 2017. (IM'17), p. 335–343.

LUKOVSZKI, T.; ROST, M.; SCHMID, S. It's a match!: Near-optimal and incremental middlebox deployment. **SIGCOMM Computer Communication Review**, v. 46, n. 1, p. 30–36, jan. 2016.

LUKOVSZKI, T.; SCHMID, S. Online admission control and embedding of service chains. In: INTERNATIONAL COLLOQUIUM ON STRUCTURAL INFORMATION AND COMMUNICATION COMPLEXITY. **Proceedings...** New York, USA: Springer-Verlag, 2015. (SIROCCO'15).

MARTINS, J.; AHMED, M.; RAICIU, C.; OLTEANU, V.; HONDA, M.; BIFULCO, R.; HUICI, F. Clickos and the art of network function virtualization. In: USENIX CONFERENCE ON NETWORKED SYSTEMS DESIGN AND IMPLEMENTATION. **Proceedings...** Berkeley, CA, USA: USENIX Association, 2014. (NSDI'14).

MCKEOWN, N.; ANDERSON, T.; BALAKRISHNAN, H.; PARULKAR, G.; PETERSON, L.; REXFORD, J.; SHENKER, S.; TURNER, J. Openflow: Enabling innovation in campus networks. **SIGCOMM Computer Communication Review**, ACM, New York, NY, USA, v. 38, n. 2, p. 69–74, mar. 2008.

MECHTRI, M.; GHRIBI, C.; SOUALAH, O.; ZEGHLACHE, D. Nfv orchestration framework addressing sfc challenges. **IEEE Communications Magazine**, v. 55, n. 6, p. 16–23, Jun. 2017.

MEHRAGHDAM, S.; KELLER, M.; KARL, H. Specifying and placing chains of virtual network functions. In: IEEE INTERNATIONAL CONFERENCE ON CLOUD NETWORKING. **Proceedings...** Piscataway, NJ, USA: IEEE Press, 2014. (CloudNet'14), p. 7–13.

MERCA, M. Fast algorithm for generating ascending compositions. **Journal of Mathematical Modelling and Algorithms**, Springer-Verlag, Berlin, Heidelberg, v. 11, n. 1, p. 89–104, 2012.

MIJUMBI, R.; SERRAT, J.; GORRICHO, J. L.; BOUTEN, N.; TURCK, F. D.; BOUTABA, R. Network function virtualization: State-of-the-art and research challenges. **IEEE Communications Surveys Tutorials**, v. 18, n. 1, p. 236–262, Jun 2016.

MOENS, H.; TURCK, F. D. Vnf-p: A model for efficient placement of virtualized network functions. In: INTERNATIONAL CONFERENCE ON NETWORK AND SERVICE MANAGEMENT. **Proceedings...** Piscataway, NJ, USA: IEEE Press, 2014. (CNSM'14 MiniConf), p. 418–423.

ONF. **OpenStack**. 2015. Available at: http://www.openstack.org. Visited on: Jul. 30, 2016.

ONF. **Open vSwitch**. 2016. Available at: http://www.openvswitch.org. Visited on: Jul. 30, 2016.

PFAFF, B.; PETTIT, J.; KOPONEN, T.; AMIDON, K.; CASADO, M.; SHENKER, S. Extending networking into the virtualization layer. In: ACM WORKSHOP ON HOT TOPICS IN NETWORKS. **Proceedings...** New York, NY, USA: ACM, 2009. (HotNets'09), p. 1–6.

PFAFF, B.; PETTIT, J.; KOPONEN, T.; JACKSON, E.; ZHOU, A.; RAJAHALME, J.; GROSS, J.; WANG, A.; STRINGER, J.; SHELAR, P.; AMIDON, K.; CASADO, M. The design and implementation of open vswitch. In: USENIX SYMPOSIUM ON NETWORKED SYSTEMS DESIGN AND IMPLEMENTATION. **Proceedings...** Oakland, CA: USENIX Association, 2015. (NSDI'15), p. 117–130.

PHAAL, P.; PANCHEN, S.; PANCHEN, S. **InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks**. 2001. Available at: https://www.ietf.org/rfc/rfc3176.txt. Accessed on: Jan. 1, 2016.

QUINN, P.; ELZUR., U. **Network Service Header**. 2016. Available at: https://datatracker.ietf.org/doc/draft-ietf-sfc-nsh/ . Accessed on: Oct. 1, 2016.

RANKOTHGE, W.; MA, J.; LE, F.; RUSSO, A.; LOBO, J. Towards making network function virtualization a cloud computing service. In: IFIP/IEEE INTERNATIONAL SYMPOSIUM ON INTEGRATED NETWORK MANAGEMENT. **Proceedings...** Piscataway, NJ, USA: IEEE Press, 2015. (IM'15), p. 89–97.

RIZZO, L. Netmap: A novel framework for fast packet i/o. In: USENIX CONFERENCE ON ANNUAL TECHNICAL CONFERENCE. **Proceedings...** Berkeley, CA, USA: USENIX Association, 2012. (ATC'12), p. 9–21.

RIZZO, L.; LETTIERI, G. Vale, a switched ethernet for virtual machines. In: INTERNATIONAL CONFERENCE ON EMERGING NETWORKING EXPERIMENTS AND TECHNOLOGIES. **Proceedings...** New York, NY, USA: ACM, 2012. (CoNEXT'12), p. 61–72.

ROST, M.; SCHMID, S. **Service Chain and Virtual Network Embeddings: Approximations using Randomized Rounding**. 2016. Available at: https://arxiv.org/abs/1604.02180 . Accessed on: Apr. 15, 2016.

SAHLING, F.; BUSCHKüHL, L.; TEMPELMEIER, H.; HELBER, S. Solving a multi-level capacitated lot sizing problem with multi-period setup carry-over via a fix-and-optimize heuristic. **Computers & Operations Research**, v. 36, n. 9, p. 2546 – 2553, 2009.

SEKAR, V.; EGI, N.; RATNASAMY, S.; REITER, M. K.; SHI, G. Design and implementation of a consolidated middlebox architecture. In: USENIX CONFERENCE ON NETWORKED SYSTEMS DESIGN AND IMPLEMENTATION. **Proceedings...** Berkeley, CA, USA: USENIX Association, 2012. (NSDI'12), p. 24.

SHAHBAZ, M.; CHOI, S.; PFAFF, B.; KIM, C.; FEAMSTER, N.; MCKEOWN, N.; REXFORD, J. Pisces: A programmable, protocol-independent software switch. In: ACM SIGCOMM CONFERENCE. **Proceedings...** New York, NY, USA: ACM, 2016. (SIGCOMM '16), p. 525–538.

YU, M. Programmable measurement architecture. In: ACM/IEEE SYMPOSIUM ON ARCHITECTURES FOR NETWORKING AND COMMUNICATIONS SYSTEMS. **Proceedings...** [S.l.], 2014. (ANCS'14).

YU, M.; JOSE, L.; MIAO, R. Software defined traffic measurement with opensketch. In: USENIX CONFERENCE ON NETWORKED SYSTEMS DESIGN AND IMPLEMENTATION. **Proceedings...** Berkeley, CA, USA: USENIX Association, 2013. (NSDI'13), p. 29–42.

YU, M.; YI, Y.; REXFORD, J.; CHIANG, M. Rethinking virtual network embedding: Substrate support for path splitting and migration. **ACM SIGCOMM Computer Communication Review (CCR)**, ACM, New York, NY, USA, v. 38, n. 2, p. 17–29, mar. 2008. ISSN 0146-4833.

ZHANG, W.; HWANG, J.; RAJAGOPALAN, S.; RAMAKRISHNAN, K.; WOOD, T. Flurries: Countless fine-grained nfs for flexible per-flow customization. In: INTERNATIONAL ON CONFERENCE ON EMERGING NETWORKING EXPERIMENTS AND TECHNOLOGIES. **Proceedings...** New York, NY, USA: ACM, 2016. (CoNEXT'16), p. 3–17.

ZHOU, D.; FAN, B.; LIM, H.; KAMINSKY, M.; ANDERSEN, D. G. Scalable, high performance ethernet forwarding with cuckooswitch. In: **Proceedings...** New York, NY, USA: ACM, 2013. (CoNEXT'13), p. 97–108.

ZHU, Y.; AMMAR, M. Algorithms for assigning substrate network resources to virtual network components. In: INTERNATIONAL CONFERENCE ON COMPUTER COMMUNICATIONS. **Proceedings...** Piscataway, NJ, USA: IEEE Press, 2006. (INFOCOM'06), p. 1–12.

ZHU, Y.; KANG, N.; CAO, J.; GREENBERG, A.; LU, G.; MAHAJAN, R.; MALTZ, D.; YUAN, L.; ZHANG, M.; ZHAO, B. Y.; ZHENG, H. Packet-level telemetry in large datacenter networks. In: **Proceedings...** New York, NY, USA: ACM, 2015. (SIGCOMM '15), p. 479–491.

APPENDIX A RESUMO ESTENDIDO DA TESE

Middleboxes (ou funções de rede - NF) desempenham um papel essencial nas redes de computadores, uma vez que fornecem um conjunto diversificado de funcionalidades que abrangem desde aspectos de segurança (por exemplo, firewall e detecção de intrusão) a questões de desempenho (por exemplo, caching e proxying) (MARTINS et al., 2014). Conforme implementado, os middleboxes são difíceis de implantar e operar. Isto se dá principalmente devido aos procedimentos que precisam ser seguidos, como lidar com uma ampla variedade de interfaces hardware customizadas e o encadeamento manual NFs para garantir o comportamento desejado de um serviço de rede. Além disso, estudos evidenciam que o número de middleboxes presente em redes empresariais (assim como nas redes de datacenter e ISP) é similar ao número de dispositivos de encaminhamento (BENSON; AKELLA; SHAIKH, 2011; SEKAR et al., 2012). Desta forma, as dificuldades acima mencionadas são exacerbadas pela complexidade imposta pelo alto número de funções de rede que um provedor de rede tem que gerencia – o que diretamente leva ao aumento dos custos operacionais. Para além dos custos relacionados à implantação e com o encadeamento das NFs, a frequente necessidade de atualizações de hardware aumenta substancialmente os investimentos necessários para operacionalizar serviços de rede.

A virtualização de funções de rede (NFV) é um novo paradigma em redes de computadores que foi proposto para migrar o processamento de NFs de dispositivos de hardware especializados para software executado em hardware de prateleira (GROUP, 2012). Além de potencialmente reduzir os custos de aquisição e manutenção de equipamentos, espera-se que o paradigma NFV permita que os provedores de rede aproveitem os benefícios da virtualização no gerenciamento de funções de rede (por exemplo, elasticidade, desempenho, flexibilidade, etc.). Neste contexto, as Redes Definidas por Software (Software-Defined Networking – SDN) podem ser consideradas uma tecnologia complementar conveniente, que, se disponível, tem o potencial de facilitar o encadeamento das funções de rede acima mencionadas. Na verdade, o paradigma SDN tem potencial para facilitar problema de encadeamento de funções de rede (Service Function Chaining – SFC).

Em suma, o problema de encadeamento de NFs consiste em garantir que os fluxos de rede sejam encaminhados de maneira eficiente através de caminhos que passem por um dado conjunto de middleboxes (MECHTRI et al., 2017). No contexto de redes NFV/SDN e considerando a flexibilidade oferecida por estes ambientes, o problema consiste em definir de forma otimizada quantas instâncias de funções de rede virtual (Virtual Network Function – VNF) são necessárias e onde posicioná-las na infraestrutura. Além disso, o problema consiste em determinar caminhos fim-a-fim em que os fluxos de rede devem ser encaminhados de modo a passar pelas funções de rede necessárias.

A.1 Definição do Problema

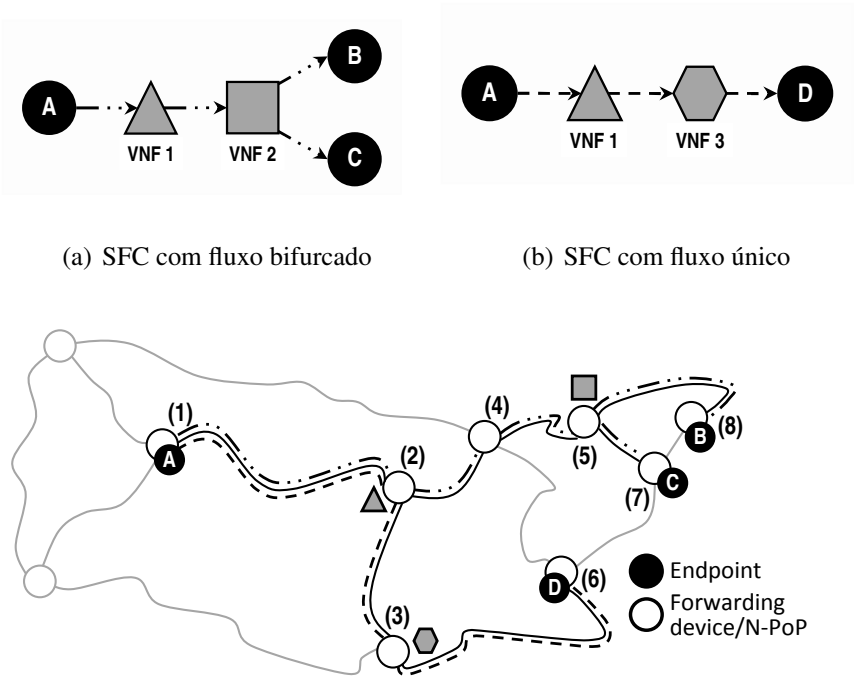
Nos últimos anos, houve significativos avanços em NFV, abordando principalmente aspectos relacionados ao planejamento e implantação (LEWIN-EYTAN et al., 2015; KUO et al., 2016; LUIZELLI et al., 2017a) e a operação e gerenciamento de tais ambientes (HWANG; RAMAKRISHNAN; WOOD, 2014; GEMBER-JACOBSON et al., 2014; ZHANG et al., 2016). No entanto, NFV é um paradigma relativamente novo e ainda em faze de amadurecimento, com várias questões de pesquisa em aberto. Como mencionado, um dos problemas mais desafiantes consiste em posicionar e encadear VNF de maneira eficiente.

Este problema é particularmente desafiador por várias razões. Primeiro, NFV é um paradigma de rede inerentemente distribuído baseado em nós com poder de computação (por exemplo, *small clouds*) espalhados pela infraestrutura. Portanto, dependendo de como os VNFs estão posicionadas e encadeadas na infraestrutura, os atrasos fim-a-fim observados em um serviço de rede podem tornar-se intoleráveis. Esse problema é agravado pelo fato de que os tempos de processamento tendem a ser maiores devido ao uso da virtualização e podem variar, dependendo do tipo de função de rede e da configuração de hardware do dispositivo que o hospeda. Em segundo lugar, mesmo quando as VNFs estejam implementadas em um único datacenter, os serviços de rede podem frequentemente enfrentar degradação de desempenho em métricas de rede (como, por exemplo, vazão, latência e jitter), dependendo de como as funções de rede estão encadeadas e implantadas em dispositivos físicos. Em terceiro lugar, a alocação de recursos deve ser realizada de maneira econômica, evitando o desperdício de recursos físicos. Por conseguinte, o posicionamento de funções de rede e o o encadeamento dos fluxos representam um passo essencial para permitir o uso de NFV em ambientes de produção. Nos parágrafos seguintes, fornece-se uma visão geral dos problemas de posicionamento e encadeamento da função de rede virtual (VNFPC) abordados nesta tese.

Posicionamento e Encadeamento de Funções Virtualizadas de Rede no contexto Inter-datacenter. Ilustra-se na Figura A.1 o problema de posicionar e encadear funções virtualizadas de rede no contexto inter-datacenter. O problema envolve a implantação de duas solicitações de SFCs. Para a primeira requisição, os fluxos de entrada devem ser transmitidos para uma instância da função de rede virtual (VNF) 1 (por exemplo, um firewall) e, em seguida, para a VNF 2 (por exemplo, um balanceador de carga). A segunda requisição especifica que os fluxos de entrada também devem ser encaminhados para a VNF 1 e, em seguida, pela VNF 3 (por exemplo, um proxy). Ambos as SFCs são ilustradas nas Figuras A.1(a) e A.1(b), respectivamente.

As instâncias de VNF necessárias para cada serviço devem ser posicionadas em pontos de presença da rede (N-PoPs). Os N-PoPs são infraestruturas (por exemplo, servidores, clusters ou mesmo datacenters) espalhados na infraestrutura nos quais as funções de rede podem ser aprovisionadas. Sem perda de generalidade, assume-se a existência de um N-PoP associado a cada dispositivo de encaminhamento em uma infraestrutura de backbone (representado pelos círculos na Figura A.1(c)). Cada N-PoP tem uma certa quantidade de recursos (por exemplo,

Figure A.1: Exemplo de SFCs e uma visão parcial da infraestrutura de rede.



(a) SFC com fluxo bifurcado

(b) SFC com fluxo único

(c) Infraestrutura de rede e o aprovisionamento das SFCs

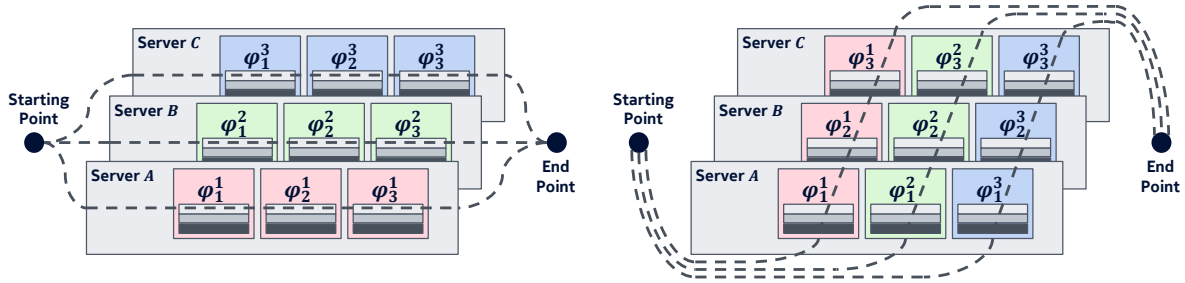
Fonte: autor (2016).

poder de computação disponível). Da mesma maneira, cada SFC tem seus próprios requisitos. Por exemplo, as funções que compõem uma SFC devem sustentar uma determinada carga de trabalho e, portanto, estão associadas a requisitos de computação. Além disso, espera-se que o tráfego entre funções alcance uma vazão máxima, que deve ser tratado pelo caminho físico que conecta os N-PoPs que hospedam essas funções.

Dado o contexto acima, o primeiro problema que abordado nesta tese consiste em encontrar um posicionamento adequada para VNFs em N-PoPs distribuídos e realizar o encadeamento das mesmas, de modo que a utilização de recursos da infraestrutura seja minimizado. O posicionamento e o encadeamento devem garantir cada um dos requisitos das SFCs, bem como as restrições da infraestrutura. Uma possível implantação é ilustrada na Figura A.1. Os *endpoints*, representados como círculos preenchidos, denotam fluxos originados/destinados de/para dispositivos/redes conectados aos dispositivos de encaminhamento da infraestrutura. Observe que ambas as SFCs compartilham uma mesma instância da VNF 1, posicionada no N-PoP (2), minimizando a alocação de recursos conforme desejado. Embora relativamente simples para pequenas instâncias, o problema de posicionamento e encadeamento de VNFs é NP-completo (LUIZELLI et al., 2017a), como discutido no Capítulo3.

Posicionamento e Encadeamento de Funções Virtualizadas de Rede no contexto Intra-datacenter. O segundo problema que abordado trata especificamente do posicionamento de

Figure A.2: Exemplo de estratégias para aprovisionar SFCs



(a) Estratégia de posicionamento “Gather” (b) Estratégia de posicionamento “Distribute”

Fonte: autor (2017).

VNFs em servidores de datacenters (N-PoPs). Uma solução para o problema acima (isto é, o inter-datacenter VNFPC) envolve o posicionamento e encadeamento de requisições de SFC em vários locais (distribuídos). Particularmente, este é o caso quando os SFCs possuem requisitos de rede rigorosos (por exemplo, atrasos fim-a-fim muito baixos). Como resultado do planejamento global (isto é, do inter-datacenter), SFCs podem ser potencialmente divididas em sub-cadeias individualmente posicionadas e encadeadas em datacenters específicos. Então, essas requisições de SFC parciais são implantadas em servidores disponíveis.

No contexto do problema VNFPC intra-datacenter, a identificação de mecanismos de implantação que minimizem o custo de provisionamento dos encadeamentos de serviços tem recebido significativa atenção da academia e indústria (MEHRAGHDAM; KELLER; KARL, 2014; CLAYMAN et al., 2014; GHAZNAVI et al., 2015; LEWIN-EYTAN et al., 2015; LUIZELLI et al., 2015; BARI et al., 2015; RANKOTHGE et al., 2015; BOUET; LEGUAY; CONAN, 2015; LUKOVSZKI; SCHMID, 2015; KUO et al., 2016; ROST; SCHMID, 2016; LUKOVSZKI; ROST; SCHMID, 2016; LUIZELLI et al., 2017a). No entanto, estudos existentes negligenciam o custo operacional real das implantações de SFCs em ambientes NFV típicos. Portanto, os modelos propostos (por exemplo, utilizados em orquestradores NFV) podem levar à soluções infactíveis (em termos de requisitos de CPU) ou sofrer penalizações no desempenho esperado.

Em uma tentativa de abordar essa lacuna, avalia-se e modela-se os custos operacionais de elementos encaminhamento virtualizados (virtual switching) em uma infraestrutura NFV (LUIZELLI et al., 2017b). Neste ambiente, virtual switching é um bloco de construção essencial que permite a comunicação flexível entre VNFs. No entanto, sua operação vem com um custo em termos de recursos computacionais que precisam ser alocados para a camada de encaminhamento, a fim de orientar o tráfego através de serviços em execução (além de recursos computacionais requeridos pelas VNFs). Esse custo depende principalmente da maneira como as VNFs são internamente encadeadas, requisitos de processamento de pacotes e tecnologias de aceleração (por exemplo, Intel DPDK (INTEL, 2016a)).

A Figura A.2 ilustra possíveis implementações de cadeias de serviço em três servidores físicos idênticos (A, B e C). Como é possível observar, todas as cadeias de serviço φ são com-

postas pelas mesmas VNFs - $\varphi^{1,2,3} = \langle \varphi_1 \rightarrow \varphi_2 \rightarrow \varphi_3 \rangle$. Por simplicidade, assume-se que toda o roteamento do tráfego é realizado pela camada de switching (virtual) existente em cada servidor. Ademais, todas as VNFs requerem a mesma quantidade de CPU para executar suas tarefas, além de estarem associadas a uma mesma quantidade conhecida de tráfego. Figuras A.2(a) e A.2(b) ilustram duas estratégias de implantação de VNFs amplamente aplicadas no contexto do OpenStack Cloud Orchestrator (ONF, 2015). Na Figura A.2(a), todas as VNFs de uma única requisição SFC são implantadas no mesmo servidor, estratégia referida como "gather". Em contraste, a Figura A.2(b) ilustra uma estratégia de implantação em que cada VNF de um mesma SFC é implantado em diferentes servidores (referenciada como "distribute"). Observe que todos os servidores têm o mesmo número de VNF e, portanto, os servidores apresentam os mesmos requisitos de processamento da CPU. No entanto, determinar a quantidade de recursos de processamento necessários para operar a camada de encaminhamento virtual (virtual switching) dentro de cada servidor está longe de ser direta mesmo quando se considera estratégias de implantação simples. Portanto, a compreensão desses custos operacionais em implantações reais de NFV é de suma importância por três razões principais: (*i*) para garantir que os requisitos de desempenho dos serviços de rede implantados (por exemplo, vazão máxima de uma SFC); (*ii*) para projetar estratégias de posicionamento eficientes e estimar com precisão o custo operacional para estratégias arbitrárias; e (*iii*) para reduzir o custo operacional (em particular, o consumo de CPU da camada de encaminhamento) dos provedores de NFV.

A.2 Objetivos e Contribuições

O estudo desenvolvido nesta tese tem cinco objetivos principais: (*i*) formalizar o problema de posicionamento e encadeamento de funções virtuais de rede (inter e intra-datacenter); (*ii*) projetar métodos algorítmicos eficientes e escaláveis para prover soluções com alta qualidade para o problema VNFPC; (*iii*) aferir e modelar os custos operacionais da implantação de SFCs, bem como as limitações de desempenho da infraestrutura de rede, em um ambiente NFV típico; (*iv*) minimizar os custos operacionais incorridos nas infraestruturas NFV; e (*v*) otimizar como as SFCs são monitoradas nas infraestruturas NFV.

Os objetivos acima descritos se desdobram nas principais contribuições científicas desta tese, descritas abaixo.

