Technische Universität München
Fakultät für Elektrotechnik und Informationstechnik
Lehrstuhl für Kommunikationsnetze

Towards Virtualization of Software-Defined Networks: Analysis, Modeling, and Optimization

Dipl.-Inf. Univ. Andreas A. Blenk

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender:		Prof. Dr.-Ing. Walter Stechele
Prüfer der Dissertation:	1.	Prof. Dr.-Ing. Wolfgang Kellerer
	2.	Prof. Dr. Rolf Stadler

Die Dissertation wurde am 21.11.2017 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 11.02.2018 angenommen.

Abstract

Today's networks lack the support to satisfy the highly diverse and fast changing demands of emerging applications and services. They have not been designed to differentiate among services or to adapt to fast changing demands in a timely manner. The paradigms Network Virtualization (NV) and Software-Defined Networking (SDN) can potentially overcome this impasse. In particular, the virtualization of software-defined networks is expected to bring dynamic resource sharing with guaranteed performance through NV and programmability through SDN; for the first time, tenants can program their requested networking resources according to their service demands in a timely manner. However, the virtualization of SDN-based networks introduces new challenges for operators, e.g., a virtualization layer that provides low and guaranteed control plane latencies for tenants. Moreover, tenants' expectations range from a fast, nearly-instantaneous provisioning of virtual networks to predictable operations of virtual networks.

This thesis presents a measurement procedure and a flexible virtualization layer design for the virtualization of software-defined networks. Focusing on the control plane, it introduces mathematical models for analyzing various design choices of virtualization layer architectures. For a fast and efficient virtual network provisioning on the data plane, this thesis proposes optimization systems using machine learning.

NV aims at providing the illusion of performance isolation; however, this is non-trivial and requires a well-designed virtualization layer. The challenging problem of such design is that the virtualization layer, and its realization through network hypervisors, can cause network and processing resource interference; such interference can lead to unpredictable network control for tenants. This thesis presents a tool for evaluating state-of-the-art network hypervisors with a focus on the control plane. The tool reports on the performance of network hypervisors in terms of provided control plane latency and CPU utilization. The measurement results are used to uncover sources of unpredictability due to tenants sharing networking and hypervisor resources. In order to mitigate the sources of unpredictability, this thesis presents and implements concepts for control plane isolation and adaptation support for the virtualization layer. Evaluation results show that the concepts can mitigate interference and provide adaptations; at the same time they reveal the impact of adapting virtualization layers at runtime, which can harm the stability of network operations.

Using network hypervisors introduces another key challenge: where to place hypervisor instances to achieve the best possible control plane performance? Similar to SDN controllers, network hypervisors are implemented in software. Therefore, they benefit from the flexibility of software-based implementations: they can be placed and executed on any commodity server. Placing network hypervisors, however, is an algorithmically hard problem. In contrast to the placement of SDN controllers, it even adds a new dimension: hypervisor must serve multiple virtual SDN networks. This thesis proposes mathematical models for optimizing network hypervisor placement problems. Solving the problems answers questions such as how many hypervisors are needed and where to deploy them among network locations. The models are designed for static and dynamic network traffic scenarios. Particularly for dynamic traffic scenarios, NV and SDN are expected to fully unfold their advantages with respect to network resource control.

The virtualization of software-defined networks also introduces new challenges with respect to resource allocation on the data plane: virtual networks can be requested and instantiated by service providers at any time. Consequently, infrastructure providers are in need for algorithms that fast and efficiently allocate physical network resources, i.e., embed virtual networks. Tenants would benefit from short virtual network provisioning times and less service interruptions - a competitive advantage for any operator of a virtualized infrastructure. This thesis investigates the potential of machine learning to provide more efficient and faster virtual network embedding solutions. It introduces a system that uses the power of neural computation; the system shrinks the overall problem space towards efficient and scalable solutions. Moreover, when operators are faced with recurring problem instances, machine learning can leverage the available data of previously solved problem instances to improve the efficiency of future executions of network algorithms.