A primeira contribuição abrange a formalização do problema de posicionar e encadear função de rede virtuais (VNFPC) por meio de um modelo de programação Linear Inteira Inteiro (LUIZELLI et al., 2015). O modelo desenvolvido considera uma ampla gama de requisitos típicos em ambientes NFV (por exemplo, poder de computação das funções de rede, requisito de capacidade de fluxo, etc.). Além disso, para lidar com as infraestruturas NFV de tamanho médio, propõe-se um procedimento heurístico que orienta eficientemente os *solvers* comerciais ao longo da exploração de soluções. Compara-se as abordagens ótimas e heurísticas considerando diferentes casos de uso e métricas, como o número de funções de rede virtual instanciadas, o

consumo de recursos físicos e virtuais e latências fim-a-fim.

Como a segunda grande contribuição, abordamos a escalabilidade do problema VNFPC, propondo um algoritmo heurístico baseado em fix-and-optimize (LUIZELLI et al., 2017a). O método combina programação matemática (Integer Linear Programming) e o método heurístico (Variable Neighborhood Search (HANSEN; MLADENOVIć, 2001), de modo a produzir soluções de alta qualidade para instâncias do problema em larga escala em um tempo hábil. Fornece-se fortes evidências, apoiadas por um extenso conjunto de experiências, que o algoritmo heurístico escala para ambientes compostos por centenas de funções de rede. Resultados mostram que o método desenvolvido é capaz de gerar soluções apenas 20 % longe do *lower-bound* calculado, e supera as soluções algorítmicas existentes, em qualidade, por um fator de 5. Além disso, prova-se a natureza NP-completa do problema em questão – este é o primeiro estudo a validar formalmente o carácter NP-completo do problema.

A terceira contribuição do nosso trabalho compreende medir e modelar custos operacionais e métricas de rede de diferentes estratégias de implantação de SFC em infraestruturas de NFV reais (LUIZELLI et al., 2017b). Realiza-se uma avaliação extensa e aprofundada, medindo o desempenho e analisando o impacto da implantação de SFC em um ambiente contendo *Open vSwitch* - o switching virtual padrão para ambientes em nuvem (PFAFF et al., 2009; PFAFF et al., 2015; ONF, 2016). Com base nessa avaliação, definiu-se uma função de custo generalizada que captura com precisão o custo da CPU associados com a camada de encaminhamento virtual. Para isso, mediu-se os custos de encaminhamento para duas estratégias de implantação de VNFs amplamente aplicadas – a saber, “distributed” e “gather”.

Como a quarta contribuição, desenvolvemos um mecanismo algorítmico de implantação de SFCs que considera tanto o desempenho real dos serviços quanto os recursos necessários para a operação (isto é, para a camada de encaminhamento). O método decompõe cada SFC em sub-cadeias e implanta cada sub-cadeia em um servidor físico (possivelmente diferente), de forma a minimizar o custo total de encaminhamento. Para isso, projetou-se um novo algoritmo o qual é baseado na redução bem conhecida ao problema de fluxo com custo mínimoi em redes. O desempenho deste algoritmo em comparação com os mecanismos existentes em orquestradores NFV (por exemplo, módulo `nova-scheduler` no OpenStack) mostra que o algoritmo desenvolvido supera significativamente o OpenStack (por um fator de até 4) em relação aos custos operacionais.

Como quinta contribuição, abordamos o problema do monitoramento eficiente de – outro importante bloco construtor para o funcionamento ideal dos serviços de rede em ambientes NFV. Primeiro, formaliza-se o problema DNM (Distributed Network Monitoring), e então propõe-se um modelo ILP para solucioná-lo. O modelo de otimização é capaz de coordenar efetivamente o monitoramento de cadeias de serviços em ambientes NFV. O modelo está ciente de componentes topológicos de SFCs, o que permite monitorar de forma independente elementos dentro de um serviço de rede com baixo *overhead* em termos de consumo de tráfego de rede e coletores implantados.

As cinco principais contribuições desta tese são, portanto, resumidas a seguir:

1. **Posicionamento e Encadeamento de VNF no Contexto Inter-datacenter.**
    - Formaliza-se o problema de posicionamento e encadeamento da funções virtualizadas de rede (VNFPC) por meio de um modelo de programação linear inteiro.
    - Prova-se que o problema VNFPC é NP-completo.
    - Propõe-se um procedimento heurístico que orienta dinamicamente e com eficiência a busca de soluções realizadas por *solvers* comerciais. Além disso, demonstra-se que a heurística desenvolvida escala para infraestruturas de tamanho médio – com soluções próximas da otimalidade.
2. **Escalabilidade Limitada das Soluções Existentes.**
    - Aborda-se a escalabilidade do VNFPC propondo um novo algoritmo heurístico baseado em fix-and-optimize. Demonstra-se que o método proposto escala para grandes infraestruturas NFV (isto é, milhares de nós NFV).
3. **Custo Operacionais não realistas.**
    - Afere-se os custos operacionais incorridos de diferentes estratégias de implantação de SFC em infraestruturas NFV típicas.
    - Desenvolve-se um modelo analítico para estimar adequadamente o custo da camada de *switching* (custo operacional) para diferentes estratégias de posicionamento em infraestruturas NFV.
4. **Posicionamento e Encadeamento de VNF no Contexto Intra-datacenter.**
    - Generaliza-se modelo analítico anterior para estimar corretamente o custo operacional de estratégia arbitrária de posicionamento de VNFs.
    - Projeta-se um algoritmo *on-line* com o objetivo de minimizar os custos operacionais (isto é, *software switching*) para requisições de SFCs.
5. **Monitoramento Eficiente de SFCs.**
    - Formaliza-se o problema de Monitoramento de Rede Distribuída (*Distributed Network Monitoring* – DNM) e propõe-se um modelo de Programação Linear Inteira.
    - Avalia-se os ganhos alcançados nas soluções DNM em relação à abordagens tradicionais de monitoramento.

APPENDIX B PAPER PUBLISHED AT IFIP/IEEE IM 2015

- **Title**: *Piecing together the NFV provisioning puzzle: Efficient placement and chaining of virtual network functions*
- **Conference**: IFIP/IEEE Integrated Network Management Symposium (IM 2015)
- **Type**: Main track (full-paper)
- **Qualis**: A2
- **Date**: May 11-15, 2015
- **Held at**: Ottawa, CA

Abstract. Network Function Virtualization (NFV) is a promising network architecture concept, in which virtualization technologies are employed to manage networking functions via software as opposed to having to rely on hardware to handle these functions. By shifting dedicated, hardware-based network function processing to software running on commoditized hardware, NFV has the potential to make the provisioning of network functions more flexible and cost-effective, to mention just a few anticipated benefits. Despite consistent initial efforts to make NFV a reality, little has been done towards efficiently placing virtual network functions and deploying service function chains (SFC). With respect to this particular research problem, it is important to make sure resource allocation is carefully performed and orchestrated, preventing over- or under-provisioning of resources and keeping end-to-end delays comparable to those observed in traditional middlebox-based networks. In this paper, we formalize the network function placement and chaining problem and propose an Integer Linear Programming (ILP) model to solve it. Additionally, in order to cope with large infrastructures, we propose a heuristic procedure for efficiently guiding the ILP solver towards feasible, near-optimal solutions. Results show that the proposed model leads to a reduction of up to 25% in end-to-end delays (in comparison to chainings observed in traditional infrastructures) and an acceptable resource over-provisioning limited to 4%. Further, we demonstrate that our heuristic approach is able to find solutions that are very close to optimality while delivering results in a timely manner.

Piecing Together the NFV Provisioning Puzzle: Efficient Placement and Chaining of Virtual Network Functions

Marcelo Caggiani Luizelli, Leonardo Richter Bays, Luciana Salete Buriol
Marinho Pilla Barcellos, Luciano Paschoal Gaspary
Institute of Informatics – Federal University of Rio Grande do Sul (UFRGS)
{mcluizelli,lrbays,buriol,marinho,paschoal}@inf.ufrgs.br

***Abstract*—Network Function Virtualization (NFV) is a promising network architecture concept, in which virtualization technologies are employed to manage networking functions via software as opposed to having to rely on hardware to handle these functions. By shifting dedicated, hardware-based network function processing to software running on commoditized hardware, NFV has the potential to make the provisioning of network functions more flexible and cost-effective, to mention just a few anticipated benefits. Despite consistent initial efforts to make NFV a reality, little has been done towards efficiently placing virtual network functions and deploying service function chains (SFC). With respect to this particular research problem, it is important to make sure resource allocation is carefully performed and orchestrated, preventing over- or under-provisioning of resources and keeping end-to-end delays comparable to those observed in traditional middlebox-based networks. In this paper, we formalize the network function placement and chaining problem and propose an Integer Linear Programming (ILP) model to solve it. Additionally, in order to cope with large infrastructures, we propose a heuristic procedure for efficiently guiding the ILP solver towards feasible, near-optimal solutions. Results show that the proposed model leads to a reduction of up to 25% in end-to-end delays (in comparison to chainings observed in traditional infrastructures) and an acceptable resource over-provisioning limited to 4%. Further, we demonstrate that our heuristic approach is able to find solutions that are very close to optimality while delivering results in a timely manner.**

I. Introduction

Middleboxes (or Network Functions - NF) play an essential role in today's networks, as they support a diverse set of functions ranging from security (*e.g.*, firewalling and intrusion detection) to performance (*e.g.*, caching and proxying) [1]. As currently implemented nowadays, middleboxes are difficult to deploy and maintain. This is mainly because cumbersome procedures need to be followed, such as dealing with a variety of custom-made hardware interfaces and manually chaining middleboxes to ensure the desired network behavior. Further, recent studies show that the number of middleboxes in enterprise networks (as well as in datacenter and ISP networks) is similar to the number of physical routers [2]–[4]. Thus, the aforementioned difficulties are exacerbated by the complexity imposed by the high number of network functions that a network provider has to cope with, leading to high operational expenditures. Moreover, in addition to costs related to manually deploying and chaining middleboxes, the need for frequent hardware upgrades adds up to substantial capital investments.

Network Function Virtualization (NFV) has been proposed to shift middlebox processing from specialized hardware appliances to software running on commoditized hardware [5]. In addition to potentially reducing acquisition and maintenance costs, NFV is expected to allow network providers to make most of the benefits of virtualization on the management of network functions (*e.g.*, elasticity, performance, flexibility, etc.). In this context, Software-Defined Networking (SDN) can be considered a convenient complementary technology, which, if available, has the potential to make the chaining of the aforementioned network functions much easier. In fact, it is not unreasonable to state that SDN has the potential to revamp the Service Function Chaining (SFC)[1] problem. In short, the problem consists of making sure network flows go efficiently through end-to-end paths traversing sets of middleboxes. In the NFV/SDN realm and considering the flexibility offered by this environment, the problem consists of (sub)optimally defining how many instances of virtual network functions are necessary and where to place them in the infrastructure. Furthermore, the problem encompasses the determination of end-to-end paths over which known network flows have to be transmitted so as to pass through the required placed network functions.

Despite consistent efforts to make NFV a reality [1], [6], little has been done to efficiently perform the placement and chaining of virtual network functions on physical infrastructures. This is particularly challenging mainly for two reasons. First, depending on how virtual network functions are positioned and chained, end-to-end latencies may become intolerable. This problem is aggravated by the fact that processing times tend to be higher, due to the use of virtualization, and may vary, depending on the type of network function and the hardware configuration of the device hosting it. Second, resource allocation must be performed in a cost-effective manner, preventing over- or under-provisioning of resources. Therefore, placing network functions and programming network flows in a cost-effective manner while ensuring acceptable end-to-end delays represents an essential step toward enabling the use of NFV in production environments.

In this paper, we formalize the network function placement and chaining problem and propose an optimization model to solve it. Additionally, in order to cope with large infrastructures, we propose a heuristic procedure. Both optimal and heuristic approaches are evaluated considering different use cases and metrics, such as the number of instantiated virtual network functions, physical and virtual resource consumption, and end-to-end latencies. The main contributions of this paper are then: *(i)* the formalization of the network function placement and chaining problem by means of an ILP model; *(ii)* the proposal of a heuristic solution, and *(iii)* the evaluation of both proposed approaches and discussion of the obtained results.

The remainder of this paper is organized as follows. In Section 2, we discuss related work in the area of network function virtualization. In Section 3, we formalize the network function placement and chaining problem and propose both an optimal ILP model and a heuristic approach to solve it. In

[1]In this paper, the terms network function and service function are used interchangeably.

978-3-901882-76-0 @2015 IFIP

Section 4, we present and discuss the results of an evaluation of the model and heuristic. Last, in Section 5 we conclude the paper with final remarks and perspectives for future work.

II. Related Work

We now review some of the most prominent research work related to network function virtualization and the network function placement and chaining problem. We start the section by discussing recent efforts aimed at evaluating the technical feasibility of deploying network functions on top of commodity hardware. Then, we review preliminary studies carried out to solve different aspects of the virtual network function placement and chaining problem.

Hwang *et al.* [6] propose the NetVM platform to allow network functions based on Intel DPDK technology to be executed at line-speed (*i.e.*, 10 Gb/s) on top of commodity hardware. According to the authors, it is possible to accelerate network processing by mapping NIC buffers to user space memory. In another investigation, Martins *et al.* [1] introduce a high-performance middlebox platform named ClickOS. It consists of a Xen-based middlebox software, which, by means of alterations in I/O subsystems (back-end switch, virtual net devices and back and front-end drivers), can sustain a throughput of up to 10 Gb/s. The authors show that ClickOS enables the execution of hundreds of virtual network functions concurrently without incurring significant overhead (in terms of delay) in packet processing. The results obtained by Hwang *et al*. and Martins *et al*. are promising and definitely represent an important milestone to make the idea of virtual network functions a reality.

With respect to efficient placement and chaining of network functions, the main contribution of this paper, Barkai *et al.* [7] and Basta *et al.* [8] have recently taken a first step toward modeling this problem. Barkai *et al.*, for example, propose mechanisms to program network flows to an SDN substrate taking into account virtual network functions through which packets from these flows need to pass. In short, the problem consists of mapping SDN traffic flows properly (*i.e.*, in the right sequence) to virtual network functions. To solve it in a scalable manner, the authors propose a more efficient topology awareness component, which can be used to rapidly program network flows. Note that they do not aim at providing a (sub)optimal solution to the network function placement and chaining problem as we do in this paper. Instead, the scope of their work is more of an operational nature, *i.e.*, building an OpenFlow-based substrate that is efficient enough to allow flows – potentially hundred of millions, with specific function processing requirements – to be correct and timely mapped and programmed. Our solution could be used together with Barkai's and therefore help the decision on where to optimally place network functions and how to correctly map network flows.

The work by Basta *et al.*, in turn, proposes an ILP model for network function placement in the context of cellular networks and crowd events. More specifically, the problem addressed is the question on whether or not virtualize and migrate mobile gateway functions to datacenters. When applicable, the model also encompasses the optimal selection of datacenters that will host the virtualized functions and SDN controllers. Although the paper covers optimal virtual function placement, the proposed model is restricted, as it does not have to deal with function chaining. Our proposal is, in comparison, a broader, optimal solution. It can be applied to plan not only the placement of multiple instances of virtual network functions on demand, but also to map and chain service functions.

Before summarizing this section, we add a note on the relation between the network function placement and chaining problem and the Virtual Network Embedding (VNE) problem [9]–[12]. Despite some similarities, solutions to the latter are not appropriate to the former. The reason is twofold. First, while in VNE we observe one-level mappings (virtual network requests $\rightarrow$ physical network), in NFV environments we have two-level mappings (service function chaining requests $\rightarrow$ virtual network function instances $\rightarrow$ physical network). Second, while the VNE problem considers only one type of physical device (*i.e.*, routers), a much wider number of different network functions coexist in NFV environments.

As one can observe from the state-of-the-art, the area of network function virtualization is still in its early stages. Most of the effort has been focused on engineering ways of running network functions on top of commodity hardware while keeping performance roughly the same as the one obtained when deploying traditional middlebox-based setups. As far as we are aware of, this paper consolidates a first consistent step towards placing virtual network functions and mapping service function chains. Besides, it captures and discusses the trade-off between resources employed and performance gains in the particular context of NFV.

III. The Network Function Placement and Chaining Problem

In this section, we describe the network function placement and chaining problem and introduce our proposed solution. Next, we formalize it as an Integer Linear Programming model, followed by an algorithmic approach.

A. Problem Overview

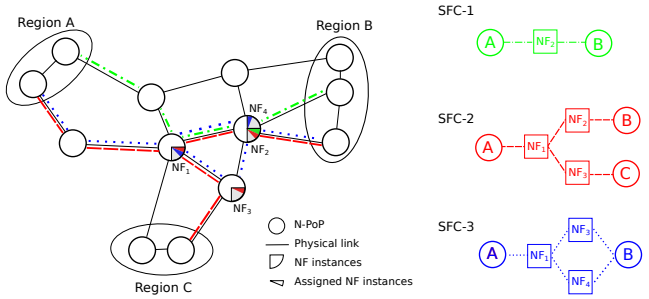
As briefly explained earlier, network function placement and chaining consists of interconnecting a set of network functions (*e.g.*, firewall, load balancer, etc.) through the network to ensure network flows are given the correct treatment. These flows must go through end-to-end paths traversing a specific set of functions. In essence, this problem can be decomposed into three phases: *(i)* placement, *(ii)* assignment, and *(iii)* chaining.

The placement phase consists of determining how many network function instances are necessary to meet the current/expected demand and where to place them in the infrastructure. Virtual network functions are expected to be placed on network points of presence (N-PoPs), which represent groups of (commodity) servers in specific locations of the infrastructure (with processing capacity). N-PoPs, in turn, would be potentially set up either in locations with previously installed commuting and/or routing devices or in facilities such as datacenters.

The assignment phase defines which placed virtual network function instances (in the N-PoPs) will be in charge of each flow. Based on the source and destination of a flow, instances are assigned to it in a way that prevents processing times from causing intolerable latencies. For example, it may be more efficient to assign network function requests to the nearest virtual network function instance or to simply split the requested demand between two or more virtual network functions (when possible).

In the third and final phase, the requested functions are chained. This process consists of creating paths that interconnect the network functions placed and assigned in the previous phases. This phase takes into account two crucial factors, namely end-to-end path latencies and distinct processing delays added by different virtual network functions. Figure 1 depicts the main elements involved in virtual network function placement and chaining. The physical network is composed of N-PoPs interconnected through physical links. There is a set of SFC requests that contain logical sequences of network

functions as well as the endpoints, which implicitly define the paths. Additionally, the provider has a set of virtual network function images that it can instantiate. In the figure, larger semicircles represent instances of network functions running on top of an N-PoP, whereas the circumscribed semicircles represent network function requests assigned to the placed instances. The gray area in the larger semicircles represents processing capacity allocated to network functions that is not currently in use. Dashed lines represent paths chaining the requested endpoints and network functions.



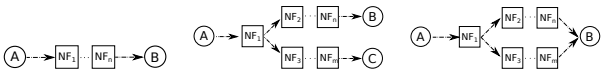
(a) Physical infrastructure. (b) SFC requests.

Fig. 1. Example SFC deployment on a physical infrastructure to fulfill a number of requests.

B. Topological Components of SFC Requests

SFC requests may exhibit different characteristics depending on the application or flow they must handle. More specifically, such requests may differ topologically and/or in size. In this paper, we consider three basic types of SFC components, which may be combined with one another to form more complex requests. These three variations – *(i)* line, *(ii)* bifurcated path with different endpoints, and *(iii)* bifurcated path with a single endpoint – are explained next.

The simplest topological component that may be part of an SFC request is a line with two endpoints and one or more network functions. This kind of component is suitable for handling flows between two endpoints that have to pass through a particular sequence of network functions, such as a firewall and a Wide Area Network (WAN) accelerator. The second and third topological components are based on bifurcated paths. Network flows passing through bifurcated paths may end up at the same endpoint or not. Considering flows with different endpoints, the most basic component contains three endpoints (one source and two destinations). Between them, there is a network function that splits the traffic into different paths according to a certain policy. A classical example that fits this topological component is a load balancer connected to two servers. As for bifurcated paths with a single end point, we consider a scenario in which different portions of traffic between two endpoints must be treated differently. For example, part of the traffic has to pass through a specific firewall, while the other part, through an encryption function. Figure 2 illustrates these topological components. As previously mentioned, more sophisticated SFC requests may be created by freely combining these basic topological components among themselves or in a recursive manner.



(a) Line. (b) Bifurcated path with different endpoints. (c) Bifurcated path with a single endpoint.

Fig. 2. Basic topological components of SFC requests.

C. Definitions and Modeling

Next, we detail the inputs, variables, and constraints of our optimization model. Superscript letters represent whether a set or variable refers to service chaining requests (S) or physical (P) resources, or whether it relates to nodes (N) or links (L).

NFV Infrastructure and Service Function Chaining. The topology of the NFV infrastructure, as well as that of each SFC, is represented as a directed graph $G = (N, L)$. Vertices N represent network points of presence (N-PoPs) in physical infrastructures or network functions in SFCs. Each N-PoP represents a location where a network function may be implemented. In turn, each edge $(i, j) \in L$ represents an unidirectional link. Bidirectional links are represented as a pair of edges in opposite directions (*e.g.*, (a, b) and (b, a)). Thus, the model allows the representation of any type of physical topology, as well as any SFC forwarding graph.

In real environments, physical devices have a limited amount of resources. In our model, the CPU capacity of each N-PoP is represented as C_i^P. In turn, each physical link in the infrastructure has a limited bandwidth capacity and a particular delay, represented by $B_{i,j}^P$ and $D_{i,j}^P$, respectively. Similarly, SFCs require a given amount of resources. Network functions require a specific amount of CPU, represented as C_i^S. Additionally, each SFC being requested has a bandwidth demand and a maximum delay allowed between its endpoints, represented as $B_{i,j}^S$ and D^S, respectively.

Virtual Network Functions. Set F represents possible virtual network functions (*e.g.*, firewall, load balancer, NAT, etc.) that may be instantiated/placed by the infrastructure operator on top of N-PoPs. Each network function can be instantiated at most U_m times, which is determined by the number of licenses the provider has purchased. Each virtual function instance requires a given amount of physical resources (which are used by SFCs mapped to that instance). Each instance provides a limited amount of resources represented by F_m^{cpu}. This enables our model to represent instances of the same type of network function with different sizes (*e.g.*, pre-configured instances for services with higher or lower demand). Each function $m \in F$ has a processing delay associated with it, which is represented by F_m^{delay}. Moreover, we consider that each mapped network function instance may be shared by one or more SFCs whenever possible.

SFC requests. Set Q represents SFCs that must be properly assigned to network functions. SFCs are composed of chains of network functions and each requested function is represented by q. Each link interconnecting the chained functions requires a given amount of bandwidth, represented by $B_{i,j}^V$. Furthermore, each request has at least two endpoints, representing specific locations on the infrastructure. The required locations of SFC endpoints are stored in set S^C. Likewise, the physical location of each N-PoP i is represented in S^P. Since the graph of an SFC request may represent any topology, we assume that the set of virtual paths available to carry data between pairs of endpoints is known in advance. As there are efficient algorithms to compute paths, we opted to compute them in advance without loss of generality to the model. This set is represented by P.

Variables. The variables are the outputs of our model, and represent the optimal solution of the service function chaining problem for the given set of inputs. These variables indicate in which N-PoP virtual network functions are instantiated (placed). Further, these variables indicate the assignment of SFCs being requested to virtual network functions placed in the infrastructure. If a request is accepted, each of its virtual functions is mapped to an N-POP, whereas each link in the

chain is mapped to one or more consecutive physical links (*i.e.*, a physical path).

- $y_{i,m,j} \in \{0,1\}$ – Virtual network function placement, indicates whether instance j of network function m is mapped to N-PoP i.
- $A^N_{i,q,j} \in \{0,1\}$ – Assignment of required network functions, indicates whether virtual network function j, required by SFC q, is serviced by a network function placed on N-PoP i.
- $A^L_{i,j,q,k,l} \in \{0,1\}$ – Chaining allocation, indicates whether physical link (i,j) is hosting virtual link (k,l) from SFC q.

Based on the above inputs and outputs, we now present the objective function and its constraints. The objective function of our model aims at minimizing the number of virtual network function instances mapped on the infrastructure. This objective was chosen due to the fact that this aspect has the most significant and direct impact on the network provider's costs. However, our model could be easily adapted to use other objective functions, such as multi-objective ones that consider different factors simultaneously (*e.g.*, number of network function instances and end-to-end delays). The purpose of each constraint of our model is explained next.