Kurzfassung

Heutige Kommunikationsnetze können die stark variierenden und sich schnell ändernden Anforderungen durch immer unterschiedlichere Applikationen und Dienste nicht mehr effizient bewältigen. Netzprotokolle und Algorithmen wurden ursprünglich nicht dafür entwickelt, unterschiedliche Applikationen und Dienste differenziell zu behandeln und sich schnell an wechselnde Dienstanforderungen anzupassen. Neue Netzkonzepte wie Network Virtualization (NV) und Software-Defined Networking (SDN) bieten das Potential, die bestehenden Probleme, wie das Fehlen an Flexibilität und schneller Anpassbarkeit, zu lösen. Die Einführung beider Konzepte führt jedoch zu neuen Herausforderungen, die bewältigt werden müssen. Unter anderem müssen Virtualisierungsschichten für Netze geschaffen werden, welche eine geringe Latenz bzgl. der Steuerung der Netzressourcen erreichen. Außerdem haben Nutzer virtueller Netze durch die Einführung beider Konzepte hohe Anforderungen an die Betreiber der Kommunikationsnetze. Sie erwarten eine schnelle, nahezu augenblickliche Bereitstellung ihrer angeforderten virtuellen Netzressourcen.

Diese Dissertation präsentiert Messmethoden sowie den Entwurf einer flexibel anpassbaren Virtualisierungsschicht für die Virtualisierung Software-basierter Netze. Weiter führt diese Arbeit mathematische Modelle ein, welche es erlauben, die Planung der zentralen Steuerung der Virtualisierungsschicht zu optimieren und zu analysieren. Mit Hinblick auf die schnelle Bereitstellung von Netzressourcen beschäftigt sich diese Arbeit mit Optimierungsansätzen, welche Methoden aus dem Bereich des Maschinellen Lernens einsetzen.

NV zielt darauf ab, virtuelle Netze mit hoher und garantierter Dienstgüte bereitzustellen. Dies benötigt allerdings eine gut konzipierte Virtualisierungsschicht. Die Herausforderung liegt darin, dass sogenannte Netzhypervisoren, welche die Virtualisierungsschicht realisieren, selbst zu Überlagerungen der Netz- und Rechenressourcen und damit zu Garantieverletzungen virtueller Netze führen können. Diese Dissertation trägt mit der Entwicklung eines Messwerkzeuges dazu bei, die Steuerung von Virtualisierungsschichten zu untersuchen. Messungen zeigen dabei die Quellen für Ressourcenüberlagerungen innerhalb von Netzhypervisoren auf. Basierend auf diesen Beobachtungen wird ein Konzept einer Virtualisierungsschicht mit hoher Garantiegüte sowie Anpassungsfähigkeit vorgestellt. Messanalysen des Konzepts zeigen, dass durch gezielte Isolation virtueller Netze deren Garantien eingehalten werden können. Gleichzeitig geben die Messungen Aufschluss darüber, welchen Einfluss Anpassungen der Virtualisierungsschicht zur Laufzeit auf die zentrale Steuerung haben können.

Die Verwendung von Netzhypervisoren birgt weitere Herausforderungen. Unter anderem muss die Frage beantwortet werden, wie viele Netzhypervisoren notwendig sind und an welchen Orten

diese für eine optimale Latenz der Steuerungsschicht platziert werden müssen. Ähnlich wie die Steuerungseinheiten von SDN Netzen werden Netzhypervisoren vorwiegend in Software implementiert. Dadurch können sie die Flexibilität nutzen, welche durch Softwareimplementierung ermöglicht wird: Netzhypervisoren können überall im Netz auf Servern ausgeführt werden. Das zugrundeliegende Platzierungsproblem von Netzhypervisoren ist jedoch wie das Platzierungsproblem von Steuerungseinheiten in SDN Netzen algorithmisch schwer. Diese Arbeit stellt Optimierungsmodelle für die Platzierung von Netzhypervisoren vor. Die Lösungen der Optimierungsmodelle geben Antworten darauf, wie viele Hypervisoren wo genutzt werden müssen. Weiter werden die Modelle für statischen und dynamischen Netzverkehr untersucht. Besonders für Netze mit einem hohen Bedarf an dynamischer Anpassung wird erwartet, dass NV und SDN neue Möglichkeiten für eine bessere Nutzung der Netzressourcen bieten.