Objective:

$$\textbf{Min} \sum_{i \in R^P, m \in F, j \in U_m} y_{i,m,j} \quad (1)$$

Subject to:

$$\sum_{m \in F, j \in U_m} y_{i,m,j} \cdot F^{cpu}_{m,j} \leq C^P_i \qquad \forall i \in R^P \quad (2)$$

$$\sum_{q \in Q, j \in R^V_q : q = F_m} C^S_{q,j} \cdot A^N_{i,q,j} \leq \sum_{j \in U_m} y_{i,m,j} \cdot F^{cpu}_{m,j} \qquad \forall i \in R^P, m \in F \quad (3)$$

$$A^N_{i,q,k} \leq \sum_{m \in F, j \in U_m : m = F_k} y_{i,m,j} \qquad \forall i \in R^P, q \in Q, k \in R^S_q \quad (4)$$

$$\sum_{q \in Q, (k,l) \in L^V} B^S_{q,k,l} \cdot A^L_{i,j,q,k,l} \leq B^P_{i,j} \qquad \forall (i,j) \in L^P \quad (5)$$

$$\sum_{i \in R^P} A^N_{i,q,j} = 1 \qquad \forall q \in Q, k \in R^S_q \quad (6)$$

$$\sum_{j \in R^P} A^L_{i,j,q,k,l} - \sum_{j \in R^P} A^L_{j,i,q,k,l} = A^N_{i,q,k} - A^N_{i,q,l} \qquad \forall q \in Q, i \in R^P, (k,l) \in L^S_q \quad (7)$$

$$A^N_{i,q,j} \cdot j = A^N_{i,q,k} \cdot l \qquad \forall (i,j) \in S^P, q \in Q, (k,l) \in S^S_q \quad (8)$$

$$\sum_{(i,j) \in L^P, (k,l) \in L^S_q : (k,l) \in p} A^L_{i,j,q,k,l} \cdot D^P_{i,j} + \sum_{i \in R^P, k \in R^S_q : k \in p} A^N_{i,q,j} \cdot F^{delay}_k \leq D^S_k \qquad \forall q \in Q, p \in P_q \quad (9)$$

Constraint 2 ensures that the sum of CPU capacities required by network function instances mapped to N-PoP i does not exceed the amount of available physical resources. In turn, constraint 3 ensure that the sum of processing capacities required by elements of SFCs does not exceed the amount of virtual resources available on network function m mapped to N-PoP i. Constraint 4 ensures that, if a network function being requested by an SFC is assigned to N-PoP i, then at least one network function instance should be running (placed) on i. Constraint 5 ensures that the virtual path between the required endpoints has enough available bandwidth to carry the amount of flow required by SFCs. Constraint 6 ensures that every required SFC (and its respective network functions) is mapped to the infrastructure. Constraint 7 consists in building the virtual paths between the required endpoints. Constraint 8 ensures that the required endpoints are mapped to devices in the requested physical locations. Last, constraint 9 ensures that end-to-end latency constraints on mapped SFC requests will be met (the first part of the equation is a sum of the delay incurred by end-to-end latencies between mapped endpoints, while the second part defines the delay incurred by packet processing on virtual network functions).

D. Proposed Heuristic

In this subsection we present our heuristic approach for efficiently placing, assigning, and chaining virtual network functions. We detail each specific procedure it uses to build a feasible solution, and present an overview of its algorithmic process.

In this particular problem, the search procedure performed by the integer programming solver leads to an extensive number of symmetrical feasible solutions. This is mainly because there is a considerable number of potential network function mappings/assignments that satisfy all constraints, in addition to the fact that search schemes conducted by commercial solvers are not specialized for the problem in hand.

To address the aforementioned issues, our heuristic approach dynamically and efficiently guides the search for solutions performed by solvers in order to quickly arrive at high quality, feasible ones. This is done by performing a binary search to find the lowest possible number of network function instances that meets the current demands. In each iteration, the heuristic employs a modified version of the proposed ILP model in which the objective function is removed and transformed into a constraint, resulting in a more bounded version of the original model. This strategy takes advantage of two facts: first, there tends to be a significant number of feasible, symmetrical solutions that meet our criteria for optimality, and once the lowest possible number of network function instances is determined, only one such solution needs to be found; and second, commercial solvers are extremely efficient in finding feasible solutions.

Algorithm 1 presents a simplified pseudocode version of our heuristic approach, and its details are explained next. The heuristic performs a binary search that attempts to find a more constrained model by dynamically adjusting the number of network functions that must be instantiated on the infrastructure. The upper bound of this search is initially set to the maximum number of network functions that may be instantiated on the infrastructure (line 2), while the lower bound is initialized as 1 (line 3). In each iteration, the maximum number of network function instances allowed is represented by variable nf, which is increased or decreased based on the aforementioned upper and lower bounds (line 6). After nf is updated, the algorithm transforms the original model into the bounded one by removing the objective function (line 7) and adding a new constraint (line 8), considering the computed value for nf. The added constraint is shown in Equation 10.

$$\sum_{i \in R^P, m \in F, j \in U_m} y_{i,m,j} \leq nf \quad (10)$$

In line 9, a commercial solver is used to obtain a solution for the bounded model within an acceptable time limit. In each iteration, the algorithm stores the best solution found so far (*i.e.*, the solution s with the lowest value for nf – line 11). Afterwards, it adjusts the upper or lower bound depending on whether the current solution is feasible or not (lines 12 and 14). Last, it returns the best solution found (s', which represents variables y, A^N and A^L).

Although the proposed heuristic uses an exact approach to find a feasible solution for the problem, $timeLimit$ (in line 9) should be fine-tuned considering the size of the instance being handled to ensure the tightest solution will be found. In our experience, for example, a time limit in the order of minutes is sufficient for dealing with infrastructures with 200 N-PoPs.

Input: $Infrastructure\ G$, $set\ Q\ of\ SFCs$, $set\ VNF\ of\ network\ functions$, $timeLimit$
Output: $Variables\ y_{i,m,j}$, $A^N_{i,q,j}$, $A^L_{i,j,q,k,l}$

```
1  s, s' ← ∅
2  upperBound ← |F|
3  lowerBound ← 1
4  nf ← (upperBound + lowerBound)/2
5  while nf ≥ lowerBound and nf ≤ upperBound do
6  |  nf ← (upperBound + lowerBound)/2
7  |  Remove objective function
8  |  Add constraint : Σ_{i∈R^P, m∈F, j∈U_m} y_{i,m,j} ≤ nf
9  |  s ← solveAlteredModel(timeLimit)
10 |  if s is feasible then
11 |  |  s' ← s
12 |  |  upperBound ← nf
13 |  else
14 |  |  lowerBound ← nf
15 |  end
16 end
17 if s' = ∅ then
18 |  return infeasible solution
19 else
20 |  return s'
21 end
```

Algorithm 1: Overview of the proposed heuristic.

IV. Evaluation

In order to evaluate the provisioning of different types of SFCs, the ILP model formalized in the previous section was implemented and run in *CPLEX Optimization Studio*[2] version 12.4. The heuristic, in turn, was implemented and run in Python. All experiments were performed on a machine with four Intel Xeon E5-2670 processors and 56 GB of RAM, using the Ubuntu GNU/Linux Server 11.10 x86_64 operating system.

A. Workloads

We consider four different types of SFC components. Each type uses either one of the topological components described in Subsection III-B or a combination of them. The first component is a line composed of a single firewall between the two endpoints (Figure 2(a)). The second component used consists of a bifurcated path with different endpoints (Figure 2(b)). This component is composed of a load balancer splitting the traffic between two servers. These two types of components are comparable since their end-to-end paths pass through exactly one network function. The third and fourth components use the same topologies of the previously described ones, but vary in size. The third component is a line (like Component 1) composed of two chained network functions – a firewall followed by an encryption network function (*e.g.*, VPN). The fourth component is a bifurcated path (like Component 2), but after the load balancer, traffic is forwarded to one more network function – a firewall. These particular network functions were chosen due to being commonly referenced in recent literature; however, they could be easily replaced with any other functions if so desired. All network functions requested by SFCs have the same requirements in terms of CPU and bandwidth. Each network function requires 12.5% of CPU, while the chainings between network functions require 1Gbps of bandwidth. When traffic passes through a load balancer, the required bandwidth is split between the paths. The values for CPU and bandwidth requirements were fixed after a preliminary evaluation, which revealed that they did not have a significant impact on the obtained results. Moreover, the establishment of static values for these parameters facilitates the assessment of the impact of other, more important factors.

The processing times of virtual network functions (*i.e.*, the time required by these functions to process each incoming packet) considered in our evaluation are shown in Table I. These values are based on the study conducted by Dobrescu *et al.* [13], in which the authors determine the average processing time of a number of software-implemented network functions.

TABLE I. Processing times of physical and virtual network functions used in our evaluation.

Network Function	Processing Time (physical)	Processing Time (virtual)
Load Balancer	0.2158 sec	0.6475 sec
Firewall	2.3590 sec	7.0771 sec
VPN Function	0.5462 sec	1.6385 sec

Networks used as physical substrates were generated with Brite[3]. The topology of these networks follows the Barabasi-Albert (BA-2) [14] model. This type of topology was chosen as an approximation of those observed in real ISP environments. Physical networks have a total of 50 N-PoPs, each with total CPU capacity of 100%, while the bandwidth of physical links is 10 Gbps. The average delay of physical links is 30ms. This value is based on the study conducted by Choi *et al.* [15], which characterizes typical packet delays in ISP networks.

In order to provide a comparison between virtualized network functions and non-virtualized ones, we consider baseline scenarios for each type of SFC. These scenarios aim at reproducing the behavior of environments that employ physical middleboxes rather than NFV. Our baseline consists of a modified version of our model, in which the total number of network functions is exactly the number of different functions being requested. Moreover, the objective function attempts to find the minimum chaining length between endpoints and network functions. In baseline scenarios, function capacities are adjusted to meet all demands and, therefore, we do not consider capacity constraints. Further, processing times are three times lower than those in virtualized environments. This is in line with the study of Basta *et al.* [8]. These processing

[2]http://www-01.ibm.com/software/integration/optimization/cplex-optimization-studio/

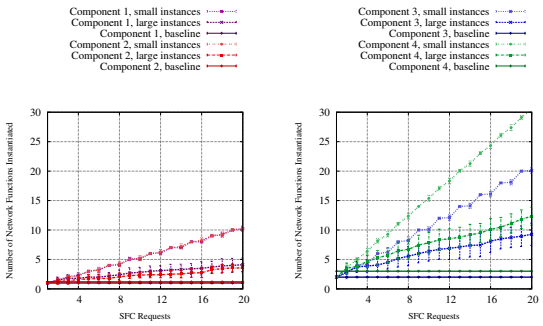
[3]http://www.cs.bu.edu/brite/

times, like the ones related to virtual network functions, are shown in Table I.

In our experiments, we consider two different profiles of network function instances. In the first one, instances may require either 12.5% or 25% of CPU, leading to smaller instance sizes. In the second profile, instances may require 12.5% or 100%, leading to larger instances overall. We first evaluate our optimal approach considering individual types of requests. Next, we evaluate the effect of a mixed scenario with multiple types of SFCs. Last, we evaluate our proposed heuristic using large instances. Each experiment was repeated 30 times, with each repetition using a different physical network topology. All results have a confidence level of 90% or higher.

B. Results

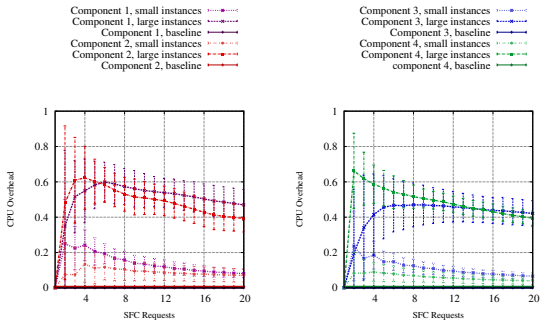
First, we analyze the number of network functions instances needed to cope with an increasing number of SFC requests. Figure 3 depicts the average number of instantiated network functions with the number of SFC requests varying from 1 to 20. At each point on the graph, all previous SFC requests are deployed together. It is clear that the number of instances is proportional to the number of SFC requests. Further, we observe that smaller instance sizes lead to a higher number of network functions being instantiated. Considering small instances, scenarios with Components 1 and 2 require, on average, 10 network function instances (Figure 3(a)). In contrast, scenarios with Components 3 and 4 require, on average, 20 and 30 instances (Figure 3(b)), respectively. For large instances, scenarios with Components 1 and 2 require, respectively, 4 and 3 network function instances, while those with Components 3 and 4 require 9 and 12 instances on average. These results demonstrate that the number of virtual network functions in a SFC request has a much more significant impact on the number of instances needed to service such requests than the chainings between network functions and endpoints. This can be observed, for example, in Figure 3(a), in which Components 1 and 2 only differ topologically and lead to, on average, the same number of instances. In contrast, Figure 3(b) shows that when handling components of type 4 (which have a higher number of network functions than those of type 3), a significantly higher number of network function instances is required.



(a) Components 1 and 2. (b) Components 3 and 4.

Fig. 3. Average number of network function instances.

Figure 4 illustrates the average CPU overhead (*i.e.*, allocated but unused CPU resources) in all experiments. Each point on the graph represents the average overhead from the beginning of the experiment until the current point. In all experiments, CPU overheads tend to be lower when small instances are used. When large instances are allocated, more resources stay idle. Considering small instances, Components 1 and 2 (Figure 4(a)) lead to, on average, CPU overhead of 7.80% and 7.28%, respectively. Components 3 and 4 (Figure 4(b)) lead to, on average, 6.58% and 3.18% CPU overhead. In turn, for large instances, Components 1 and 2 lead to average CPU overheads of 45.61% and 38.68%, respectively. Components 3 and 4 lead to, on average, 40.21% and 40.36% CPU overhead. Observed averages demonstrate that the impact of instance sizes is notably high, with smaller instances leading to significantly lower overheads. Further, we can observe that, in general, CPU overheads tend to be lower when higher numbers of SFCs are being deployed. As more requests are being serviced simultaneously, network function instances can be shared among multiple requests, increasing the efficiency of CPU allocations. In these experiments, the baseline has 0% of CPU overhead as network functions are planned in advance to support the exact demand. Since in a NFV environment network function instances are hosted on top of commodity hardware (as opposed to specialized middleboxes), these overheads – especially those observed for small instances – are deemed acceptable, as they do not incur high additional costs.



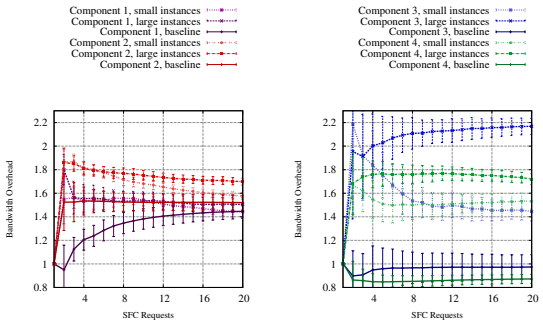
(a) Components 1 and 2. (b) Components 3 and 4.

Fig. 4. Average CPU overhead of network function instances.

Next, Figure 5 shows the average overhead caused by chaining network functions (through virtual links) in each experiment. This overhead is measured as the ratio between the effective bandwidth consumed by SFC virtual links hosted on the physical substrate and the bandwidth requested by such links. In general, the actual bandwidth consumption is higher than the total bandwidth required by SFCs, due to the frequent need to chain network functions through paths composed of multiple physical links. The absence of overhead is observed only when each virtual link is mapped to a single physical link (ratio of 1.0), or when network functions are mapped to the same devices as the requested endpoints (ratio < 1.0). Lower overhead rates may potentially lead to lower costs and allow more SFC requests to be serviced.

Considering large instances, the observed average overhead is 50.49% and 69.87% for scenarios with Components 1 and 2, respectively. In turn, Components 3 and 4 lead to overhead ratios of 116% and 72.01%. This is due to the low number of instantiated network functions (Figure 3), which forces instances to be chained through long paths. Instead, when small instances are considered (*i.e.*, more instances running in a distributed way), overheads tend to be lower. Components 1 and 2 lead to, on average, 44.30% and 57.60% bandwidth overhead, while Components 3 and 4, 44.53% and 53.41%, respectively. When evaluating bandwidth overheads, we can observe that the topological structure of SFC requests has the most significant impact on the results (in contrast to previously discussed experiments). More complex chainings tend to lead to higher bandwidth overheads, although these results are also influenced by other factors such as instance sizes and the number of instantiated functions. In these experiments the baseline overhead tends to be lower than the others as the

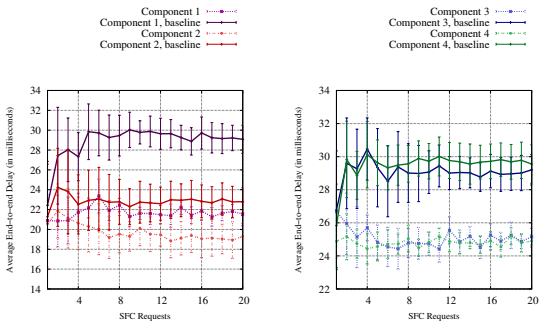
objective function prioritizes shortest paths (in terms of number of hops) between endpoints and network functions.



(a) Components 1 and 2. (b) Components 3 and 4.

Fig. 5. Average bandwidth overhead of SFCs deployed in the infrastructure.

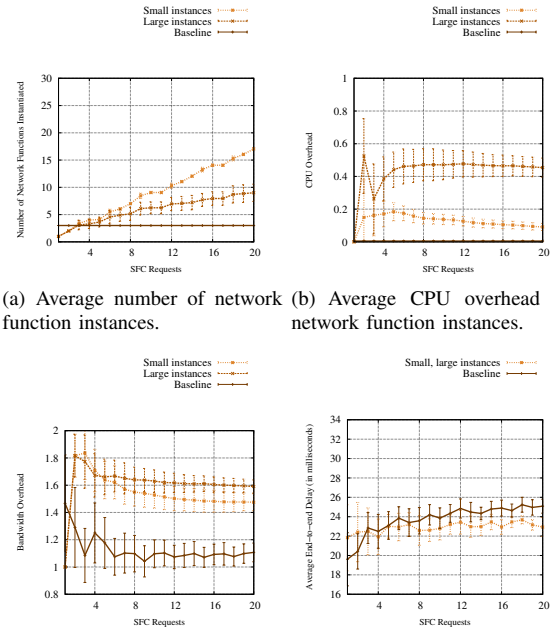
Figure 6 depicts the average end-to-end delay, in milliseconds, observed between endpoints in all experiments. The end-to-end delay is computed as a sum of the path delays and network function processing times. In this figure, results for scenarios with small and large instances are grouped together, as average delays are the same. The observed end-to-end delay for all components tends to be lower than the delay observed for the baseline scenario. This is mainly due to the better positioning of network functions and chainings between them. Furthermore, the model promotes a better utilization of the variety of existing paths in the infrastructure. Although the baseline scenario aims at building minimum chainings (in terms of hops), we observe that: *(i)* minimum chaining does not always lead to global minimum delay; *(ii)* when baseline scenarios overuse the shortest paths, other alternative paths remain unused due to the depletion of resources in specific locations (mainly in the vicinity of highly interconnected nodes). In comparison with baseline scenarios, Component 1 leads to, on average, 25% lower delay (21.55ms compared to 29.07ms), while Component 2 leads to, on average, 15.40% lower delay (19.28ms compared to 22.79ms). In turn, Component 3 leads to, on average, 13.86% lower delay than its baseline (25.15ms compared to 29.20ms), while Component 4 leads to 15.75% lower delay (24.89ms compared to 29.55ms). In summary, even though our baseline scenarios are planned in advance to support exact demands and we consider processing times of virtual network functions to be three times those of physical ones, end-to-end delays are still lower in virtualized scenarios. This advantage may become even more significant as the estimated processing times of virtual network functions get closer in the future to those observed in physical middleboxes.



(a) Components 1 and 2. (b) Components 3 and 4.

Fig. 6. Average end-to-end delay of SFCs deployed in the infrastructure.

After analyzing the behavior of SFCs considering homogeneous components, we now analyze the impact of a mixed scenario. In it, Components 1, 2, and 4 are repeatedly deployed in the infrastructure sequentially. Figure 7 presents the results for the mixed scenario. Although there are different topological SFC components being deployed together in the same infrastructure, the results exhibit similar tendencies as those of homogeneous scenarios. In Figure 7(a), we observe that the average number of network functions (on average, 17 network functions when considering small instances and 9 functions considering large instances) is proportional to the obtained average values depicted in Figures 3(a) and 3(b). The average CPU overhead also remains similar (9.12% considering small instances and 45.37% considering large ones). In turn, the average overhead caused by chaining network functions in the mixed scenario is of 59.06% and 47.51%, for small and large instances, respectively. Despite these similarities, end-to-end delays tend to be comparatively lower than the ones observed in homogeneous scenarios. The delay observed in the proposed chaining approach is 8.57% lower than that of the baseline (22.93ms in comparison to 25.08ms). This is due to the combination of requests with different topological structures, which promotes the use of a wider variety of physical paths (which, in turn, leads to lower overutilization of paths). Similarly to homogeneous scenarios, average end-to-end delays are the same considering small and large instances.



(a) Average number of network function instances. (b) Average CPU overhead of network function instances.

(c) Average bandwidth overhead. (d) Average end-to-end delay.

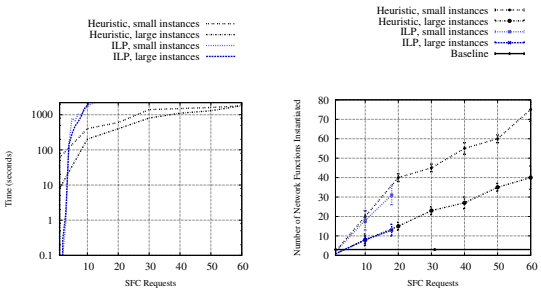
Fig. 7. Mixed scenario including Components 1, 2, and 4.

We now proceed to the evaluation of our proposed heuristic approach. The heuristic was subjected the same scenarios as the ILP model, in addition to ones with a larger infrastructure. Considering the scenarios presented so far (*i.e.*, with physical infrastructures with 50 nodes and 20 SFC requests), our heuristic was able to find an optimal solution in all cases. We omit such results due to space constraints. We emphasize, however, that the heuristic approach was able to find an optimal solution in a substantially shorter time frame in comparison to the ILP model, although the solution times of both approaches remained in the order of minutes. The average solution times of the ILP model and the heuristic considering all scenarios were of, respectively, 8 minutes and 41 seconds and 1 minute

and 21 seconds.

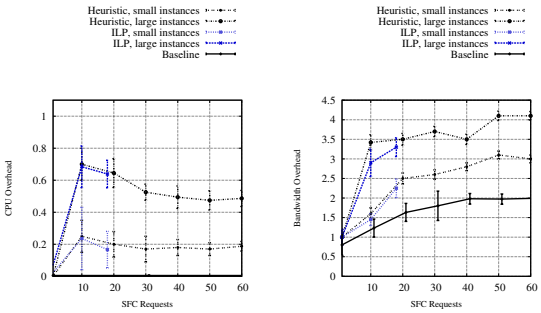
Last, we evaluate our heuristic approach on a large NFV infrastructure. In this experiment, we consider a physical network with 200 N-PoPs and a maximum of 60 SFC components of type 4. The delay limit was scaled up to 90ms in order to account for the larger network size. Figure 8(a) depicts the average time needed to find a solution using both the ILP model and the heuristic. The ILP model was not able to find a solution in a reasonable time in scenarios with more than 18 SFCs (the solution time was longer than 48 hours). The heuristic approach, in turn, is able to properly scale to cope with this large infrastructure, delivering feasible, high-quality solutions in a time frame of less than 30 minutes. As in previous experiments, small network function instances lead to higher solution times than large ones. This is mainly because smaller instances lead to a larger space of potential solutions to be explored.

Although the heuristic does not find the optimal solution (due to time constraints), Figures 8(b), 8(c), 8(d) and 8(e) show that the solutions obtained through this approach present a similar level of quality to the ones obtained optimally. Figure 8(b) depicts the average number of instantiated network functions with the number of SFC requests varying from 1 to 60. As in previous experiments, the number of instances remains proportional to the number of SFC requests. Smaller instance sizes lead to a higher number of network functions being instantiated. Considering small sizes, 75 network functions instances are required on average. In contrast, for large sizes, 40 instances are required on average. Figure 8(c), in turn, illustrates the average CPU overhead. For small instances, CPU overhead is limited to 18.77%, while for large instances it reaches 48.65%. Similarly to the results concerning the number of network function instances, CPU overheads in these experiments also follow the trends observed in previous ones. Next, Figure 8(d) presents bandwidth overheads. Small instances lead to a bandwidth overhead of 300%, while for large instances this overhead is, on average, 410%. These particularly high overheads are mainly due to the increase on the average length of end-to-end paths, as the physical network is significantly larger. Note that the bandwidth overhead observed in the baseline scenario (198%) is also significantly higher than those observed in experiments employed on the small infrastructure. Last, Figure 8(e) depicts the average end-to-end delay observed in large infrastructures. In line with previous results, the end-to-end delay tends to be lower than the delay observed in the baseline scenario. The scenario considering Component 4 presents, on average, 17.72% lower delay than the baseline scenario (70.30ms compared to 82.76ms). In short, these results demonstrate that: *(i)* the heuristic is able to find solutions with a very similar level of quality as the optimization model for small infrastructures; and *(ii)* as both infrastructure sizes and the number of requests increase, the heuristic is able to maintain the expected level of quality while still finding solutions in a short time frame.