Die Verwendung von NV und SDN erhöht auch die Anforderungen an die Bereitstellung virtueller Netze: virtuelle Netze können jederzeit angefordert werden. Dazu benötigen Netzbetreiber jedoch effiziente und insbesondere schnelle Algorithmen, welche die angeforderten virtuellen Ressourcen im physikalischen Netz reservieren. Die Betreiber virtueller Netze profitieren generell von einer schnellen Bereitstellung – dementsprechend bieten schnelle Algorithmen Geschäftsvorteile für die Betreiber der physikalischen Infrastruktur. Diese Dissertation untersucht das Potential von Maschinellem Lernen, um schnellere und effizientere Reservierungen physikalischer Ressourcen für virtuelle Netze zu ermöglichen. Es werden Methoden vorgestellt, die mithilfe neuronaler Berechnung Optimierungsprobleme schneller lösen: durch die neuen Methoden kann der Suchraum von Optimierungsproblemen effizient eingeschränkt werden. Außerdem wird ein System vorgestellt, das die Lösungsdaten von Optimierungsproblemen nutzt, um die Effizienz von Algorithmen im Hinblick auf zukünftige Probleminstanzen zu steigern.

Contents

Chapter 1

Introduction

Communication networks such as the Internet, data center networks or enterprise networks have become a critical infrastructure of our society. Although these communications networks and their protocols have been a great success, they have been designed for providing connectivity in a best-effort manner. However, given the current shift away from human-to-human communication toward machine-to-machine communication, e.g., in the context of (distributed) cloud computing or Cyber-Physical Systems (CPSs), designing networks for best-effort transmission is no longer sufficient. The reasons are manifold: future applications like Internet-of-Things or robotics require communication networks providing Quality-of-Service (QoS) guarantees and predictable performance while they are sharing the same underlying network infrastructure. Whereas traditional communication networks have been planned and operated by humans, resulting in a rather slow operation and update, modern applications require fast and automatic changes to new requirements as those of future networking concepts such as 5G [AIS+14].

Indeed, traditionally it has been assumed that communication networks serve applications with homogeneous network resource requirements not changing over time. However, today's application requirements fluctuate on time scales from minutes to milliseconds and are possibly highly diverse with respect to their required network resources [BAM10; ER13; GDFM+12; GFM+12]. For example for CPSs, communication networks must serve latency-critical control loops where resource adaptations must be put into effect within millisecond timescales. Despite of their different requirements, applications typically share the same physical infrastructures. As a consequence, they rely on the same protocol stack. However, communication network infrastructures with their current protocol stacks lack adequate mechanisms to handle changing application requirements in a timely manner. Hence, today's communication networks lack the flexibility in providing efficient resource sharing with a high level of adaptability, needed to support demands with diverse network resource requirements changing over time. Overall, this results in a performance that is far from perfect for both the network operator and the network users.

Two paradigms, namely Network Virtualization (NV) [APS+05] and Software-Defined Networking (SDN) [MAB+08], are expected to cope with those requirements for flexible network resource sharing and adaptability. Whereas NV is seen as a key enabler to overcome the ossification of the Internet by introducing flexible resource sharing [APS+05], SDN introduces a new way of flexibly programming the shared resources at runtime [MAB+08].

NV abstracts physical resources of Infrastructure Providers (InP) and enables tenants, i.e., Service Providers (SP), to use virtual resources according to their users' demands. Due to NV, InPs and SPs can control physical and virtual resources respectively in a dynamic and independent manner. To gain the highest efficiency out of virtualized networks, InPs need mechanisms that quickly provide (virtual) network resources in a predictable and isolated manner. On the other side, SPs should be able to flexibly request and control their resources with high degree of freedom. Hence, NV opens a new path towards communication systems hosting multiple virtual networks of SPs.

SDN decouples control planes of network devices, such as routers and switches, from their data planes. Using open interfaces such as OpenFlow [MAB+08], SDN provides new means of network programmability [MAB+08]. With networks being completely programmable, SDN can realize Network Operating Systems (NOSs) integrating new emerging concepts; NOSs can be tailored to application-, service-, and user-specific demands. As an example, NOSs can integrate raising mechanisms from the research field of Artificial Intelligence (AI). This might lead to future NOSs, and communication networks that self-adapt to unforeseen events, e.g., based on knowledge that is inferred at runtime from the behavior of network users, network topologies, or the behavior of network elements.