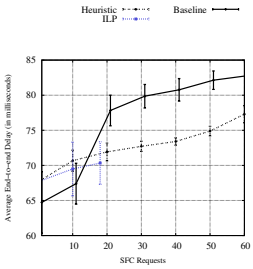
(a) Average time needed to find a solution.

(b) Average number of network function instances.



(c) Average CPU Overhead of network function instances.

(d) Average bandwidth overhead of SFCs deployed in the infrastructure.



(e) Average end-to-end delay of SFCs deployed in the infrastructure.

Fig. 8. Scenario considering a large infrastructure and components of type 4.

V. Conclusion

NFV is a prominent network architecture concept that has the potential to revamp the management of network functions. Its wide adoption depends primarily on ensuring that resource allocation is efficiently performed so as to prevent over- or under-provisioning of resources. Thus, placing network functions and programming network flows in a cost-effective manner while guaranteeing acceptable end-to-end delays represents an essential step towards a broader adoption of this concept.

In this paper, we formalized the network function placement and chaining problem and proposed an optimization model to solve it. Additionally, in order to cope with large infrastructures, we proposed a heuristic procedure that dynamically and efficiently guides the search for solutions performed by commercial solvers. We evaluated both optimal and heuristic approaches considering realistic workloads and different use cases. The obtained results show that the ILP model leads to a reduction of up to 25% in end-to-end delays and an acceptable resource over-provisioning limited to 4%. Further, we demonstrate that our heuristic scales to larger infrastructures while still finding solutions that are very close to optimality in a timely manner.

As perspectives for future work, we envision extending the evaluation of the proposed solutions by applying them to other types of SFCs and ISP topologies, as well as conducting an in-depth analysis of the inter-relationships between their parameters. Moreover, we intend to explore mechanisms to reoptimize network function placements, assignments, and chainings. Further, we intend to explore exact solutions for the problem, such as matheuristics.

REFERENCES

[1] 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*, 2014.

[2] D. A. Joseph, A. Tavakoli, and I. Stoica, "A policy-aware switching layer for data centers," in *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2008.

[3] T. Benson, A. Akella, and A. Shaikh, "Demystifying configuration challenges and trade-offs in network-based isp services," in *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2011.

[4] 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*, 2012.

[5] Network Functions Industry Specification Group, "Network function virtualisation (nfv): An introduction, benefits, enablers, challenges and call for action," in *SDN and OpenFlow World Congress*, 2012, pp. 1–16.

[6] 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*, 2014.

[7] S. Barkai, R. Katz, D. Farinacci, and D. Meyer, "Software defined flow-mapping for scaling virtualized network functions," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 2013.

[8] A. Basta, W. Kellerer, M. Hoffmann, H. J. Morper, and K. Hoffmann, "Applying nfv and sdn to lte mobile core gateways, the functions placement problem," in *Proceedings of the 4th Workshop on All Things Cellular: Operations, Applications and Challenges*, 2014.

[9] M. Yu, Y. Yi, J. Rexford, and M. Chiang, "Rethinking virtual network embedding: Substrate support for path splitting and migration," *SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 17–29, Mar. 2008.

[10] M. Chowdhury, M. R. Rahman, and R. Boutaba, "Vineyard: Virtual network embedding algorithms with coordinated node and link mapping," *IEEE/ACM Transactions on Networking*, vol. 20, no. 99, pp. 206–219, 2012.

[11] M. Rabbani, R. Pereira Esteves, M. Podlesny, G. Simon, L. Zambenedetti Granville, and R. Boutaba, "On tackling virtual data center embedding problem," in *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, May 2013, pp. 177–184.

[12] L. R. Bays, R. R. Oliveira, L. S. Buriol, M. P. Barcellos, and L. P. Gaspary, "A heuristic-based algorithm for privacy-oriented virtual network embedding," in *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, Krakow, Poland, May 2014.

[13] M. Dobrescu, K. Argyraki, and S. Ratnasamy, "Toward predictable performance in software packet-processing platforms," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2012.

[14] R. Albert and A.-L. Barabási, "Topology of evolving networks: Local events and universality," *Physical Review Letters*, vol. 85, pp. 5234 – 5237, Dec 2000.

[15] B.-Y. Choi, S. Moon, Z.-L. Zhang, K. Papagiannaki, and C. Diot, "Analysis of point-to-point packet delay in an operational network," *Computer Networks*, vol. 51, pp. 3812–3827, 2007.

APPENDIX C JOURNAL PAPER PUBLISHED AT ELSEVIER COMPUTER COMMUNICATIONS

- **Title**: *A fix-and-optmize approach for efficient and large scale virtual network function placement and chaining*
- **Journal**: Elsevier Computer Communications
- **Qualis**: A2

Abstract. Network Function Virtualization (NFV) is a novel concept that is reshaping the middlebox arena, shifting network functions (*e.g.* firewall, gateways, proxies) from specialized hardware appliances to software images running on commodity hardware. This concept has potential to make network function provision and operation more flexible and cost-effective, paramount in a world where deployed middleboxes may easily reach the order of hundreds. In spite of recent research activity in the field, little has been done towards efficient and scalable placement & chaining of virtual network functions (VNFs) – a key feature for the effective success of NFV. More specifically, existing strategies have either neglected the chaining aspect of NFV, focusing on efficient placement only, or failed to scale to hundreds of network functions. In this paper, we approach VNF placement and chaining as an optimization problem, and propose a fix-and-optimize-based heuristic algorithm for tackling it. Our algorithm incorporates a Variable Neighborhood Search (VNS) meta-heuristic, for efficiently exploring the placement and chaining solution space. The goal is to minimize required resource allocation, while meeting network flow requirements and constraints. We provide evidence that our algorithm is able to find feasible, high quality solutions efficiently, even in scenarios scaling to hundreds of VNFs.

Computer Communications 102 (2017) 67–77

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

A fix-and-optimize approach for efficient and large scale virtual network function placement and chaining

Marcelo Caggiani Luizelli, Weverton Luis da Costa Cordeiro*, Luciana S. Buriol, Luciano Paschoal Gaspary

Institute of Informatics – Federal University of Rio Grande do Sul, Av. Bento Gonçalves, 9500, 91.501-970 – Porto Alegre, RS, Brazil

ARTICLE INFO

Article history:
Received 31 March 2016
Revised 9 August 2016
Accepted 12 November 2016
Available online 14 November 2016

Keywords:
NFV
Network function placement
Service Function Chaining
Optimization
Fix-and-optimize
Variable Neighborhood Search

ABSTRACT

Network Function Virtualization (NFV) is a novel concept that is reshaping the middlebox arena, shifting network functions (*e.g.* firewall, gateways, proxies) from specialized hardware appliances to software images running on commodity hardware. This concept has potential to make network function provision and operation more flexible and cost-effective, paramount in a world where deployed middleboxes may easily reach the order of hundreds. In spite of recent research activity in the field, little has been done towards efficient and scalable placement & chaining of virtual network functions (VNFs) – a key feature for the effective success of NFV. More specifically, existing strategies have either neglected the chaining aspect of NFV, focusing on efficient placement only, or failed to scale to hundreds of network functions. In this paper, we approach VNF placement and chaining as an optimization problem, and propose a fix-and-optimize-based heuristic algorithm for tackling it. Our algorithm incorporates a Variable Neighborhood Search (VNS) meta-heuristic, for efficiently exploring the placement and chaining solution space. The goal is to minimize required resource allocation, while meeting network flow requirements and constraints. We provide evidence that our algorithm is able to find feasible, high quality solutions efficiently, even in scenarios scaling to hundreds of VNFs.

1. Introduction

Network Function Virtualization (NFV) is a novel trend in which middlebox functions (firewalling, caching, encryption, etc.) are shifted from specialized hardware to software-centric solutions. These, in turn, run on top of off-the-shelf commodity hardware. The benefits of NFV are manifold, reaching middlebox consumers, vendors, and operators. For example, it has potential to enable companies and organizations (consumers) reduce capital expenditures on middlebox hardware purchases. Network operators also benefit from flexible placement of virtual network functions (VNFs) across the infrastructure. Examples include smooth reconfiguration of flow chaining, given processing requirements, and resource (de)allocation/dimensioning, in response to variations in traffic demand. NFV even enables small industry players (vendors) to easily step in and develop customized, software-centric middlebox solutions, reducing their time-to-market.

There have been significant achievements in NFV, addressing aspects from effective planning and deployment [11,17] to efficient operation and management [6,9]. Nevertheless, NFV is a relatively new and yet maturing paradigm, with various research questions open. One of the most challenging is how to efficiently find a proper VNF placement and chaining. Each instance of this problem, also known as Service Function Chaining (SFC), involves ensuring that allocated functions satisfy flow processing requirements (placement), and steering flows across functions in a specific order (chaining). In this context, Software Defined Networking (SDN) can be seen as a convenient ally, due to its flexible flow handling capability, thus making placement and chaining technically easier.

In spite of the convenience of bringing together NFV and SDN, the VNF placement and chaining problem, under certain requirements and constraints, is NP-complete. This aspect becomes especially relevant if we consider that the number of middleboxes may scale to the order of hundreds. Some surveys have shown that certain very large networks have around 2000 middleboxes deployed [21], and that this number often compares to those of routers and switches in place [20]. Recent investigations have targeted the placement and chaining problem [11,12,17], however they scale to a few dozens of VNFs, or to a handful of SFC submitted in parallel (as it will be discussed later).

* Corresponding author.
E-mail addresses: mcluizelli@inf.ufrgs.br (M.C. Luizelli), weverton.cordeiro@inf.ufrgs.br, weverton.cordeiro@gmail.com (W.L. da Costa Cordeiro), buriol@inf.ufrgs.br (L.S. Buriol), paschoal@inf.ufrgs.br (L.P. Gaspary).

http://dx.doi.org/10.1016/j.comcom.2016.11.002
0140-3664/

As a major contribution to the state-of-the-art, in this paper we address scalability of VNF placement and chaining (VNFPC) – deemed a very important design requirement of real operational networks – by proposing a novel fix-and-optimize-based heuristic algorithm. It combines mathematical programming (Integer Linear Programming, ILP) and a meta-heuristic method (Variable Neighborhood Search, VNS [7]), generating a math-heuristic method [15], so as to produce high quality solutions for large scale network setups in a timely fashion. We provide evidences, supported by an extensive set of experiments, that our heuristic algorithm scales to environments comprised of hundreds of network functions. It produces timely results on average only 20% far from a computed lower bound, and outperforms the best existing algorithmic solution, quality-wise, by a factor of 5. As another important contribution, we prove that the VNFPC problem belongs to class NP-complete. As far as we are aware of, this is the first paper to formally validate such an important claim.

The remainder of this paper is organized as follows. In Section 2 we cover related works. In Section 3 we introduce a formal definition to the VNFPC problem, along with the optimization model that approaches it. Our math-heuristic algorithm for VNF placement and chaining is presented in Section 4. In Section 5 we discuss evaluation results, and also compare our approach with the state-of-the-art, whereas in Section 6 we close the paper with concluding remarks.

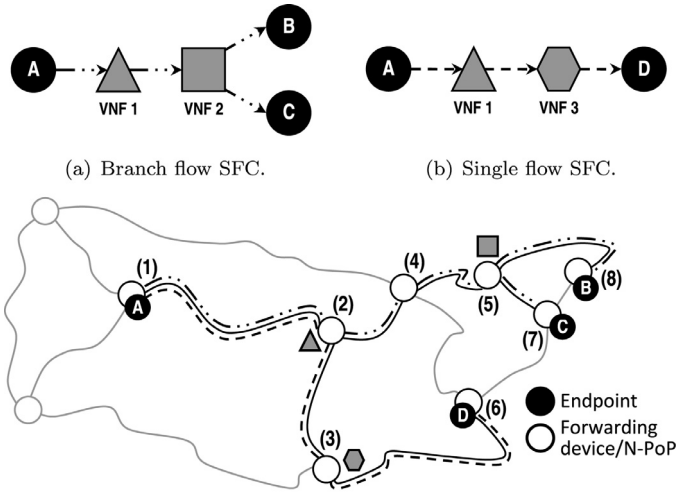
2. Related work

To the best of our knowledge, network function placement and chaining has not been investigated before the inception of NFV, *e.g.* for planning and deployment of physical middleboxes. One of the most similar problems in the networking literature is Virtual Network Embedding (VNE): how to deploy virtual network requests on top of a physical substrate [22,23]. In spite of the similarities, VNF placement and chaining is a broader, two-phase problem, as it also encompasses the assignment of SFCs to deployed functions.

Moens et al. [17] were the first to address VNF placement and chaining, by formalizing it as an optimization problem. The authors considered a hybrid scenario, where SFCs can be instantiated using existing middlebox hardware and/or virtual functions. Luizelli et al. [12] also approached the problem from an optimization perspective. The authors introduced a set of heuristics for pruning the search space, thus reducing the complexity of finding feasible solutions.

Lewin-Eytan et al. [11] followed a similar direction, and used an optimization model along with approximation algorithms to solve the problem. The focus however was VNF placement: where to deploy VNFs, and how to assign traffic flows to them. Their work is relevant for having established a theoretical background for NFV placement, building on two classical optimization problems, namely *facility location* and *generalized assignment*. Nonetheless, network function chaining (key for NFV planning and deployment) was left out of scope. Following the same line as the seminal work by Lewin-Eytan et al., Rost et al. [18] and Lukovszki et al. [13] also leveraged approximation algorithms for the VNFPC optimization problem, the former considering SFC admission control, whereas the latter covering incremental deployment.

There have been other proposals that approached specific versions of the VNFPC problem [2,10,14,16]. Mehraghdam et al. [16] and Bari et al. [2] introduced joint optimization problems for VNFPC, the former analyzing the optimization model under different objective functions, whereas the latter focusing on reducing operational expenditures on datacenters. Kuo et al. [10] analyzed resource consumption on physical servers and links, and Lukovszki and Schmid [14] presented a deterministic online algorithm with logarithm competitive ratio on the length of service chains.



(a) Branch flow SFC. (b) Single flow SFC.

(c) Network backbone, and deployment view of SFCs.

Fig. 1. Examples of SFCs, and partial view of the network backbone (focusing on the set of N-PoPs available for placing VNFs) considered in our scenario.

In spite of their potentialities, the investigations above either ignore a number of aspects vital for the placement & chaining problem (*e.g.* network functions deployed as part of a service chain, capacity constraints of network links, etc.) [13,18], or do not properly scale to large network settings and several SFCs submitted in parallel. Moens et al. [17], for example, were limited to small scale scenarios. Lewin-Eytan et al. [11] and Luizelli et al. [12] have shown to be only partially effective in scaling to scenarios with a few hundreds nodes and allocating network resources wisely (and the former does not approach chaining, as mentioned earlier). As it will be shown latter, our approach is able to outperform both, coming up with feasible, high quality solutions for larger scenarios in a timely fashion.

3. Problem overview and optimization model

We begin a more in-depth description of the placement and chaining problem with an example, illustrated in Fig. 1. It involves the deployment of two Service Function Chaining requests (SFCs, also referred to as “composite services”) onto a backbone network. For the first one, incoming flows must be passed through (an instance of) virtual network function (VNF) 1 (*e.g.* a firewall), and then VNF 2 (*e.g.* a load balancer). The second specifies that incoming flows must also be passed through VNF 1, and then VNF 3 (*e.g.* a proxy). Both SFCs are sketched in Figs. 1(a) and (b), respectively.

The VNF instances required for each SFC must be placed onto Network Points of Presence (N-PoPs). N-PoPs are infrastructures (*e.g.* servers, clusters or even datacenters) spread in the network and on top of which (virtual) network functions can be provisioned. Without loss of generality, we assume the existence of one N-PoP associated with every major forwarding device comprising a backbone network (see circles in Fig. 1(c)). Each N-PoP has a certain amount of resources (*e.g.* computing power) available. Likewise, each SFC has its own requirements. For example, functions composing an SFC (*e.g.* caching) are expected to sustain a given load, and thus are associated with a computing power requirement. Also, traffic between functions can be expected to reach some peak throughput, which must be handled by the physical path connecting the N-PoPs hosting those functions.

Given the context above, the problem we approach is finding a proper placement of VNFs onto N-PoPs and chaining of placed functions, so that overall network resource commitment is minimized. The placement and chaining must ensure that each of the

SFC requirements, as well as network constraints, are met. In our illustrating example, a possible deployment is shown in Fig. 1(c). The endpoints, represented as filled circles, denote flows originated/destinated from/to devices/networks attached to core forwarding devices. Observe that both SFCs share a same instance of VNF 1, placed on N-PoP (2), therefore minimizing resource allocation as desired.

3.1. Model description and notation

The optimization model is an important building block for the math-heuristic algorithm proposed in this paper. In this iteration of our work, we adopt a revised version of the model proposed by Luizelli et al. [12], which captures the placement and chaining aspects we are currently interested in. We start by describing both the input and output of the model, and establishing a supporting notation. We use superscript letters ^{P} and ^{S} to indicate symbols that refer to physical resources and SFC requests, respectively. Similarly, superscript letters ^{N} and ^{L} indicate references to N-PoPs/endpoints, and the links that connect them. We also use superscript ^{H} to denote symbols that refer to a subset (sub-graph) of an SFC request.

The optimization model we use for solving the VNFPC problem considers a set of SFCs Q and a physical infrastructure p, the latter is a triple $p = (N^P, L^P, S^P)$. N^P is a set of network nodes (either an N-PoP or a packet forwarding/switching device), and pairs $(i, j) \in L^P$ denote unidirectional physical links. We use two pairs in opposite directions (*e.g.*, (i, j) and (j, i)) to denote bidirectional links. The set of tuples $S^P = \{\langle i, r\rangle \mid i \in N^P \wedge r \in \mathbb{N}^*\}$ contains the actual location (represented as an integer identifier) of N-PoP i. Observe that more than one N-PoP may be associated to the same location (*e.g.* N-PoPs in a specific room or datacenter). The model captures the following resource constraints: computing power for N-PoPs (c^P_i), and one-way bandwidth and delay for physical links ($b^P_{i,j}$ and $d^P_{i,j}$, respectively).

Observe that the forwarding graph of a SFC may represent any topology. Figs. 1a and (b) illustrate topologies containing simple transitions and flow branches (note that flow joins may also be used). We assume, for simplicity, that the set of virtual paths available to carry traffic flows is known in advance. This is important because these paths enable determining end-to-end delays among pairs of endpoints, in our model. To illustrate, Fig. 1(a) shows two virtual paths: one starting in A and ending in B, and another starting in A and ending in C (both traversing VNFs 1 and 2).

It is important to emphasize that the set of virtual paths can be defined, for example, according to the network policy determined by the network operator. For example, a policy may allow traffic to go from and to all endpoints. In this case, our model requires all paths to be known in advance. Observe that *path* refer to the sequence of virtual network functions and endpoints that a specific traffic should pass through in a particular SFC. This is not to be confused with the routing path in the physical infrastructure, which is performed by the model – as shown next. This assumption does not restrict the model, neither increase its complexity, since finding paths (*e.g.*, the shortest one) is known to have polynomial time complexity.

The set of virtual paths of a SFC q is denoted by H_q. Each element $H_{q,i} \in H_q$ is one possible sub-graph of q, and contains one source and one sink endpoint, and only one possible forward path (as discussed above). The subsets $N^H_{q,i} \subseteq N^S_q$ and $L^H_{q,i} \subseteq L^S_q$ contain the VNFs and links that belong to $H_{q,i}$.

A SFC $q \in Q$ is an aggregation of network functions and chaining between them. It is represented as a triple $q = (N^S_q, L^S_q, S^S_q)$. Sets N^S_q and L^S_q contain the SFC nodes and virtual links connecting them, respectively. Each SFC has at least two endpoints, denoting specific locations in the infrastructure. The required locations of SFC endpoints is determined in advance, and given by a function $S^S_q : N^S_q \to \mathbb{N}^*$, where N^S_q contains endpoints and $\mathbb{N}^*$ represent possible identifiers of physical world locations. For each SFC q, we capture the following resource requirements: computing power required by a network function i ($c^S_{q,i}$), minimum bandwidth required for traffic flows between functions i and j ($b^S_{q,i,j}$), and maximum tolerable end-to-end delay (d^S_q).

F denotes the set of types of VNFs (*e.g.*, firewall, gateway) available for deployment. Each VNF has a set of U_m instances, and may be instantiated at most $|U_m|$ times (*e.g.* number of network functions of the same type available for deployment, having same requirements with respect to CPU usage, etc.). We denote as f_{type}: $N^P \cup N^S \to F$ the function that indicates the type of some given VNF, which can be either one instantiated in some N-PoP (N^P) or one requested in an SFC (N^S). We also use functions $f_{cpu} : (F \times U_m) \to \mathbb{R}_+$ and $f_{delay} : F \to \mathbb{R}_+$ to denote computing power requirement and processing delay of a VNF.

The model output is denoted by a 3-tuple $\chi = \{Y, A^N, A^L\}$. Variables from $Y = \{\, y_{i,m,j},\ \forall i \in N^P, m \in F, j \in U_m \,\}$ indicate a VNF placement, *i.e.* whether instance j of network function m is mapped to N-PoP i. The variables from $A^N = \{\, a^N_{i,q,j},\ \forall i \in N^P, q \in Q, j \in N^S_q \,\}$, in turn, represent an assignment of required network functions/endpoints. They indicate whether node j (either a network function or an endpoint), required by SFC q, is assigned to node i (either an N-PoP or another device in the network, respectively). Finally, variables from $A^L = \{\, a^L_{i,j,q,k,l},\ \forall (i, j) \in L^P, q \in Q, (k, l) \in L^S_q \,\}$ indicate a chaining allocation, *i.e.* whether the virtual link (k, l) from SFC q is being hosted by physical path (i, j). Each of these variables may assume a value in {0, 1}.

3.2. Model formulation

Next we describe the ILP formulation for the VNFPC problem. For convenience, Table 1 presents the complete notation used in the formulation. The goal of the objective function is to minimize the number of VNF instances mapped on the infrastructure. That choice was based on the fact that resource allocation accounts for a significant and direct impact on operational costs. It is important to emphasize, however, that other objective functions could be straightforwardly adopted (either exclusively or several functions combined). Examples include number of VNF instances deployed, overall bandwidth commitment, end-to-end delays, energy-aware deployments, survivability of SFCs, VNF load balancing, just to name a few. As constraint sets (1)-(11) (detailed next) ensure a feasible solution, any objective function being considered that does not depend on any other constraints should work properly. However, there are objective functions that might require some minor modifications to the model (*e.g.*, additional constraints) in order to work as expected (which is out of the scope of this work).

$$\textbf{minimize} \sum_{i \in N^P} \sum_{m \in F} \sum_{j \in U_m} y_{i,m,j}$$

subject to

$$\sum_{m \in F} \sum_{j \in U_m} y_{i,m,j} \cdot f_{cpu}(m, j) \leq c^P_i \qquad \forall i \in N^P \tag{1}$$

$$\sum_{q \in Q} \sum_{j \in N^S_q : f_{type}(j) = f_{type}(m)} c^S_{q,j} \cdot a^N_{i,q,j} \leq \sum_{j \in U_m} y_{i,m,j} \cdot f_{cpu}(m, j) \quad \forall i \in N^P,\ m \in F \tag{2}$$

$$\sum_{q \in Q} \sum_{(k,l) \in L^S_q} b^S_{q,k,l} \cdot a^L_{i,j,q,k,l} \leq b^P_{i,j} \qquad \forall (i, j) \in L^P \tag{3}$$

Table 1
Glossary of symbols and functions related to the optimization model.