Combining NV and SDN offers the advantages of both worlds: a flexible and dynamic resource acquisition by tenants through NV and a standardized way to program those resources through SDN. This is called *the virtualization of software-defined networks*, leading to the existence of multiple Virtual Software-Defined Networks (vSDNs) sharing one infrastructure [SGY+09; SNS+10; ADG+14; ALS14]. With both paradigms, it is expected that multiple vSDNs coexist while each one is individually managed by its own NOS. The combination makes it possible to implement, test, and even introduce new NOSs at runtime into existing networking infrastructures.

Like in computer virtualization where a hypervisor manages Virtual Machines (VMs) and their physical resource access [BDF+03], a so-called virtualization layer realizes the virtualization of SDNs [SGY+09]. The virtualization layer assigns, manages, and controls the physical network resources, while coordinating the access of virtual network tenants. The virtualization layer in SDN-based networks is realized by one or many network hypervisors [KAB+14]. They implement the control logic needed for virtualizing software-defined networks. They act as proxies between the NOSs of tenants and the shared physical infrastructure, where the vSDN networks reside. Due to their key position, a deep understanding of design choices and performance implications of network hypervisor implementations is of significant importance. Without this understanding, network operators cannot provide SPs with guaranteed and predictable network performance - a critical obstacle for the success of combining NV and SDN.

Beside implementation aspects of vSDNs, resource planning and management is another challenging task when combining NV and SDN. With SDN, the control logic can be flexibly distributed and placed among the network [HSM12]. Similar, network hypervisors implementing the logic for virtualization can also be distributed among the network. In order to provide low and predictable control plane latencies for tenants, the placement of network hypervisors is crucial for an efficient virtualization of SDN-based networks.

Moreover, it is expected that virtual network requests arrive over time with various characteristics, such as different topologies, diverse requirements for network resources like data rate or CPU, etc. The allocation of arriving requests, i.e., the embedding of virtual networks, needs to be solved fast and efficiently for an overall optimal network performance: e.g., fast and low-cost acquisition is a competitive advantage when virtualizing an infrastructure like in cloud environments [AFG+10; ZJX11]. Further, fast and efficient provisioning of virtual networks enhances the ability to react, which is essential, e.g., in case of Flash Crowds [EH08]. Hence, managing virtualization layers requires new mechanisms to improve the embedding quality of virtual networks and to speed up existing embedding algorithms.

Research on combining NV and SDN yields many open questions. Hence, the objectives of this doctoral thesis can be summarized as follows: as detailed performance models of network hypervisors are missing, the first objective is to deepen the understanding of existing network hypervisors. This includes the measurements of SDN-based virtualization architectures. Gaining deeper insights into the performance models of different architectures is a prerequisite for realizing predictable network performance. As the locations of control logics have an impact on network operations in SDN-based networks, the second goal is the optimization of network hypervisor placements. Static and dynamic virtual demand setups are comprehensively analyzed for a varying set of metrics tailored towards vSDNs such as latency reduction or reconfiguration minimization. For an efficient and fast embedding of virtual networks, the last objective targets improving optimization systems, e.g., to speed up embeddings, by applying methods from Machine Learning (ML).

1.1 Research Challenges

Modeling, analyzing, and optimizing virtualized software-defined networks comprises various challenges. This section summarizes the main research challenges targeted in the subsequent Chapters 3 to 5 in more detail.

Virtualization Layer Measurement Procedures

Many hypervisor architectures already exist in different flavors: from pure software-based to hardware-enhanced implementations, centralized and distributed ones, and designs tailored towards use cases such as mobile or data center networks. A general understanding of how to virtualize SDN networks is needed in order to design and conduct measurement studies. For this purpose, existing hypervisors must be analyzed and classified. The analysis should further investigate potential sources of processing overhead and network resource interference. While various hypervisor concepts already exist, a measurement schematic that can target all different hypervisor implementations has been missing so far in literature.