Symbol	Formal specification	Definition
Sets and set objects		
p	$p = (N^P, L^P, S^P)$	Physical network infrastructure, composed of nodes and links
$i \in N^P$	$N^P = \{i \mid i$ is a N-PoP $\}$	Network points of presence (N-PoPs) in the physical infrastructure
$(i, j) \in L^P$	$L^P = \{(i, j) \mid i, j \in N^P\}$	unidirectional links connecting pairs of N-PoPs i and j
$\langle i, r\rangle \in S^P$	$S^P = \{\langle i, r\rangle \mid i \in N^P \wedge r \in \mathbb{N}^*\}$	identifier r of the actual location of N-PoP i
$m \in F$	$F = \{m \mid m$ is a function type$\}$	types of virtual network functions available
$j \in U_m$	$U_m = \{j \mid j$ is an instance of $m \in F\}$	instances of virtual network function m available
$q \in Q$		Service Function Chaining (SFC) requests that must be deployed
q	$q = (N_q^S, L_q^S, S_q^S)$	a single SFC request, composed of VNFs and their chainings
$i \in N_q^S$	$N^S = \{i \mid i$ is a VNF instance or endpoint $\}$	SFC nodes (either a network function instance or an endpoint)
$(i, j) \in L_q^S$	$L_q^S = \{(i, j) \mid i, j \in N^S\}$	unidirectional links connecting SFC nodes
S_q^S	$S_q^S : N_q^S \rightarrow \mathbb{N}^*$	function that determines the physical location of SFC endpoints
$H_{q,i}^H \in H_q^S$		distinct forwarding paths (subgraphs) contained in a given SFC q
$H_{q,i}^H$	$H_{q,i}^H = (N_{q,i}^H, L_{q,i}^H)$	a possible subgraph (with two endpoints only) of SFC q
$N_{q,i}^H$	$N_{q,i}^H \subseteq N_q^S$	VNFs that compose the SFC subgraph $H_{q,i}^H$
$L_{q,i}^H$	$L_{q,i}^H \subseteq L_q^S$	links that compose the SFC subgraph $H_{q,i}^H$
Parameters		
$c_i^P \in \mathbb{R}_+$		computing power capacity of N-PoP i
$b_{i,j}^P \in \mathbb{R}_+$		one-way link bandwidth between N-PoPs i and j
$d_{i,j}^P \in \mathbb{R}_+$		one-way link delay between N-PoPs i and j
$c_{q,i}^S \in \mathbb{R}_+$		computing power required for network function i of SFC q
$b_{q,i,j}^S \in \mathbb{R}_+$		one-way link bandwidth required between nodes i and j of SFC q
$d_q^S \in \mathbb{R}_+$		maximum tolerable end-to-end delay of SFC q
Functions		
$f_{type}(m)$	$f_{type}: N^P \cup N^S \rightarrow F$	type of some given virtual network function (VNF)
$f_{cpu}(m, j)$	$f_{cpu} : (F \times U_m) \rightarrow \mathbb{R}_+$	computing power associated to instance j of VNF type m
$f_{delay}(m)$	$f_{delay} : F \rightarrow \mathbb{R}_+$	processing delay associated to VNF type m
Variables		
$y_{i,m,j} \in Y$	$Y = \{y_{i,m,j}, \forall i \in N^P, m \in F, j \in U_m\}$	VNF placement
$a_{i,q,j}^N \in A^N$	$A^N = \{a_{i,q,j}^N, \forall i \in N^P, q \in Q, j \in N_q^S\}$	assignment of required network functions/endpoints
$a_{i,j,q,k,l}^L \in A^L$	$A^L = \{a_{i,j,q,k,l}^L, \forall (i, j) \in L^P, q \in Q, (k, l) \in L_q^S\}$	chaining allocation

$$\sum_{i \in N^P} a_{i,q,j}^N = 1 \qquad \forall q \in Q,\ k \in N_q^S \tag{4}$$

$$a_{i,q,k}^N \cdot l = a_{i,q,k}^N \cdot j \qquad \forall \langle i, j\rangle \in S^P,\ q \in Q,\ \langle k, l\rangle \in S_q^S \tag{5}$$

$$a_{i,q,k}^N \leq \sum_{m \in F} \sum_{j \in U_m : m = f_{type}(k)} y_{i,m,j} \qquad \forall i \in N^P,\ q \in Q,\ k \in N_q^S \tag{6}$$

$$\sum_{j \in N^P} a_{i,j,q,k,l}^L - \sum_{j \in N^P} a_{j,i,q,k,l}^L = a_{i,q,k}^N - a_{i,q,l}^N$$
$$\forall q \in Q,\ i \in N^P,\ (k, l) \in L_q^S \tag{7}$$

$$\sum_{(i,j) \in L^P} \sum_{(k,l) \in L_{q,t}^H} a_{i,j,q,k,l}^L \cdot d_{i,j}^P + \sum_{i \in N^P} \sum_{k \in N_{q,t}^H} a_{i,q,j}^N \cdot f_{delay}(k) \leq d_q^S$$
$$\forall q \in Q,\ (N_{q,t}^H, L_{q,t}^H) \in H_q \tag{8}$$

$$y_{i,m,j} \in \{0, 1\} \qquad \forall i \in N^P, m \in F, j \in U_m \tag{9}$$

$$a_{i,q,j}^N \in \{0, 1\} \qquad \forall i \in N^P, q \in Q, j \in N_q^S \tag{10}$$

$$a_{i,j,q,k,l}^L \in \{0, 1\} \qquad \forall (i, j) \in L^P, q \in Q, (k, l) \in L_q^S \tag{11}$$

The first three constraint sets refer to limitations of physical resources. Constraint set (1) ensures that, for each N-PoP, the sum of computing power required by all VNF instances mapped to it does not exceed its available capacity. Constraint set (2) certifies that the sum of required flow processing capacities does not exceed the amount available on a VNF instance deployed on a given N-PoP. Finally, constraint set (3) ensures that the physical path between the required endpoints has enough bandwidth.

Constraint sets (4)–(6) ensure the mandatory placement of all virtual resources. Constraint set (4) certifies that each SFC (and its respective network functions) is mapped to the infrastructure (and only once). Constraint set (5), in turn, seeks to guarantee that required endpoints are mapped to network devices in the requested physical locations. Constraint set (6) certifies that, if a VNF being requested by an SFC is assigned to a given N-PoP, then at least one VNF instance should be running (placed) on that N-PoP.

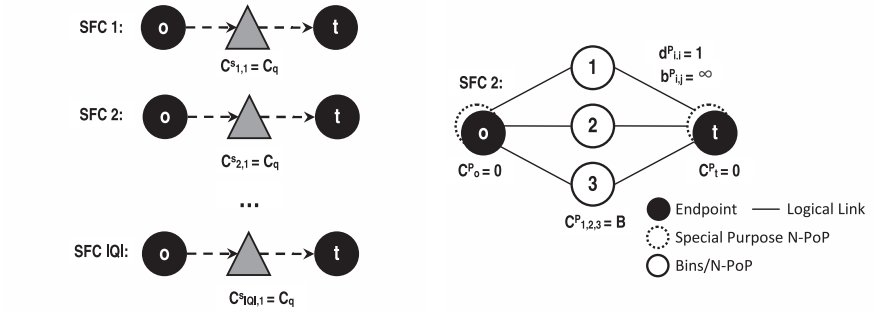
The constraints (7) refer to VNF chaining. Constraint set (7) ensures that there is an end-to-end path between required endpoints. Constraint set (8) certifies that latency constraints on mapped SFC requests are met for each path. The first part of the equation is a sum of the delay incurred by end-to-end latencies between mapped endpoints belonging to the same path. The second part defines the delay incurred by packet processing on VNFs that are traversed by flows in the same path.

3.3. Proof of NP-completeness

Although relatively simple for small instances, the VNF placement and chaining (VNFPC) problem belongs to complexity class NP-Complete. This aspect is further discussed next.

Lemma 1. *The VNFPC problem belongs to the class NP.*

Proof. A solution for the problem is the set of paths used to map the virtual links, as well as the nodes where the functions were placed. Endpoints have to be mapped to pre-defined nodes, and checking this is trivial. Moreover, resources consumed by all functions installed in each N-PoP cannot surpass their capacity. All di-



(a) BPP items represented as SFCs.

(b) NFV infrastructure created from Bin Packing instance.

Fig. 2. Bin Packing instance reduction to the VNFPC problem.

rected paths between two endpoints of an SFC must be mapped to a valid path in the infrastructure. Given the link mapping of the solution, it can be verified if each path starts and ends at the pre-defined nodes, forming a valid path. While traversing the mapped paths, the flow processing capacities of nodes can be accounted, the path delay computed, and it is verified if the path has enough bandwidth. Checking all that can be done in $O(n^3)$ per virtual network, since O(n) is spent traversing each path, and at most there n^2 paths per virtual network. Thus, a VNFPC solution can be verified in polynomial time, and the problem is then classified as NP. □

Lemma 2. *Any Bin Packing instance can be reduced to an instance of the VNFPC problem.*

Proof. An instance of the Bin Packing Problem (BPP), which is a classical NP-Complete problem [5], comprises a set Q of items, a size c_q for each item $q = 1, \ldots, |Q|$, a positive integer bin capacity B, and a positive integer k. The decision version of the BPP asks if there is a partition of Q into k disjoint sets $Q_1, Q_2, \ldots, Q_k$ such that the sum of sizes of the items in each subset is at most B. We reduce any instance of the BPP to an instance of the VNFPC using the following procedure:

1. $|Q|$ SFCs are created, each with exactly three nodes and two links. Each SFC has one origin endpoint, which must be mapped to N-PoP o, and one target endpoint, which must be mapped to t. The middle node is a VNF, which requires a computing power of c_q. Each link demands unitary bandwidth, and the maximum delay requirement of each SFC is set to two (see Fig. 2(a)).
2. An NFV infrastructure is created with $k + 2$ N-PoPs (nodes) and $2 \cdot k$ logical links (arcs). The k central N-PoPs have a capacity B, and are linked to two other special purpose N-PoPs, called o and t. The logical links have an arbitrarily large bandwidth and an insignificant delay (see Fig. 2(b)).

The reduction has polynomial time complexity $O(|Q|)$. □

Theorem 1. *VNFPC is an NP-Complete problem.*

Proof. By the instance reduction presented, if the BPP instance has a solution using k bins, then the VNFPC has a solution using k N-PoPs. Consider that each item of size c_q allocated to a bin j corresponds to place function from SFC q into N-PoP node j. Conversely, if the VNFPC has a solution using k N-PoPs, then the corresponding BPP instance has a solution using k bins. To place function from SFC q into N-PoP node j corresponds to allocate item of size c_q to a bin j. Lemmas 1 and 2 complete the proof. □

4. Fix-and-optimize heuristic for the VNF placement & chaining problem

As shown in Section 3.3, VNFPC optimization problem is NP-complete. To tackle this complexity and come up with high quality solutions efficiently, we introduce an algorithm that combines mathematical programming with heuristic search. Our algorithm is based on fix-and-optimize and Variable Neighborhood Search (VNS), techniques that have been successfully applied to solve large/hard optimization problems [7,8,19]. A comprehensive view of our proposal is shown in Algorithm 1.

4.1. Overview

In this section we provide a general view of our algorithm, using Fig. 3 as basis. It illustrates the search for a solution to the VNFPC problem, considering the SFCs and network infrastructure shown in Fig. 1.

In our algorithm, we first compute an initial, feasible solution (or *configuration*) χ to the optimization problem (see Section 4.3). In the example shown in Fig. 3, the initial configuration (1) has VNF 1 placed on N-PoPs 3 and 5; VNF 2 placed on N-PoP 5; and VNF 3 placed on N-PoP 3.

We then iteratively select a subset of N-PoPs, and enumerate the list of variables $y_{i,m,j} \in \mathcal{D} \subseteq Y$ related to them; variables listed in $\mathcal{D}$ will be subject to optimization, while others will remain unchanged (or fixed, hence fix-and-optimize). We take advantage of VNS to systematically build subsets of N-PoPs (Section 4.4). We also use a prioritization scheme to give preference to those subsets with higher potential for improvement (Section 4.5).

For each candidate subset of N-PoPs (processed in order of priority), we submit its decomposed set of variables $\mathcal{D}$ along with χ to a mathematical programming solver (Section 4.6). Here the goal is to obtain a set of values to those variables listed in $\mathcal{D}$, so that a better configuration is reached. We evaluate configurations according to the objective function of the model. In case there is no improvement, we rollback and pick the N-PoP subset that follows. We run this process iteratively until a better configuration χ' is found. Once it happens, we replace the current configuration with χ', and restart the search process (using χ' as basis). This loop continues until either we have explored the most promising combinations of N-PoP subsets, or a time limit is exceeded.

The loop explained above is illustrated in Fig. 3 through instances (1)–(10). Note that, for each instance, a different subset of N-PoPs (determined using VNS) is picked, and the resulting configuration (after optimized by the solver) is evaluated using the model objective function. For example, instance (6) failed as it violated delay constraints, whereas the other instances did not reduce resource commitment. When an improved configuration is found

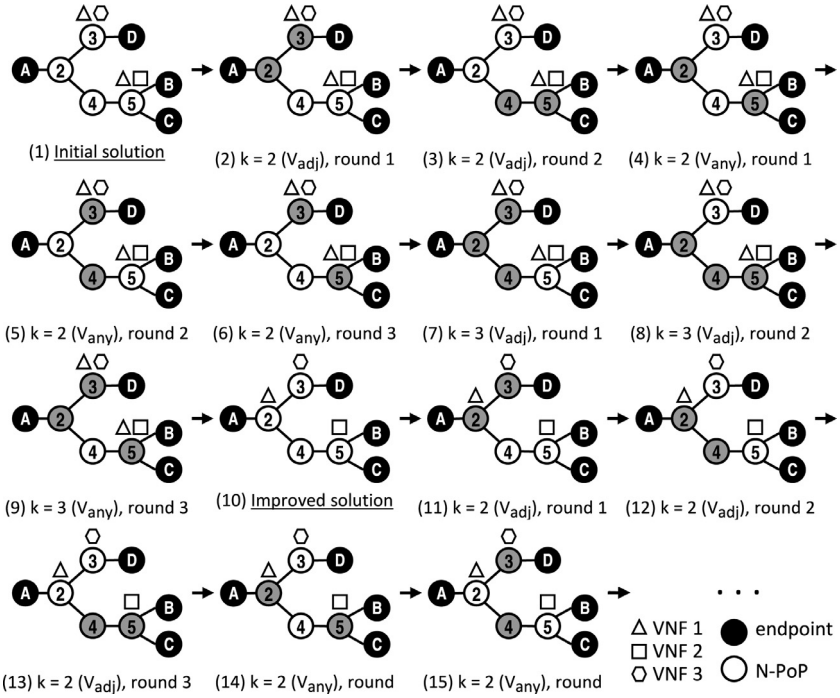


Fig. 3. We start with an initial solution (1), which satisfies both SFCs: VNF 1 is placed on N-PoPs 3 and 5, VNF 2 placed on N-PoP 5, and VNF 3 on N-PoP 3. We then explore the space of possible VNF assignments using VNS, starting with neighborhood size $k = 2$, and incrementing it in each iteration. For each round within a given iteration, we pick a 2-tuple of N-PoPs from the 2-neighborhood (gray nodes in each round), and evaluate those N-PoPs as a prospective better solution. Observe that we first evaluate tuples of N-PoPs which form a connected graph (instances (2) and (3)), and them the remainder tuples ((4)–(6)). We increment the neighborhood size to $k = 3$, after evaluating all tuples in the 2-neighborhood (note the transition from instance (6) to (7)). An improvement (*i.e.* a solution with fewer resource commitment) is reached in instance (9). The improved solution is shown in (10): it is obtained by removing instances of VNF 1 from N-PoPs 3 and 5, and placing a single VNF on N-PoP 2 (which now serves both SFCs). After that, we reset k and use the improved solution as basis for exploring another neighborhood.

(10), the search process is restarted, now taking as basis the configuration found.

4.2. Inputs and output

In addition to the optimization model input (described in the previous section), our algorithm takes five additional parameters. The first three are 1) a global execution time limit T_{global}, for the algorithm itself, 2) an initial neighborhood size $\mathcal{N}_{init}$, for the VNS meta-heuristic, and 3) an increment step $\mathcal{N}_{inc}$, for increasing the neighborhood size. These parameters are discussed in Section 4.4. The other parameters are 4) $NoImprov_{max}$, which represents the maximum number of rounds without improvement allowed per neighborhood, and 5) T_{local}, which indicates the maximum amount of time the solver may take for a single optimization run. They are discussed in Sections 4.5 and 4.6, respectively.

Our algorithm produces as output a configuration $\chi = \{Y, A^N, A^L\}$ for the VNFPC problem. This configuration may be null, in case a feasible solution does not exist, given the network topology in place and SFCs submitted.

4.3. Obtaining an initial configuration

The first step of our algorithm (line 1) is generating a configuration χ to serve as basis for the fix-and-optimize search. To this end, we remove the objective function, therefore turning the optimization problem into a factibility one. Then we use a commercial solver (CPLEX[1]) for solving it. The resulting configuration is one that satisfies all constraints, though not necessarily a high quality one, in terms of resources required. Observe that a solution to the VNFPC problem may not exist (line 2). In this case, no solution is returned. Note that heuristic procedures could be used for generating χ, instead of a solver. However, we observed in practice that CPLEX quickly finds an initial solution for this problem.

4.4. Variable Neighborhood Search

One important aspect of our algorithm is how to choose which subset of variables $\mathcal{D} \in Y$ will be passed to the solver for optimization. As mentioned earlier, we approach it using VNS. It enables organizing the search space in k-neighborhoods. Each neighborhood is determined as a function of the incumbent solution, and a neighborhood size k. In each round, a neighbor (a tuple having k elements) is picked, and used to determine which subset of the incumbent solution will be subject to optimization. In case an improvement is made, VNS uses that improved solution as basis, thus moving to a potentially more promising neighborhood (for achieving further improvements).

We build a neighborhood as a combination (unordered tuple) of any k N-PoPs, with at least one having a VNF assignment (line 5). Each tuple is then decomposed, resulting in the subset of variables $\mathcal{D} \in Y$ that will be subject to optimization (variable decomposition is discussed in Section 4.6). The reason we focus only on N-PoPs to build these tuples is that determining the assignment of required network functions ($a^N_{i,q,j}$) and chaining allocations ($a^L_{i,j,q,k,l}$) can be done straightforwardly once VNF placements ($y_{i,m,j}$) are defined.

The initial value for k is given as input to the algorithm, $\mathcal{N}_{init}$ (line 3). In the example shown in Fig. 3, we start with $k = 2$. In each iteration, we explore a neighborhood of size k (line 4) and, if no improvement is made, we increment k in $\mathcal{N}_{inc}$ units (line 21). Observe in Fig. 3 that, after exploring the 2-neighborhood space (sub-figures (2)-(6)), we make $k \leftarrow k + 1$ and start exploring

[1] http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/

Algorithm 1 Overview of the fix-and-optimize heuristic for the VNF placement and chaining problem.

Input: T_{global} global time limit
T_{local} time limit for each solver run
$\mathcal{N}_{init}$ initial neighborhood size
$\mathcal{N}_{inc}$ increments for neighborhood size
$NoImprov_{max}$ max. rounds without improvement
Output: χ: best solution found to the optimization model
1 $\chi \leftarrow$ initial feasible configuration
2 **if** a feasible configuration does not exist **then** fail **else**
3 $k \leftarrow \mathcal{N}_{init}$
4 **while** T_{global} is not exceeded **and** $k \leq |N^P|$ **do**
5 $V_k \leftarrow$ current neighborhood, formed of unordered tuples of k nodes only, one of them (at least) having a VNF assignment in the incumbent configuration χ
6 $V_{k,adj} \leftarrow$ neighbor tuples from V_k, whose nodes (for each tuple) form a connected graph
7 $V_{k,any} \leftarrow V_k \setminus V_{k,adj}$
8 **for** each list $\mathcal{V} \in \{V_{k,adj}, V_{k,any}\}$, while a better configuration is not found **do**
9 $NoImprov \leftarrow 0$
10 **while** $\mathcal{V} \neq \emptyset$ **and** $NoImprov \leq NoImprov_{max}$ **and** better configuration is not found **do**
11 $v \leftarrow$ next unvisited, highest priority neighbor tuple from $\mathcal{V}$
12 $\mathcal{D} \leftarrow$ decomposed list of variables $y_{i,m,u}$ from those nodes listed in neighbor tuple v
13 $\chi' \leftarrow$ configuration χ optimized by the solver, performed under time limit T_{local}, and making those variables not listed in $\mathcal{D}$ as fixed
14 **if** χ' is a better configuration than χ **then** update χ to reflect configuration χ'
15 **else**
16 $NoImprov \leftarrow NoImprov + 1$
17 **end if**
18 **end while**
19 **end for**
20 **if** no improvement was made **then** $k \leftarrow k + \mathcal{N}_{inc}$
21 **else** $k \leftarrow \mathcal{N}_{init}$ **end if**
22 **end while**
23 **return** χ
24 **end if**

a neighborhood composed of 3-tuples of N-PoPs (instance (7)). The highest value k may assume is limited by the number of N-PoPs.

On the event of an improvement, we reset the neighborhood size (to $\mathcal{N}_{init}$), and restart the search process. This is illustrated in Fig. 3, in the transition from instance (10) to (11). It is important to emphasize that same size neighborhoods do not imply in same neighborhoods. For example, consider subfigures (2) and (11) in Fig. 3. Although the 2-tuples are composed of the same N-PoPs, each belong to a different 2-neighborhood, as each is associated to a distinct configuration χ. The iteration over neighborhoods, and evaluation of k-tuples within in a neighborhood, continues until either T_{global} is exceeded, or k exceeds the number of N-PoPs in the infrastructure (line 4).

4.5. Neighborhood selection and prioritization

The time required by the solver to process a configuration χ and a subset $\mathcal{D} \subseteq Y$ is often small. However, processing every candidate subset $\mathcal{D}$ from the entire k-neighborhood can be cumbersome. For this reason, we prioritize those neighbor tuples that might lead to a better configuration, or whose processing might be relatively less complex.

Our heuristic for prioritizing tuples in the k-neighborhood set V_k relies on three key observations. First, solving VNF allocations/relocations becomes less complex when N-PoPs are adjacent. In other words, setting values for variables $a^N_{i,q,j}$ and $a^L_{i,j,q,k,l}$ is relatively less complex when N-PoPs i and j are directly connected through a physical path.

To explore the observation above in our algorithm, we break down a k-neighborhood set into two distinct ones. The first one (line 6) is formed by tuples whose participating N-PoPs form a connected graph. The second set (line 7) is formed by remaining tuples in V_k ($V_{k,any} = V_k \setminus V_{k,adj}$), *i.e.* those tuples whose participating nodes form a disconnected graph. Observe in Fig. 3 that, for each k-neighborhood, we first process the tuples of adjacent N-PoPs (V_{adj}): sub-figures (2) and (3); (7) and (8); and (11)–(13)). Then, we process the remainder tuples (V_{any}): sub-figures (2)-(6); (9); and (14)-(15).

The second key observation is that N-PoPs having a higher number of VNFs allocated, but fewer SFC requests assigned, are more likely candidates for optimization. Examples of such optimization are the removal of (some) VNFs allocated to those N-PoPs, or merge of those VNFs in a single N-PoP. We explore this observation by establishing a tuple priority, as a function of its residual capacity, and processing higher priority tuples first (line 11). The residual capacity of a tuple v is given by $r : V_k \rightarrow \mathbb{R}$, defined as a ratio of assigned VNFs to placed ones (according to Eq. (9)).

$$r(v) = \exp\left(\sum_{i \in N^P} \sum_{q \in Q} \sum_{j \in N^S_q} a^N_{i,q,j} - \sum_{i \in N^P} \sum_{m \in F} \sum_{j \in U_m} y_{i,m,j}\right) \tag{9}$$

The third observation is a complement of the previous one. As the priority of a neighbor decreases, it becomes less likely that it will lead to any optimization at all. The reason is trivial: lower priority tuples are often composed of overcommitted N-PoPs, for which removal/relocation of allocated/assigned VNFs is more unlikely. Therefore, when processing a neighborhood, we can skip a certain fraction of low priority neighbors, with a high confidence that no optimization opportunity will be missed.

To this end, our algorithm takes as input $NoImprov_{max}$. It indicates the maximum number of rounds without improvement that is allowed over a neighborhood subset. Before processing a given subset, we reset the counter of rounds without improvements, $NoImprov$ (line 9). This counter is incremented for every round in which no improvement is made (lines 14–17). We stop processing the current neighborhood subset once $NoImprov$ exceeds $NoImprov_{max}$ (line 10).

4.6. Configuration decomposition and optimization

Decomposing a configuration (*i.e.* building a subset of variables $\mathcal{D} \subseteq Y$ from χ for optimization) means enumerating all $y_{i,m,j}$ variables that can be optimized (line 12). It basically consists of listing all $y_{i,m,j}$ variables related to each N-PoP in the current neighbor tuple v, considering all network function types $m \in F$, and function instances available $j \in U_m$. Formally, we have $\mathcal{D} = \{y_{i,m,j} \in Y \mid i \in v\}$.

Once the decomposition is done, the incumbent configuration χ and set $\mathcal{D}$ are submitted to the solver (line 13). As mentioned earlier, the solver will consider variables listed in $\mathcal{D}$ as free (for optimization), and those not listed as fixed (*i.e.* no change in their values may be made). Observe that this restriction does not affect those variables related to the assignment of required network functions ($a^N_{i,q,j}$) and to chaining allocation ($a^L_{i,j,q,k,l}$).

We also limit each optimization run to a certain amount of time T_{local} (passed as parameter). This enables us to allocating a significant amount of the global time limit T_{global} on fewer but extremely complex χ and $\mathcal{D}$ instances.

5. Evaluation

In this section we evaluate the effectiveness of our math-heuristic algorithm in generating feasible solutions to the VNFPC problem, comparing its performance with those of polynomial (Lewin-Eytan et al. [11]) and non-polynomial approaches (namely, our previous work [12] and CPLEX optimization solver). We analyze placement & chaining solutions achieved, also discussing them from the perspective of a globally optimal one obtained with CPLEX (whenever they are feasible to compute). For the sake of comparing with Lewin-Eytan et al. [11], we have adjusted our experiment instances to be applied to their algorithm, without considering the chaining part (not approached in their research), thus enabling a fair comparison of results achieved. We also establish a lower bound for the VNFPC optimization, computed by relaxing output variables of the mathematical model and making them linear (instead of discrete), thus representing a solid reference for what the optimal solution to the VNFPC problem looks like, in a given large scenario.

We used CPLEX v12.4 for solving the optimization models, and Java for implementing the algorithms. The experiments were run on Windows Azure Platform – more specifically, an Ubuntu 14.04 LTS VM instance, featuring an Intel Xeon processor E5 v3 family, with 32 cores, 448 GB of RAM, and 6 TB of SSD storage. Next we describe our experiment setup, followed by results obtained.

5.1. Setup

The physical network substrate was generated with Brite,[2] following the Barabasi-Albert (BA-2) [1] model. The reason to use this model is the flexibility it enables for generating large enough physical infrastructures for the purposes of our evaluation. Observe that such infrastructures are in line with scenarios likely to adopt NFV, as network functions shift to cloud nodes highly distributed over the infrastructure [21]. Also, as we mentioned earlier, some surveys have shown that certain very large networks have around 2000 functions deployed [21], and that this number often compares to those of routers and switches in place [20]. Barabasi-Albert model is capable of generating network topologies that are comparable in size and complexity, thus enabling us to reasonably assess performance aspects of our algorithm (and weigh it against related ones).

The number of N-PoPs and physical paths between them varied in each scenario, ranging from 200 to 1000 N-PoPs, and from 793 to 3993 links. The computing power of each N-PoP is normalized and set to 1, and each link has bandwidth capacity of 10 Gbps. Average delay between any pair of N-PoPs is 90 ms, value adopted after a study from Choi et al. [3]. It is important to emphasize here that our goal was to assign latency values comparable to those observed in real-world settings, for all pairs of endpoints in the infrastructure. In this context, the values adopted ensure that there exists at least one delay-bounded path between any pair of nodes (i.e., the delay of all links used in a single path should be less than 90ms).

In order to fully grasp the efficiency and effectiveness of our approach, we carried out experiments considering a wide variety of scenarios and parameter settings. For the sake of space constraints, we concentrate on a subset of them. Our workload, for example, comprised from 1 to 80 SFCs. The topology of each SFC consisted of: a source endpoint and then a link (supporting 1 Gbps of traffic flow) to an instance of VNF X; a flow branch follows VNF X, each branch with 500 Mbps output flow, linking to a distinct instance of VNF Y; finally, each flow links to a distinct sink endpoint. Despite their simplicity, we argue that such service chains are enough to evaluate the scalability of our model and represent the usual service chains use cases (i.e., lines or simple bifurcation). For details regarding resource consumption for different service chains, we kindly refer the reader to our previous work [12]. Instances of VNF X require a normalized computing power capacity of 0.125 for small loads, and 1 for large loads, and have a processing delay of 0.6475 s. Instances of VNF Y require, in turn, 0.125 and 0.25 for small and large loads, and impose a processing delay of 7.0771s. VNF processing delays were determined after a study from Dobrescu et al. [4].

In the sections that follow, we adopted the following parameter setting for our approach: $T_{global} = 10,000$ s, $T_{local} = 200$ s, $\mathcal{N}_{init} = 2$, $\mathcal{N}_{inc} = 1$, and $NoImprov_{max} = 15$. In Section 5.4 we present an analysis of our approach considering other parameter settings.

5.2. Our heuristic algorithm compared to existing approaches

Fig. 4 provides an overview of achieved results, focusing on resource commitment of computed solutions, and time required to generate them. The main takeaway here is that our approach is able to generate significantly better solutions (closer to the lower bound) in a reasonable time, compared to existing ones. Observe in Fig. 4(a), for example, that our approach generated a solution distant at most 2.1 times from the lower bound, in a more extreme case with 1000 N-PoPs and 180 requested functions. This is a significantly smaller gap compared to other approaches, even considering their best cases. The measured distance was 4.97 times for Lewin-Eytan et al. [11], in the scenario with 200 N-PoPs and 120 requested functions, whereas for Luizelli et al. [12] it was 6.7 times, considering 200 N-PoPs and 60 functions.

Observe that finding an optimum solution (with CPLEX) is unfeasible even for very small instances (200 N-PoPs and less than 20 requested functions). Luizelli et al. [12] also failed short in scaling to such small instances, requiring more than 10k s for those around 60 requested functions.

An important caveat regarding the time required to obtain a solution is that, for our approach, we measured the time elapsed until it generated the highest quality solution. Note however that our algorithm is allowed to continue running, until T_{global} is passed or all neighborhoods are explored.

5.3. Qualitative analysis of generated solutions

Here we dive deeper on measuring the quality of generated solutions, analyzing their over-provisioning with regard to computing power and bandwidth. Observe from Fig. 5 that our proposal led to comparatively higher quality solutions, minimizing the amount of allocated but unused computing power in N-PoPs. By “unused” we mean that resources were allocated to VNFs deployed on N-PoPs, but the capacity of the VNFs is not fully consumed by assigned SFCs.

The performance of Lewin-Eytan et al. [11] is worse in this aspect mostly because their goal is not only minimize resource commitment, but also the distance (hops) a flow must traverse to reach a required VNF. It is also important to emphasize that their theoretical model has simplifications, which result in solutions that often extrapolate resource commitment. Since the authors did not approach VNF chaining in their model, bandwidth overhead could not be measured for their case.

[2] http://www.cs.bu.edu/brite/

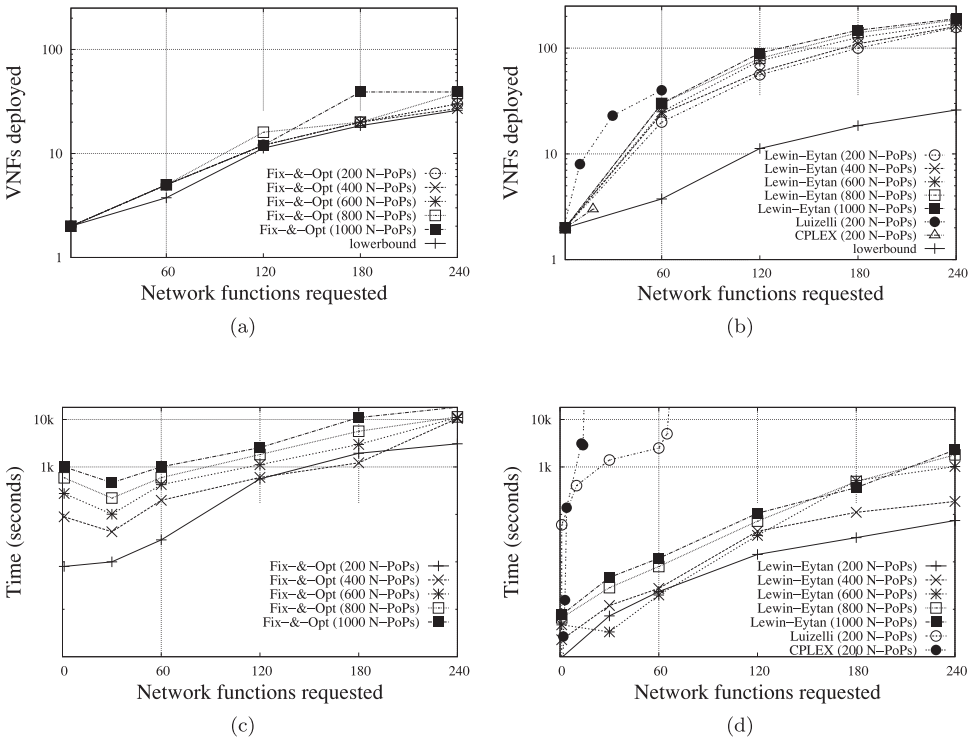


Fig. 4. Number of deployed VNFs and required time to compute a feasible placement and chaining.

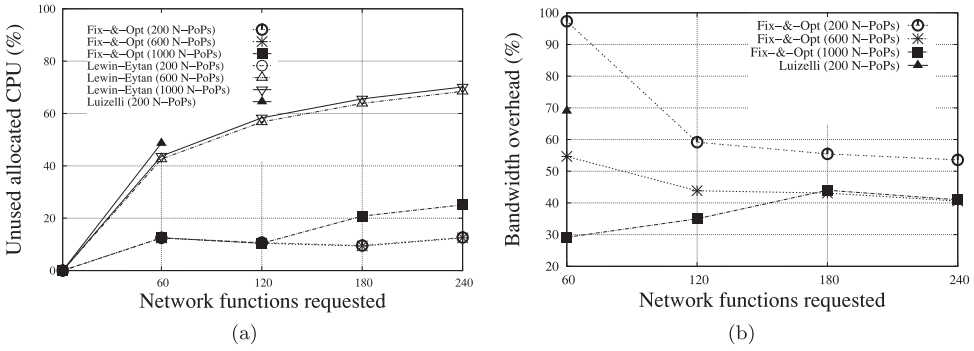


Fig. 5. Analysis of resource commitment (computing power and bandwidth) of each solution generated.

5.4. Sensitivity analysis

We conclude our analysis with a glimpse on experiments varying input parameter settings, whose results are summarized in Table 2. The cells in gray show how far our approach is from the lower bound (in terms of resources allocated), in a scenario with 600 N-PoPs and 120 functions. We focused on the following values for T_{global}: 600, 1,000, and 1500 s.; for T_{local}: 50, 60, and 100 s.; and for $\mathcal{N}_{init}$: 2, 4, and 6.

Observe in Table 2 that our approach was at most 5.24 times far from the lower bound, having allocated 59 VNFs (compared to a lower bound of 11.75 VNFs). More importantly, observe that such distance decreases as we provide more execution time for both the algorithm itself (T_{global}) and for each optimization instance (T_{local}). In the best case scenario, our approach was only 1.06 times distant from the lower bound, allocating the least number of VNFs (12).

Observe also a trade-off when setting T_{local}. On one hand, larger values enables solving each sub-problem instance with higher quality, as one may note in Table 2. On the other hand, smaller

Table 2
Assessment of the quality of generated solutions, and distance to lower bound, under various parameter settings.

	$\mathcal{N}_{init}$	T_{local} 50	factor	60	factor	100	factor
$T_{global} = 600$ sec.	2	59	5.24	51	4.53	51	4.53
	4	59	5.24	59	5.24	42	3.73
	6	59	5.24	59	5.24	33	2.93
	$\mathcal{N}_{init}$	T_{local} 50	factor	60	factor	100	factor
$T_{global} = 1{,}000$ sec.	2	32	2.84	16	1.42	15	1.33
	4	44	3.91	12	1.06	12	1.06
	6	40	3.55	18	1.60	12	1.06
	$\mathcal{N}_{init}$	T_{local} 50	factor	60	factor	100	factor
$T_{global} = 1{,}500$ sec.	2	20	1.77	12	1.06	12	1.06
	4	36	3.20	14	1.24	12	1.06
	6	36	3.20	15	1.33	12	1.06

Table 3
Assessment of the performance of the worst case scenario (i.e., 240 VNFs) for all instances sizes.

Instance size	Solution		Improv. factor	Iterations	Avg. T_{Local}	ratio (%)
	Initial	Best				
200	155	30	5.16	153	6	15.38
400	121	27	4.48	91	30	0.38
600	109	27	4.03	77	88	0.38
800	102	31	3.29	55	121	19.23
1000	92	38	2.42	60	130	46.15

values enable avoiding more complex sub-problem instances, for which CPLEX requires far more RAM memory to solve. It also leaves the algorithm with more global time available for exploring other promising sub-problem instances.

Finally, in Table 3 we illustrate how our algorithm improves the initial solution obtained with CPLEX, when running a feasibility version of the problem (as discussed in Section 4.3) in the worst case scenario, i.e., when deploying 240 VNFs altogether. One may observe that, on average, our approach is able to compute a solution up to 5.16 times better (regarding deployed VNFs) than the initial solution. In this table, we also included the number of iterations our algorithm runs until finding the best-reported solution. Note that it decreases as larger instances are handled by the algorithm since the time needed to compute each sub-problem increases with the instance size. Further, the observed ratio (%) between best-reported solution and lower bound is of 17.69 in the worst case. Recall that the lower bound for the VNFPC optimization is a solution computed by relaxing output variables of the mathematical model and making them linear (instead of discrete).

6. Final considerations

While Network Function Virtualization (NFV) is increasingly gaining momentum, with promising benefits of flexible service function deployment and reduced operations & management costs, there are several challenges that remain to be properly tackled, so that it can realize its full potential. One of these challenges, which has a significant impact on the NFV production chain, is effectively and efficiently deploying service functions, while ensuring that service level agreements are satisfied, and making wise allocation of network resources.

Amid various other aspects involved, Virtual Network Function Placement and Chaining (VNFPC) is key to fulfilling this challenge. VNFPC poses however an important trade-off between quality, efficiency, and scalability, that previously proposed solutions [11,12,17] have failed to satisfy simultaneously. This is no surprise, though, given that the complexity of solving this problem is NP-complete, as we have demonstrated.

In this paper, we combined mathematical programming and meta-heuristic algorithms to propose a novel approach for solving VNFPC. The results achieved not only evidenced the potentialities of fix-and-optimize and Variable Neighborhood Search as building blocks for systematically exploring the VNFPC solution space. More importantly, they have shown the significant improvement of our approach over the state-of-the-art – considering the quality (500% better solutions on average), efficiency (in the order of a couple of hours in the worst case), and scalability (to the order of hundreds) tripod.

In spite of the progresses achieved, much work remains. One promising direction is extending our approach to deal with constantly evolving network conditions. In such cases, assigned SFCs need to be reorganized (reassigned) in response to fluctuations in traffic demand (for example), so that service level agreements are not violated. We also intend addressing metrics (such as delay and throughput) that vary significantly throughout time – an essentially challenging problem.

References

[1] R. Albert, A.-L. Barabási, Topology of evolving networks: local events and universality, Phys. Rev. Lett. 85 (2000) 5234–5237.
[2] M.F. Bari, S.R. Chowdhury, R. Ahmed, R. Boutaba, On orchestrating virtual network functions, in: Proceedings of the 2015 11th International Conference on Network and Service Management (CNSM), 2015, pp. 50–56.
[3] B.-Y. Choi, S. Moon, Z.-L. Zhang, K. Papagiannaki, C. Diot, Analysis of point–to-point packet delay in an operational network, Comput. Netw. 51 (2007) 3812–3827.
[4] M. Dobrescu, K. Argyraki, S. Ratnasamy, Toward predictable performance in software packet-processing platforms, in: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NDSI 2012), 2012.
[5] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman & Co., New York, NY, USA, 1979.
[6] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, A. Akella, Opennf: enabling innovation in network function control, in: Proceedings of the 2014 ACM Conference on Computer Communications (SIGCOMM 2014), ACM, New York, NY, USA, 2014, pp. 163–174.
[7] P. Hansen, N. Mladenovi, Variable neighborhood search: principles and applications, Eur. J. Oper. Res. 130 (3) (2001) 449–467.
[8] S. Helber, F. Sahling, A fix-and-optimize approach for the multi-level capacitated lot sizing problem, Int. J. Prod. Econ. 123 (2) (2010) 247–256.
[9] J. Hwang, K.K. Ramakrishnan, 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 (NSDI 2014), 2014, pp. 445–458.
[10] T. Kuo, B. Liou, J. Lin, M. Tsai, Deploying chains of virtual network functions: on the relation between link and server usage, in: Proceedings of the IEEE International Conference on Computer Communications (INFOCOM 2016), San Francisco, USA, 2016.
[11] L. Lewin-Eytan, J. Naor, R. Cohen, D. Raz, Near optimal placement of virtual network functions, in: Proceedings of the IEEE International Conference on Computer Communications (INFOCOM 2015), IEEE, New York, NY, USA, 2015, pp. 1346–1354.
[12] M.C. Luizelli, L.R. Bays, L.S. Buriol, M.P. Barcellos, L.P. Gaspary, Piecing together the NFV provisioning puzzle: efficient placement and chaining of virtual network functions, in: Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management (IM 2015), IEEE, New York, NY, USA, 2015, pp. 98–106.
[13] T. Lukovszki, M. Rost, S. Schmid, It'S a match!: near-optimal and incremental middlebox deployment, SIGCOMM Comput. Commun. Rev. 46 (1) (2016) 30–36.
[14] T. Lukovszki, S. Schmid, Online Admission Control and Embedding of Service Chains, Springer International Publishing, Cham, pp. 104–118. doi:10.1007/978-3-319-25258-2_8.
[15] V. Maniezzo, T. Stützle, S. Voß, Hybridizing metaheuristics and mathematical programming, Ann.Inf. Syst. 10 (2009).
[16] S. Mehraghdam, M. Keller, H. Karl, Specifying and placing chains of virtual network functions, in: Proceedings of the 2014 IEEE 3rd International Conference on Cloud Networking (CloudNet), 2014, pp. 7–13.
[17] H. Moens, F. De Turck, Vnf-p: a model for efficient placement of virtualized network functions, in: Proceedings of the 10th International Conference on Network and Service Management (CNSM 2014), 2014, pp. 418–423.
[18] M. Rost, S. Schmid, Service chain and virtual network embeddings: approximations using randomized rounding, CoRR abs/1604.02180 (2016).
[19] F. Sahling, L. Buschkhl, H. Tempelmeier, S. Helber, Solving a multi-level capacitated lot sizing problem with multi-period setup carry-over via a fix-and-optimize heuristic, Comput. Oper. Res. 36 (9) (2009) 2546–2553.
[20] V. Sekar, S. Ratnasamy, M.K. Reiter, N. Egi, G. Shi, The middlebox manifesto: enabling innovation in middlebox deployment, in: Proceedings of the 10th ACM Workshop on Hot Topics in Networks (HotNets-X), ACM, New York, NY, USA, 2011, pp. 21:1–21:6.
[21] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, V. Sekar, Making middleboxes someone else's problem: network processing as a cloud service, ACM SIGCOMM Comput. Commun. Rev. 42 (4) (2012) 13–24.
[22] M. Yu, Y. Yi, J. Rexford, M. Chiang, Rethinking virtual network embedding: substrate support for path splitting and migration, ACM SIGCOMM Comput. Commun. Rev. 38 (2) (2008) 17–29.
[23] Y. Zhu, M. Ammar, Algorithms for assigning substrate network resources to virtual network components, in: Proceedings of the 25th IEEE International Conference on Computer Communications (INFOCOM 2006), 2006, pp. 1–12.

Marcelo Caggiani Luizelli is a Ph.D. candidate at the Institute of Informatics (INF) of the Federal University of Rio Grande do Sul (UFRGS), Brazil. He holds a M.Sc. degree in Computer Science from Federal University of Rio Grande do Sul (2014). Currently, he is a visiting student at the Computer Science Department of Technion University (Israel) and a visiting research student at Bell Labs (Israel). His research interests are computer networks, algorithms, and optimization on NFV and SDN.

Weverton Luis da Costa Cordeiro is a Postdoctoral Research Fellow at the Institute of Informatics of the Federal University of Rio Grande do Sul, Brazil. He holds a Ph.D. degree in Computer Science from the Federal University of Rio Grande do Sul (2014). His research interests include large scale distributed systems, information technology service management, software defined networks, network security and monitoring, future internet, and mobile ad hoc networks.

Luciana Salete Buriol received her B.Sc. in Computer Science (1998) from the Federal University of Santa Maria, Brazil. Her M.Sc. (2000) and Ph.D. (2003) were obtained at the State University of Campinas (UNICAMP), Sao Paulo, Brazil. In 2001 and 2002 she spent 15 months as a visiting scholar at the Algorithms and Optimization Research Department, AT&T Labs. Research, USA. In 2004 and 2005 she was a postdoc at the Computer Science Department, University of Rome, Italy. Since 2006 she is Associate Professor of Computer Science at the Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil. Her main research interests are in optimization and algorithms. Buriol is currently a CNPq (Brazilian National Research Council) Advanced Fellow, was President of ALIO in the period of 2012–2014, and vice-Presidente of IFORS (International Federation of Operational Research Societies) in the period 2016–2018.

Luciano Paschoal Gaspary holds a Ph.D. in Computer Science (PPGC/UFRGS, 2002) and serves as Associate Professor at the Institute of Informatics, UFRGS. From 2008 to 2014, he worked as Director of the National Laboratory on Computer Networks (LARC) and, from 2009 to 2013, was Managing Director of the Brazilian Computer Society (SBC). Prof. Gaspary has been involved in various research areas, mainly Computer Networks, Network Management and Computer System Security. He is author of more than 120 full papers published in leading peer-reviewed publications and has a history of dedication to research activities such as organization of scientific events, participation in the TPC of relevant IEEE, IFIP and ACM conferences and symposia, and participation as editorial board member of various journals. He is member of the SBC, IEEE, and ACM. More information about Prof. Gaspary can be found at http://www.inf.ufrgs.br/~paschoal/.

APPENDIX D PAPER PUBLISHED AT IFIP/IEEE IM 2017

- **Title**: *The Actual Cost of Software Switching for NFV Chaining*
- **Conference**: IFIP/IEEE Integrated Network Management Symposium (IM 2017)
- **Type**: Main track (full-paper)
- **Qualis**: A2
- **Date**: May 8-12, 2017
- **Held at**: Lisbon, Portugal

Abstract. *Network Function Virtualization* (NFV) is a novel paradigm that enables flexible and scalable implementation of network services on cloud infrastructure. An important enabler for the NFV paradigm is software switching, which should satisfy rigid network requirements such as high throughput and low latency. Despite recent research activities in the field of NFV, not much attention was given to understand the costs of software switching in NFV deployments. Existing approaches for traffic steering and orchestration of virtual network functions either neglect the cost of software switching or assume that it can be provided as an input, and therefore real NFV deployments of network services are often suboptimal. In this work, we conduct an extensive and in-depth evaluation that examines the impact of service chaining deployments on *Open vSwitch* – the de facto standard software switch for cloud environments. We provide insights on network performance metrics such as throughput, CPU utilization and packet processing, while considering different placement strategies of a service chain. We then use these insights to provide an abstract generalized cost function that accurately captures the CPU switching cost of deployed service chains. This cost is an essential building block for any practical optimized placement management and orchestration strategy for NFV service chaining

The Actual Cost of Software Switching for NFV Chaining

Marcelo Caggiani Luizelli*
Federal University of
Rio Grande do Sul, Brazil
mcluizelli@inf.ufrgs.br

Danny Raz
Nokia, Bell Labs
danny.raz@nokia-bell-labs.com

Yaniv Sa'ar
Nokia, Bell Labs
yaniv.saar@nokia-bell-labs.com

Jose Yallouz*
Technion Israel Institute
of Technology, Israel
jose@tx.technion.ac.il

***Abstract*— *Network Function Virtualization* (NFV) is a novel paradigm that enables flexible and scalable implementation of network services on cloud infrastructure. An important enabler for the NFV paradigm is software switching, which should satisfy rigid network requirements such as high throughput and low latency. Despite recent research activities in the field of NFV, not much attention was given to understand the costs of software switching in NFV deployments. Existing approaches for traffic steering and orchestration of virtual network functions either neglect the cost of software switching or assume that it can be provided as an input, and therefore real NFV deployments of network services are often suboptimal.**

In this work, we conduct an extensive and in-depth evaluation that examines the impact of service chaining deployments on *Open vSwitch* – the de facto standard software switch for cloud environments. We provide insights on network performance metrics such as latency, throughput, CPU utilization and packet processing, while considering different placement strategies of a service chain. We then use these insights to provide an abstract generalized cost function that accurately captures the CPU switching cost of deployed service chains. This cost is an essential building block for any practical optimized placement management and orchestration strategy for NFV service chaining

I. Introduction

Traditional telecommunication and service networks heavily rely on proprietary hardware appliances (also called *middleboxes*) that implement *Network Functions* (NF). These appliances support a variety of NF ranging from security (e.g., firewall, intrusion detection/prevention system) through performance (e.g., caching and proxy [1]) to wireless and voice functions (e.g., vRAN and IMS [2]).

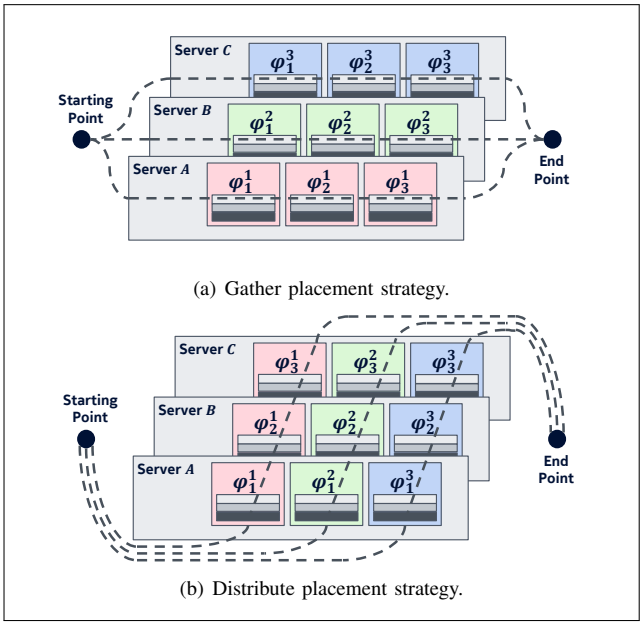
Due to the increasing demand for network services, providers are deploying an increasing number of NFs. This requires more and more hardware-based dedicated boxes, and thus deploying new NFs is becoming a challenging (and cumbersome) task – which directly leads to high operational costs. The emerging paradigm of Network Function Virtualization (NFV) aims to address the aforementioned problems by reshaping the architectural design of the network [2]. Essentially, NFV leverages traditional virtualization by replacing NFs with software counterparts that are referred to as Virtual Network Function (VNF). Virtualization technologies enable to consolidate VNF onto standard commodity hardware (e.g. servers, switches, and storage), and support dynamic situations in a flexible and cost-effective way (e.g., service provisioning on demand).

*Work done while in Nokia, Bell Labs.

In current hardware-based networks, operators are required to manually route cables and compose physical chains of middleboxes in order to provide services, a problem that is know as *Service Function Chaining* (SFC) [3]. The manual composition of SFC is error prone and static in nature (i.e., hard to change or physically relocate), and therefore expensive. On the other hand, NFV-based service chaining (*forwarding graph*), enables flexible placement and management of services, which allows a much more efficient utilization of resources, since the same computation resources can be consumed by different NFs in a dynamic way. Thus, a key expected success criteria for NFV is efficient resource utilization and reduced cost of operation.

However, placement of service chain in NFV-based infrastructure is not trivial, and introduces new challenges that need to be addressed. First, NFV is an inherently distributed network design based on small cloud nodes spread over the network. Second, even when considering a single (small) data center, network services might face performance penalties and limitations on critical network metrics, such as throughput, latency, and jitter, depending on how network functions are chained and deployed. This is one of the most interesting challenges for network providers in the shift to NFV, namely identify good placement strategies that minimize the provisioning and operation cost of deploying a service chain.

OpenStack [4] is the most popular open source cloud orchestration system, its scheduling (placement) component is called `nova-scheduler`, and can be utilized to place a sequence of VNFs. When doing so the resulting placement objective can be either load balancing (i.e., distributing VNFs between resources) or energy conserving (i.e., gathering VNFs on a selected resource). Consider the two placement strategies presented in Figure 1. In this example, we are given three identical servers A, B and C, and three identical service chains $\{\varphi^1, \varphi^2, \varphi^3\}$. Each service chain φ^i (for $i = 1, 2, 3$) is tagged with a fixed amount of required traffic and is composed of three chained VNFs: $\varphi^i = \langle \varphi^i_1 \rightarrow \varphi^i_2 \rightarrow \varphi^i_3 \rangle$. Figure 1(a) illustrates the first placement strategy (referred to as "gather"), where all VNFs composing a single chain are deployed on the same server. Note that in the gather case the majority of traffic steering is done by the server's internal virtual switching. Figure 1(b) illustrates the second placement strategy (referred to as "distribute"), where each VNF is deployed on a different server, and therefore the majority of traffic steering is done between servers by external switches.



(a) Gather placement strategy.

(b) Distribute placement strategy.

Fig. 1: Example of strategies to deploy 3 given service chains.

One can immediately see that these two placement strategies require from the VNF the same amount of resources to operate, however differ in the cost of their software switching (in terms of CPU consumption). Additionally, it is also not completely clear how would the network perform for each of the placement strategies, with respect to metrics such as throughput and latency. For instance, Figure 2 depicts the service latency (end-to-end), given a set of servers that are installed with either kernel-OvS or DPDK-OvS. Per each installed environment, we evaluate the two placement strategies discussed above. Obviously the latency increases as the chain size increases, however note that there is a non-negligible difference between the two placement strategies. When the servers are installed with kernel-OvS (resp. DPDK-OvS) the service latency of the distribute placement requires 50% more time (resp. 100% more time) than the service latency time required for the gather placement.

Going back to Figure 1, in order to decide which of the placement strategy is better, we need to identify the specific optimization criteria of interest. In this paper we focus on accurately estimating the virtual switching cost. For NFV-based infrastructure, virtual switching is an essential building block that on one hand enables flexible communication between VNFs; but on the other hand, it comes with a cost of resource utilization. Therefore assessing, understanding, and crafting a monolithic CPU cost function for software switching in a NFV-based environment is an extremely important task and a crucial step in order to: *(i)* efficiently guide VNF placement in a NFV-based infrastructure; *(ii)* understand how software switching impacts deployed services, namely analyze and comprehend how latency- and/or throughput-sensitive services behave; *(iii)* whenever possible, reduce the costs to the NFV provider and ultimately, foster the development of cost optimized deployment solutions.

In this paper, we develop a model to estimate the cost

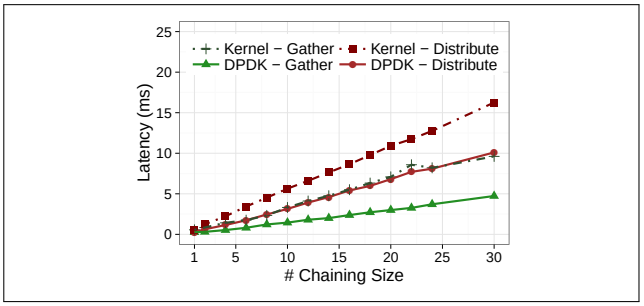


Fig. 2: End-to-end latency on different deployment strategies.

of software switching for different placement strategies over NFV-based infrastructure. We conduct an extensive and in-depth evaluation, measuring the performance and analyzing the impact of deploying a service chain on *Open vSwitch* – the de facto standard software switch for cloud environments [5]–[7]. Based on our evaluation, we then continue to craft a generalized abstract cost function that accurately captures the CPU cost of network switching. The main contributions of this paper can be summarized as follows: ***(i)* comprehensive evaluation of placement strategies of service chains** – based on a real NFV-based infrastructure, we examine our placement strategies, and assess performance metrics such as throughput, packet processing, and resource consumption[1]; and ***(ii)* model the cost of network switching** – given an arbitrary number of service chains, our model accurately predicts the CPU cost of the network switching they require.

The rest of the paper is organized as follows. In Section II we define our model and in Section III we provide an extensive and in-depth evaluation of software switching performance. In Section IV we analyze the results and build an abstract generalized cost function for our model. In Section V we discuss related works, and finally in Section VI we conclude our work and provide future directions.

II. Model Definition

In this section we define our model to estimate the cost of software switching for different placement strategies.

For a server S we denote by S^t the maximum throughput that server S can support (i.e., the wire limit of the installed NIC). Following the recommended best practice for virtualization intense environment ([8]), NFV servers require to be configured with two disjoint sets of CPU-cores, one for the hypervisor and the other for the VNF to operate. We denote by S^h (and S^v) the number of CPU-cores that are allocated and reserved solely for the hypervisor (and guests, respectively) to operate. The total number of CPU-cores installed in server S is denoted by S^c. Throughout this paper, unless explicitly saying otherwise, we assume that we are given a set of k servers $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$.

A *service chain* is defined to be an ordered set of VNFs $\varphi = \langle \varphi_1 \rightarrow \varphi_2 \ldots \rightarrow \varphi_n \rangle$. We denote by $\varphi|^n$ the length of the service chain, and define $\varphi|^p$ (and $\varphi|^s$) to be the number of packets per second (and average packet size, respectively) that

[1]Unlike other performance studies, our goal is *not* to delivers the best performance but rather to evaluate the performance of a given NF configuration.

service chain φ is required to process. Note that $\varphi|^p$ and $\varphi|^s$ are properties of service chain φ, namely given two service chains φ and ψ if $\varphi|^p \neq \psi|^p$ or $\varphi|^s \neq \psi|^s$ then $\varphi \neq \psi$. Throughout the paper, unless explicitly saying otherwise, we assume that we are given a set of m identical service chains $\Phi = \{\varphi^1, \varphi^2, \varphi^3, \ldots \varphi^m\}$, i.e. $\forall_{i,j=1..m} : \varphi^i = \varphi^j$.

We define $\mathcal{P} : \Phi \rightarrow S^k$ to be a *placement function* that for every service chain $\varphi \in \Phi$, maps every VNF $\varphi_i \in \varphi$ to a server $S_j \in \mathcal{S}$. We identify two particularly interesting placement functions:

(i) $\mathcal{P}_g$ – we call *gather* placement, where for every service chain the placement function deploys all VNFs $\varphi_i \in \varphi$ on the same server, namely:

$$\forall_{\varphi \in \Phi} \forall_{\varphi_i, \varphi_j \in \varphi} : \quad \mathcal{P}_g(\varphi_i) = \mathcal{P}_g(\varphi_j)$$

(ii) $\mathcal{P}_d$ – we call *distribute* placement, where for every service chain the placement function deploys each VNFs $\varphi_i \in \varphi$ on a different server, namely:

$$\forall_{\varphi \in \Phi} \forall_{\varphi_i, \varphi_j \in \varphi} : \quad i \neq j \rightarrow \mathcal{P}_d(\varphi_i) \neq \mathcal{P}_d(\varphi_j)$$

As explained above, both placement functions being considered, follow OpenStack and the objective goal implemented by `nova-scheduler` which can be either load balancing that distributes VNFs between resources, or energy conserving that gathers VNFs on a selected resource. Figure 1 illustrates our deployment strategies. Figure 1(a) depicts deployment of VNFs that follows the gather placement functions $\mathcal{P}_g$, and Figure 1(b) depicts deployment of VNFs that follows the distribute placement functions $\mathcal{P}_d$.

For a given placement function $\mathcal{P}$ that deploys all service chains in Φ on the set of servers $\mathcal{S}$, we define $\mathcal{C} : \mathcal{P} \rightarrow (\mathbb{R}^+)^k$ to be a *cpu-cost functions* that maps placement $\mathcal{P}$ to the required CPU per server. We say the cpu-cost function is *feasible* if there are enough resources to implement it. Namely, cpu-cost function $\mathcal{C}$ is feasible with respect to a given placement $\mathcal{P}$ if for every server $S_j \in \mathcal{S}$, the function do not exceed the number of CPU-cores installed in the server: $\forall_{S_j \in \mathcal{S}} : \mathcal{C}(\mathcal{P})_j \leq S_j^c$

Note that our main focus is the evaluation of the cost-functions with respect to the hypervisor performance, and therefore we evaluate deployments of VNFs that are fixed to do the minimum required processing, i.e. forward the traffic from one virtual interface to another.

III. Deployment Evaluation

In this section, we evaluate the performance of software switching for NFV chaining considering several performance metrics such as throughput, CPU utilization, and packet processing. We begin by describing our environment setup, followed by a discussion on performance metrics and possible bottlenecks. We evaluate two types of OvS installations: Linux kernel and DPDK; and compare between the two types of VNF placements: gather placement function $\mathcal{P}_g$, versus distribute placement function $\mathcal{P}_d$.

A. Setup

Our environment setup consists of two high-end HP ProLiant DL380p Gen8 servers, each server has two Intel Xeon E5-2697v2 processors, and each processor is made of 12 physical cores at 2.7 Ghz. One server is our *Design Under Test* (DUT), and the other is our traffic generator. The servers have two NUMA nodes, each has 192 GBytes RAM (total of 384 GBytes), and an Intel 82599ES 10 Gbit/s network device with two network interfaces (physical ports). We disabled HyperThreading in our servers in order to have more control over core utilization.

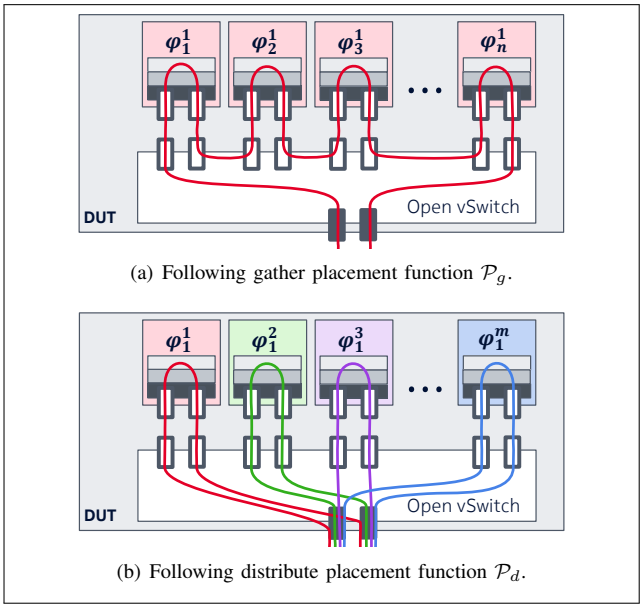
In both types of OvS installations (Linux kernel and DPDK), we isolate CPU-core and separate between two disjoint sets CPU-cores: $S^c = 24 = S^h + S^v$, i.e. CPU-cores used for management and for allocating resources for the VNFs to operate. This a priori separation between these two disjoint sets of CPU-cores plays an important role in the behavior of CPU consumption (and packet processing). In our experiments the size of the disjoint set of cores that are given to the hypervisor (S^h) varies between 2 to 12 physical cores, while the remainder is given to the VNFs (i.e. S^v ranges between 22 to 12). The exact values depend on the type of installation (Linux kernel or DPDK).

The DUT is installed with CentOS version 7.1.1503, Linux kernel version 3.10.0. All guests operating system are installed with Fedora 22, Linux kernel version 4.0.4 – running on top of qemu version 2.5.50. Each VNF is configured to have a single virtual CPU pinned to a specific physical core and 4GBytes of RAM. Network offload optimizations (e.g., `TSO`, `GSO`) in all network interfaces are disabled in order to avoid any noise in the analysis and provide a fair comparison between the two types of OvS installations. We evaluated Open vSwitch version 2.4 in both kernel mode and DPDK. For the latter, we compiled OvS against DPDK version 2.2.0 (without shared memory mechanisms – in compliance with rigid security requirements imposed by NFV). Packet generation is performed on a dedicated server that is interconnected to the DUT server with one network interface to send data, and another network interface to receive. In all experiments, we generate the traffic with Sockperf version 2.7 [9], that invokes 50 TCP flows.

Note that our objective is to provide an analytic model that captures the cost of software switching. In order to be able to examine and provide a clear understanding of the parameters that impact the cost of software switching, we need to simplify our environment by removing optimizations such as network offloads, and fixing resource allocation. For achieving optimal performance, the reader is referred to [8].

We evaluate the placement functions defined in Section II (i.e., the gather placement function $\mathcal{P}_g$ and the distribute placement function $\mathcal{P}_d$) on our DUT as follows. We vary the number of simultaneously deployed VNFs from 1 to 30 in each of the placement functions. All VNFs forward their internal traffic between two virtual interfaces (using Linux IP forwarding), which are configured in different sub-domains.

Figure 3 illustrates deployments of VNFs on a single server (our DUT). Figure 3(a) depicts the traffic flow of the gather placement function $\mathcal{P}_g$. Ingress traffic is arriving from the physical NIC to the OvS that forwards it to the first vNIC of the first VNF. The VNF then returns the traffic to the OvS through its second vNIC, and the OvS forwards the traffic to the following VNF, composing a chain that eventually egress the traffic through the second physical NIC. On the other hand,



(a) Following gather placement function $\mathcal{P}_g$.

(b) Following distribute placement function $\mathcal{P}_d$.

Fig. 3: Given a set of m service chains Φ, illustrating deployment of VNFs on a single server (our DUT).

Figure 3(b)) depicts the traffic flow of the distribute placement function $\mathcal{P}_d$. For this placement, it is the responsibility of the traffic generator to spread the workload between the VNFs. Ingress traffic arriving from the physical NIC to the OvS that forwards it to one of the VNFs through its first vNIC. The VNF then returns the traffic to the OvS through its second vNIC, that egress the traffic through the second physical NIC.

B. Evaluating Packet Intense Traffic

In order to discuss utilization of resources, we measure and analyze the results of our DUT for several configurations, while receiving intense network workload. To generate an intense network workload we generate traffic where each packet is composed of 100Bytes (avoid reaching the NIC's wire limitation). We examine the behaviour of throughput, packet processing and CPU consumption for increasing chain size, using both placement functions, i.e. $\mathcal{P}_g$ and $\mathcal{P}_d$.

The six graphs in Figure 4 (and Figure 5) depict CPU consumption, packet processing, and throughput performance on a DUT that is installed with kernel OvS (and DPDK-OvS, respectively). The three graphs on the top (Figures 4(a), 4(b), and 4(c) for kernel OvS and Figures 5(a), 5(b), and 5(c) for DPDK-OvS) present the results following the gather placement function $\mathcal{P}_g$ and the three graphs on the bottom (Figures 4(d), 4(e), and 4(f) for kernel OvS and Figures 5(d), 5(e), and 5(f) for DPDK-OvS) present the results following the distribute placement function $\mathcal{P}_d$.

We measure throughput by aiming at the total of traffic that the traffic generator successfully sends and receives after routing through the DUT. To measure packet processing we aggregate the number of received packets in all interfaces of the OvS (including TCP acknowledgments). For CPU consumption, we present the results for both the CPU-cores allocated to the hypervisor to manage and support the VNF and CPU-cores allocated for the VNFs to operate.

1) Packet Processing and Throughput: A key factor in the behavior of our environment is the a priori separation between two disjoint sets of CPU-cores (S^h and S^v). Thus, in our experiments, we vary values of CPU-cores that are allocated to the OvS (hypervisor). For ease of presentation, we show only several selected values.

Figure 4(a) depicts the average throughput for gather placement function $\mathcal{P}_g$, and Figure 4(d) for distribute placement function $\mathcal{P}_d$. For the case of distribute placement function $\mathcal{P}_d$, the more VNFs we deploy on the server, the average throughput increases, while for the gather placement function $\mathcal{P}_g$ the average throughput decreases. Figures 4(b) and 4(e) depict packet per second for both our placement function $\mathcal{P}_g$ and $\mathcal{P}_d$. The results show that OvS can seamlessly scale to support 5-10 VNF, however at that point OvS reaches saturation, we observe mild degradation when keeping increasing the number of deployed VNFs.

Figures 5(a) and 5(d) (and Figures 5(b) and 5(e)) depict average throughput (and packet per second, respectively) of our two placement function $\mathcal{P}_g$ and $\mathcal{P}_d$, on a DUT that is installed with OvS-DPDK. The behavior of OvS-DPDK is similar to that of the kernel OvS, with the exception of having better network performance.

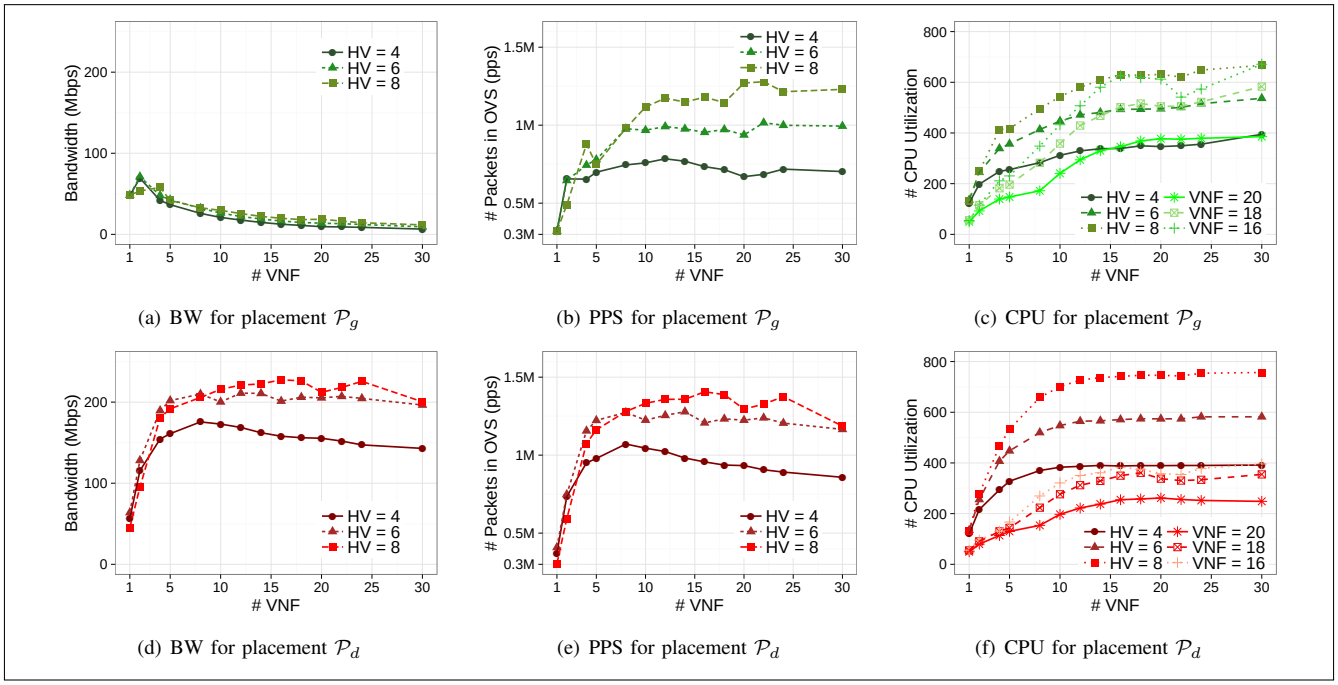
2) CPU Consumption: Figures 4(c) and 4(f) depict the average CPU consumption of OvS in Kernel mode. For both placement functions, the CPU consumption of the VNFs is bounded by the number of allocated cores for management S^h. A tighter bound of the CPU consumed by the VNFs (for networking) is a function of the CPU consumed by the hypervisor to manage the traffic, namely the total CPU consumed by the VNF is proportional to the total CPU consumed by the hypervisor in order to steer the traffic.

Comparing CPU utilization between the two placement functions, we observe that both placement strategies behave similarly for short service chains with little traffic. However, for longer service chains with increasing traffic requirements the behaviour of the two placements differs. Namely, in the gather placement, the CPU consumed by the VNF is almost identical to the CPU consumed by the hypervisor, whereas in the distribute placement the CPU consumed by the VNFs are 70% to 50% of the CPU consumed by the hypervisor. This observation suggests that in order to achieve CPU cost efficient placement, different traffic might require different placement strategies (as will be shown in the results of our analytical analysis in Section IV).

Figures 5(c) and 5(f) depict the average CPU consumption of DPDK-OvS. For both placement functions, the CPU consumed by the hypervisor is fixed and derived from the DPDK poll-mode driver (with the additional CPU-core for the management of other non-networking provision tasks). Similarly to the behavior observed for OvS in Kernel mode, we observe that CPU utilization (for the two placement functions) behave similarly to a small amount of VNFs and have correlation with the amount of traffic being handled (derived from throughput analysis).

C. Evaluating Throughput Intense Traffic

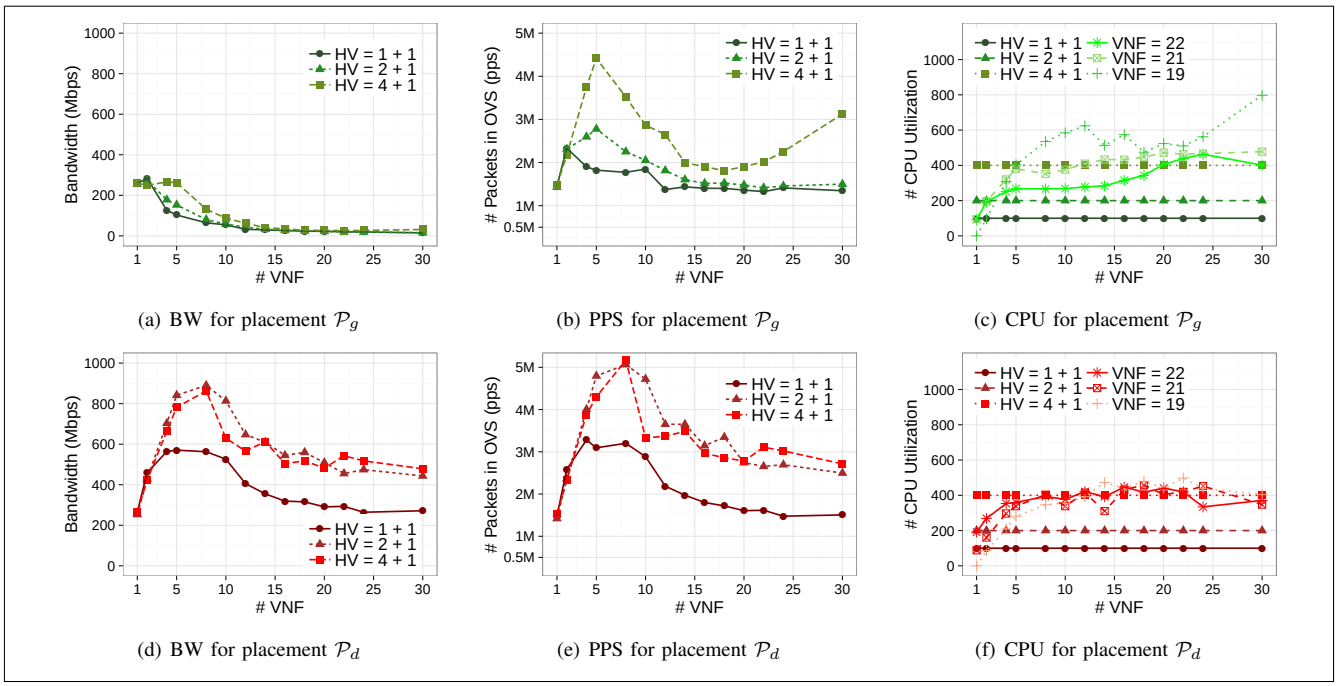
We reiterate the experiment presented in Subsection III-B in the context of achieving maximum throughput. Note that as



(a) BW for placement $\mathcal{P}_g$ (b) PPS for placement $\mathcal{P}_g$ (c) CPU for placement $\mathcal{P}_g$

(d) BW for placement $\mathcal{P}_d$ (e) PPS for placement $\mathcal{P}_d$ (f) CPU for placement $\mathcal{P}_d$

Fig. 4: Experiment results showing throughput, packet processing and CPU consumption for traffic generated in 100Bytes packets, that is examined over increasing size of chain, on a DUT that is installed with kernel OvS.

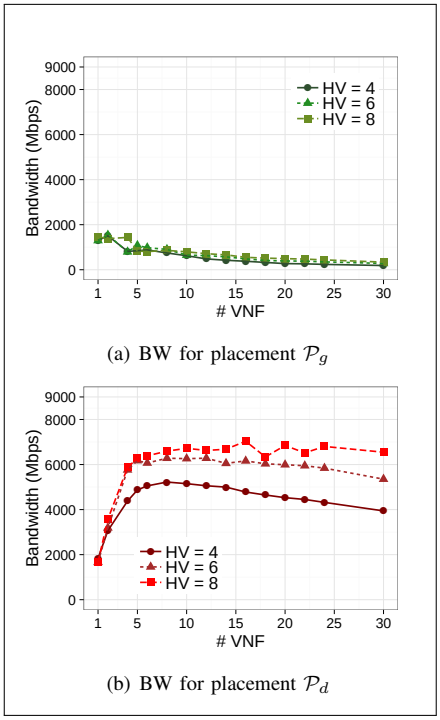


(a) BW for placement $\mathcal{P}_g$ (b) PPS for placement $\mathcal{P}_g$ (c) CPU for placement $\mathcal{P}_g$

(d) BW for placement $\mathcal{P}_d$ (e) PPS for placement $\mathcal{P}_d$ (f) CPU for placement $\mathcal{P}_d$

Fig. 5: Experiment results showing throughput, packet processing and CPU consumption for traffic generated in 100Bytes packets, that is examined over increasing size of chain, on a DUT that is installed with DPDK-OvS.

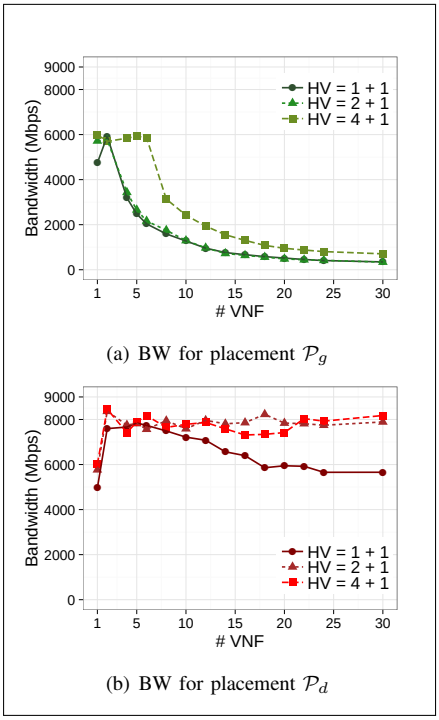
opposed to the previous section where we wanted to examine the CPU cost of the transferred traffic, here our goal is solely to maximize our throughput. In order to discuss maximum throughput, we measure and analyze the results of our DUT for several configurations, while receiving maximum transferable unit, i.e. we generate traffic where each packet is composed of 1500Bytes. We examine the behaviour of throughput for increasing chain size, using both placement functions $\mathcal{P}_g$ and $\mathcal{P}_d$. Since the behaviour of CPU consumption, and packet processing are similar to the behaviour observed for 100Bytes



(a) BW for placement $\mathcal{P}_g$

(b) BW for placement $\mathcal{P}_d$

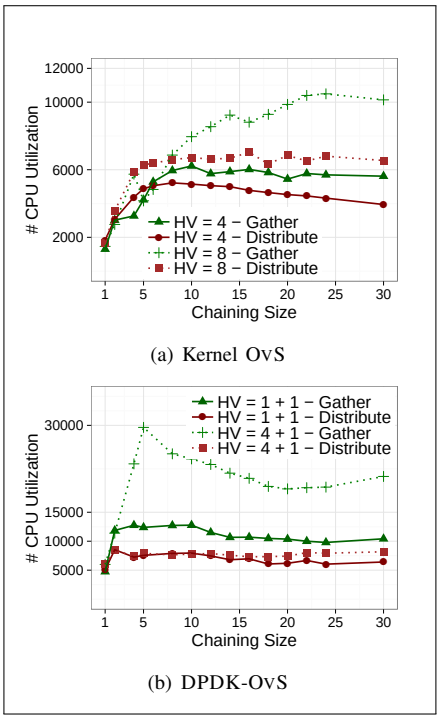
Fig. 6: Throughput for traffic generated in 1500Bytes packets, that is examined over increasing size of chain, on a DUT that is installed with kernel-OvS



(a) BW for placement $\mathcal{P}_g$

(b) BW for placement $\mathcal{P}_d$

Fig. 7: Throughput for traffic generated in 1500Bytes packets, that is examined over increasing size of chain, on a DUT that is installed with DPDK-OvS



(a) Kernel OvS

(b) DPDK-OvS

Fig. 8: Analysis of multiple servers, showing total throughput for traffic generated in 1500Bytes packets, that is examined over increasing size of chain

packet (in bigger scale), we omit their presentation.

Figure 6 (and Figure 7) depicts throughput performance on a DUT that is installed with kernel OvS (DPDK-OvS, respectively). Both Figures 6(b) for kernel-OvS, and 7(b) for DPDK-OvS show that the average throughput can scale up as long as the server is not over-provisioning resources (when deploying 1-5 VNFs). For the case of kernel OvS the bottleneck is the packet processing limit, while the NIC's wire limit is the bottleneck for the case of DPDK-OvS. The same effect can also be seen in Figures 6(a) and 7(a) where again as long as the server is not over-provisioning resources, the chain of VNFs is able to forward ∼0.8-1.5 Gbit/s in the case of kernel OvS, and ∼6 Gbit/s in the case of DPDK-OvS. Note that DPDK-OvS does not reach its packet processing limit (as it was seen in the case of 100-Byte packets – Subsection III-B). Instead, the observed limit is induced by the amount of packet processing that a single CPU core (allocated for the VNF) can perform (∼6 Gbit/s). This bottleneck can be mitigated if VNFs are set to have more than a single vCPU – that are configured to enable Receive Side Scaling (RSS).

So far, we presented the average throughput of a single server (our DUT). A naive straightforward analysis might lead to the wrong conclusion that the distribute placement function $\mathcal{P}_d$ outperforms the gather placement function $\mathcal{P}_d$. As can be seen in Figures 8(a) and 8(b), this is not the case. Figures 8(a) and 8(b) present the average overall throughput estimated by the model defined in Section II on a set of many servers that are installed with kernel OvS (DPDK-OvS, respectively).

IV. Monolithic Cost Function

Section III focuses on exhaustive evaluations, analyzing throughput, OvS packet processing and CPU consumption for different placement functions and software switching technologies (kernel OvS and DPDK-OvS). In the following, we describe how we build the generalized abstract cpu-cost functions, and then present and discuss the results.

A. Building an Abstract Cost Function

Based on the measured results on a single server (presented in Figures 4 - 5, and 6 - 7), we are now ready to craft an abstract generalized cost function that accurately captures the CPU cost of network switching.

We iterate the following process for both kernel OvS and DPDK-OvS. For each placement function ($\mathcal{P}_g$ or $\mathcal{P}_d$) and for each packet size (100Bytes or 1500Bytes), we split the construction and build a set of sub-functions that compose the cpu-cost function, namely $\mathcal{C} = \{\mathcal{C}_{gs}, \mathcal{C}_{ds}, \mathcal{C}_{gb}, \mathcal{C}_{db}\}$ where each is a cpu-cost sub-function for:

- $\mathcal{C}_{gs}$ – placement $\mathcal{P}_g$ and packet size 100Bytes.
- $\mathcal{C}_{ds}$ – placement $\mathcal{P}_d$ and packet size 100Bytes.
- $\mathcal{C}_{gb}$ – placement $\mathcal{P}_g$ and packet size 1500Bytes.
- $\mathcal{C}_{db}$ – placement $\mathcal{P}_d$ and packet size 1500Bytes.

Per each sub-function and for all measured service chain length, we sample the throughput (Figures 4(a) - 4(d), 5(a) -

5(d), 6(a) - 6(b), and 7(a) - 7(b)) to estimate the amount of packets that a single server can process, and correlate between the service chain length and the amount of packets. Next, we correlate the resulting packet processing, with the measured CPU consumption (Figures 4(c) - 4(f), 5(c) - 5(f)).

Finally, after extracting a 3-dimensional correlation between *(i)* service chains length; *(ii)* packet processing; and *(iii)* CPU consumption, we use the results to extract a set of cpu-cost functions using logarithmic regression, as follows:

$$log(\mathcal{C}_X) = \alpha \cdot log(\varphi|^n) + \beta \cdot log(\varphi|^p) + \gamma$$

Where $X \in \{gs, ds, bd, db\}$, namely gather or distribute placements for 100Bytes or 1500Bytes size of packets. Table I lists per each sub-function the coefficients α and β, and the constant factor γ.

	Kernel OvS			DPDK-OvS		
	α	β	γ	α	β	γ
$\mathcal{C}_{gs}$	0.586	0.858	-1.789	0.370	0.467	1.543
$\mathcal{C}_{ds}$	0.660	0.243	-2.661	0.217	0.091	3.795
$\mathcal{C}_{gb}$	0.752	0.979	-3.856	0.478	0.578	0.194
$\mathcal{C}_{db}$	1.009	0.268	-7.176	0.157	0.109	4.718

TABLE I: Coefficients α and β, and the constant factor γ, per each cpu-cost sub-function.

B. Insights

The values presented in Table I reflect the real CPU cost of the various deployments, but they provide very little insight regarding our motivation question (see Figure 1). In order to get a real understanding of this cost we provide graphs that depict the CPU cost for various service chains characterized by the length of the chain $\varphi|^n$, and the amount of packets to process $\varphi|^p$.

Figures 9(a), 9(b), and 9(c) depict the CPU cost for both placement functions when increasing the number of VNFs (and also the number of service chains) on servers that are installed with kernel-OvS, and traffic is received in large packets (1500Bytes per packet).

In all graphs, the CPU consumption is the total amount of CPU required on all physical machines to support the service chain (where the value 100 is a single CPU-core). For service chains with low packet processing requirements (10 Kpps - 50 Kpps), the cpu-cost function of the distribute placement $\mathcal{C}_{db}$, outperforms its gather placement counterpart $\mathcal{C}_{gb}$. However, as the requirement for packet processing increases (100 Kpps to 1.5 Mpps), the behaviour turns over and favors the gather placement cpu-cost function $\mathcal{C}_{gb}$.

In turn, Figures 10(a), 10(c), and 10(b) depict the CPU cost for both placement functions when increasing the number of VNFs (and also the number of service chains) on servers that are installed with DPDK-OvS, and traffic is received in 1500Bytes per packet. In this case the behaviour changes. For service chains with low packet processing requirements (10Kpps - 50Kpps), the cpu-cost function of the gather placement $\mathcal{C}_{gb}$, outperform its distribute placement counterpart $\mathcal{C}_{db}$. However, as the requirement for packet processing increases (1Mpps to 15Mpps), the behaviour turns over and favors the distribute placement cpu-cost function $\mathcal{C}_{db}$. Both results presented above show that deciding which of the placement strategy is better, depends on the required demand of packets to process, where the exact dependency varies according to the technology used.

Next we examine the cpu-cost function by varying the demand (i.e. required packets to process) of a service chain, for arbitrarily selected few service chain lengths.

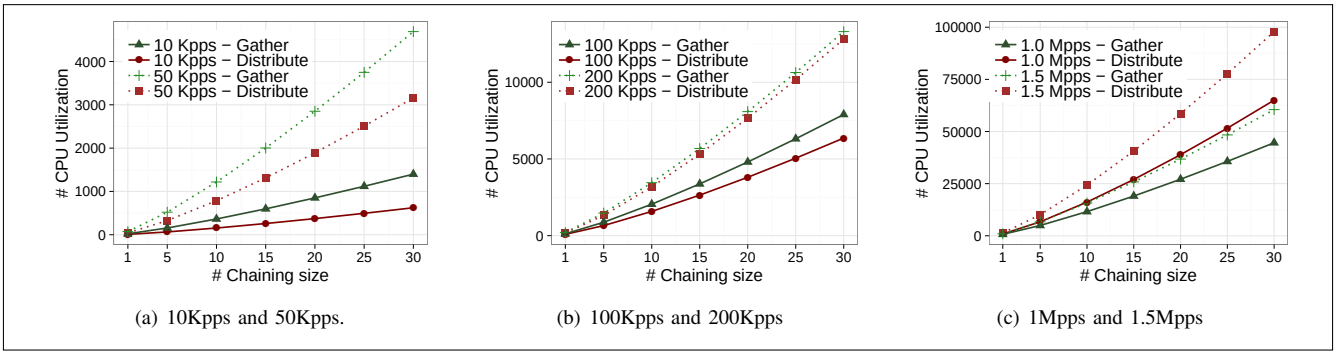
Figures 11 and 12 depicts the CPU cost for both placement functions when increasing the required number of packet to process (1500Bytes per packet) on servers that are installed with kernel OvS (Figure 11) and DPDK-OvS (Figure 12). In these graphs we focus on the definition of a feasible cpu-cost function. Thus, we normalize the cpu-cost values (i.e., the value 100% is now the total CPU-core on all servers), and scale the amount of required packets to process, in order to focus on the bounds. All values presented in the graphs are in log scale. The graphs reaffirm the results discussed in Figures 9 and 10, that is, deciding the appropriate placement strategy depends on the required network traffic demand to process.

Recall the definition of feasible cpu-cost function from Section II: a cpu-cost function is feasible if there are enough resources to implement it. In the result presented in Figures 11 and 12 the value 100 indicates that we have reached the processing bound and we cannot process more packets. We can observe that the infeasibility point is reached differently in each deployment strategy. For instance, the cpu-cost function of the distribute placement $\mathcal{C}_{db}$ reaches its infeasibility point faster than the gather placement $\mathcal{C}_{gb}$ when using OvS in Kernel model (Figure 11).

V. Related Work

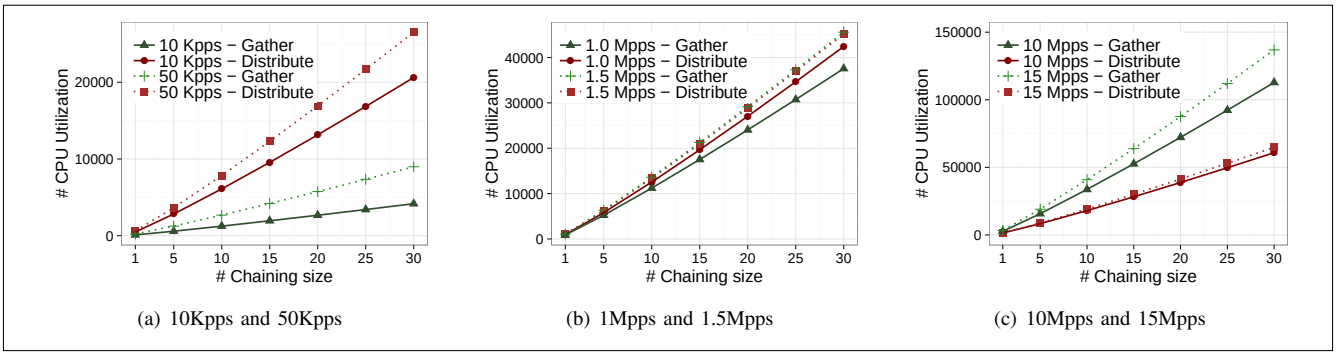
Network function placement. A proliferating field of interest in NFV is VNF placement [10]–[12] and chaining [13]–[17] strategies. The efficient orchestration of VNFs and its routing (or chaining) play a crucial role in the performance of deployed network services. In this regard, [12] focus on where to place VNFs and how to assign flows to them. In turn, [14], [15] continue and focus on joint optimization of VNF placement and chaining. Specifically, they defined optimized placement functions which take into account how VNFs are interconnected. However, all mentioned lines of work have neglected the cost of software switching and its limitations. In general, those works have focused on minimizing arbitrary cost functions. Therefore, the usage of such models in real NFV deployments might either lead to infeasible solutions or suffer form high penalties on the expected performance.

Middleboxes and Traffic steering. Both academia and industry show interest in virtualizing network appliances through a programmable, flexible and scalable architectures [18]–[21]. Traffic steering is an essential building block for enabling flexible NFV deployment. SDN is a complementary technology that enables the dynamic traffic steering between middleboxes and commodity servers [22]–[26]. However, most of the recent literature on SDN has analyzed inter-server



(a) 10Kpps and 50Kpps. (b) 100Kpps and 200Kpps (c) 1Mpps and 1.5Mpps

Fig. 9: Cpu-cost ranging over different service chain length ($\varphi|^n$), while receiving traffic generated in 1500Bytes packets, for both placement functions on servers that are installed with kernel OvS.



(a) 10Kpps and 50Kpps (b) 1Mpps and 1.5Mpps (c) 10Mpps and 15Mpps

Fig. 10: Cpu-cost ranging over different service chain length ($\varphi|^n$), while receiving traffic generated in 1500Bytes packets, for both placement functions on servers that are installed with DPDK-OvS.

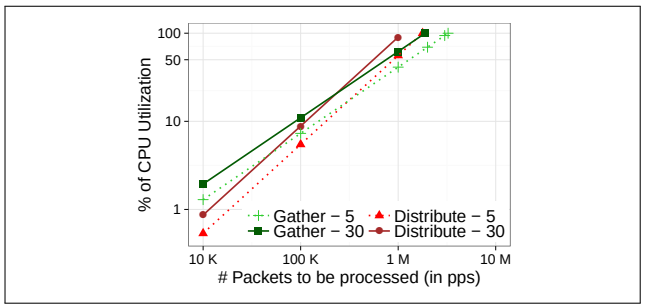


Fig. 11: Cpu-cost ranging over different packet processing requirements ($\varphi|^p$) 1500Bytes per packet, for both placement functions, on servers that are installed with kernel OvS.

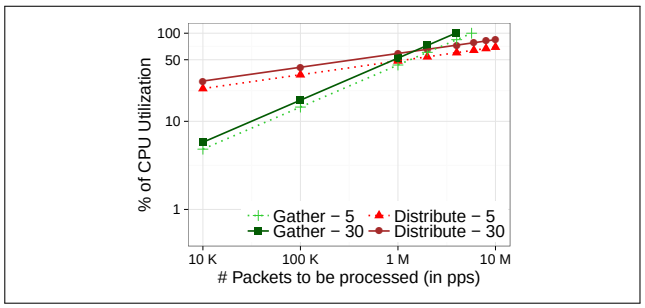


Fig. 12: Cpu-cost ranging over different packet processing requirements ($\varphi|^p$) 1500Bytes per packet, for both placement functions, on servers that are installed with DPDK-OvS.

traffic, rather than intra-server traffic – that is the focus of this work.

VI. Conclusion and Future Work

Software switching is a key component that enables the communication between VNFs. in any cloud based infrastructure. The rigid network requirements introduced by network function virtualization (i.e. high throughput and low latency) makes it a crucial component in that paradigm.

In this paper, we conduct an extensive and in-depth evaluation, measuring the performance and analyzing the impact of deploying service chains on a real NFV-based infrastructure. We provide insights on how latency, throughput, packet processing and CPU consumption behave when scaling up service chaining deployments. Furthermore, we develop a generalized cost function that accurately captures the CPU cost of software switching in this setting.

Understanding the costs and the limitations of software switching in NFV environments is a key ingredient in the ability to design efficient solutions for VNF management and orchestration – possibly leading to lower operational costs. Thus, a natural extension of this work is to develop cost-efficient service chain deployment schemes based on the devised CPU-cost function.

REFERENCES

[1] J. Martins, M. Ahmed, C. Raiciu, V. A. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, (NSDI 2014)*, April 2014, pp. 459–473.

[2] ETSI Industry Specification Group (ISG) Network Functions Virtualisation (NFV), "Network functions virtualisation," Available: http://www.etsi.org/technologies-clusters/technologies/nfv.

[3] P. Quinn and T. Nadeau, "Problem Statement for Service Function Chaining," Internet Requests for Comments, RFC Editor, RFC 7498, July 2015. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7498.txt

[4] "Openstack," http://www.openstack.org/, accessed: 04-19-2016.

[5] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker, "Extending networking into the virtualization layer," in *Proceedings of HotNets*, Oct. 2009.

[6] B. Pfaff, J. Pettit, T. Koponen, E. 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 Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, May 2015, pp. 117–130.

[7] "Open vswitch," http://www.openvswitch.org/, accessed: 06-09-2016.

[8] Intel, "Intel open network platform release 2.1: Performance test report," Internet Engineering Task Force, Mar. 2016, Available: https://01.org/packet-processing/intel®-onp.

[9] "Sockperf," https://github.com/Mellanox/sockperf/, accessed: 04-19-2016.

[10] S. Clayman, E. Maini, A. Galis, A. Manzalini, and N. Mazzocca, "The dynamic placement of virtual network functions," in *2014 IEEE Network Operations and Management Symposium (NOMS)*, May 2014, pp. 1–9.

[11] M. Ghaznavi, A. Khan, N. Shahriar, K. Alsubhi, R. Ahmed, and R. Boutaba, "Elastic virtual network function placement," in *Cloud Networking (CloudNet), 2015 IEEE 4th International Conference on*, Oct 2015, pp. 255–260.

[12] R. Cohen, L. Lewin-Eytan, J. S. Naor, and D. Raz, "Near optimal placement of virtual network functions," in *2015 IEEE Conference on Computer Communications (INFOCOM)*, April 2015, pp. 1346–1354.

[13] S. Mehraghdam, M. Keller, and H. Karl, "Specifying and placing chains of virtual network functions," in *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*, Oct 2014, pp. 7–13.

[14] M. C. Luizelli, L. R. Bays, L. S. Buriol, M. P. Barcellos, and L. P. Gaspary, "Piecing together the nfv provisioning puzzle: Efficient placement and chaining of virtual network functions," in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, May 2015, pp. 98–106.

[15] M. F. Bari, S. R. Chowdhury, R. Ahmed, and R. Boutaba, "On orchestrating virtual network functions," in *Proceedings of the 2015 11th International Conference on Network and Service Management (CNSM)*, ser. CNSM '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 50–56.

[16] W. Rankothge, J. Ma, F. Le, A. Russo, and J. Lobo, "Towards making network function virtualization a cloud computing service," in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, May 2015, pp. 89–97.

[17] M. Bouet, J. Leguay, and V. Conan, "Cost-based placement of vdpi functions in nfv infrastructures," in *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*, April 2015, pp. 1–9.

[18] A. Gember, R. Grandl, J. Khalid, and A. Akella, "Design and implementation of a framework for software-defined middlebox networking," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 467–468.

[19] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat, "xomb: Extensible open middleboxes with commodity servers," in *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '12. New York, NY, USA: ACM, 2012, pp. 49–60.

[20] 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.

[21] D. H. Anat Bremler-Barr, Yotam Harchol, "Openbox: A software-defined framework for developing, deploying, and managing network functions," in *Proceedings of the ACM SIGCOMM 2016 Conference on SIGCOMM*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 1–15.

[22] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "Simple-fying middlebox policy enforcement using sdn," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 27–38, Aug. 2013.

[23] Y. Zhang, N. Beheshti, L. Beliveau, G. Lefebvre, R. Manghirmalani, R. Mishra, R. Patneyt, M. Shirazipour, R. Subrahmaniam, C. Truchan, and M. Tatipamula, "Steering: A software-defined networking for inline service chaining," in *2013 21st IEEE International Conference on Network Protocols (ICNP)*, Oct 2013, pp. 1–10.

[24] 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 *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 543–546.

[25] G. W. Adiseshu Hari, T. V. Lakshman, "Path switching: Reduced-state flow handling using path information," in *Proceedings of the 11th International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '15. New York, NY, USA: ACM, 2015, pp. 1–7.

[26] J. R. A. S. J. R. Nanxi Kang, Monia Ghobadi, "Efficient traffic splitting on commodity switches," in *Proceedings of the 11th International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '15. New York, NY, USA: ACM, 2015, pp. 1–13.