A general measurement schematic needs to consider many aspects. First, baseline measurements of non-virtualized SDN network performance must exist, i.e., the performance in a non-virtualized environment needs to be known. This holds for SDN controllers as well as SDN switches. Baseline measurements then allow to identify the overhead added by network hypervisors. Second, a measurement setup has to consider the aspects of virtualization, e.g., it needs to identify potential interference in case of multiple control connections between tenants and a network hypervisor - a main issue of vSDNs. This means in order to precisely identify and isolate root causes for overhead and interference, a full benchmark procedure must be able to emulate both parts, i.e., SDN switches and SDN controllers. Actually, a measurement tool for vSDNs has to orchestrate and manage multiple emulations of SDN controllers and switches simultaneously.

Virtualization Layer Design Towards Predictable Network Performance

Network hypervisor concepts exist that are designed to run on different execution platforms, in a centralized or distributed manner, or are designed for specific network types. All concepts have in common that they act on the network control traffic between tenants and the physical SDN hardware. While tenants mostly demand a strict resource isolation from their neighbors, resource sharing might lead to resource interference. Allocating resources without considering potential resource interference can lead to unpredictable resource delivery, i.e., virtual networks without effective resource guarantees for tenants. In particular in overload situations, hypervisors might significantly add latency on control plane operations due to their need to process the control plane traffic of the tenants. Hence, control plane isolation mechanisms that mitigate the overload and resource interference need to be designed and analyzed. These mechanisms should then be quantified by decent measurements.

Beside providing isolation and resource predictability, hypervisor architectures should be flexible and scalable. Flexibility in this context means to cope with network dynamics, such as virtual network demands changing over time, changing resource management objectives, or adaptations needed due to failures of the infrastructure. These network dynamics affect both the data plane and the control plane in vSDNs. For instance, locations of tenant controllers might also be adapted based on the time of the day. As a result, the virtualization layer needs to adapt. Network hypervisors might change their locations or switches might be reassigned to other hypervisor instances for the sake of resource efficiency. Accordingly, sophisticated reallocation mechanisms on the control plane of network hypervisors are needed. These reallocation mechanisms should then again be quantified in terms of overhead they add.

Modeling and Analyzing Placement Opportunities of Virtualization Layers

For distributed hypervisor architectures, the locations of the hypervisor instances should be optimally placed to reduce the virtualization overhead, e.g., additional control plane latencies. The optimization analysis is investigated in the context of the Network Hypervisor Placement Problem (HPP). Similar to the Controller Placement Problem (CPP), the HPP focuses on answering questions about how many hypervisor instances are needed and where they should be placed inside the network. Generally, both the CPP and the HPP are covering problems in the uncapacitated case. Accordingly, both are related, for instance, to uncapacitated facility location problems; facility location problems are shown to be NP-hard due to their reduction to set covering problems [KV07].

Similar to the initial controller placement studies, where control plane latencies were optimized, the initial hypervisor placement focuses on optimizing control plane latencies. In case of varying traffic demands, e.g., changing number of virtual networks over time, hypervisor locations might need to be adapted. A dynamic adaptation of the virtualization layer may introduce new sources of failures and service interruptions. Accordingly, reconfigurations should be avoided, if possible, as those might lead to high latencies or even network outages [GJN11; PZH+11]. Hence, there is a need for minimizing the amount of reconfiguration events when adapting the network hypervisor placement.

Fast and Efficient Virtual Network Provisioning

Allocation of virtual networks requires the placement (embedding) of virtual nodes and their interconnection via virtual links (paths). The mapping of physical to virtual resources has been extensively studied in the context of the Virtual Network Embedding (VNE) problem [FBB+13]. The challenge of the VNE problem is its NP-hard nature [ACK+16]. Accordingly, many heuristic VNE algorithms as well as optimal VNE algorithms targeting small networking setups have been proposed. As the network hypervisor might autonomously accept and reject virtual network demands, infrastructure providers would benefit from fast and efficient executions of existing VNE algorithms. Thus, a sophisticated hypervisor scheme should also involve new mechanisms such as from ML that optimize the overall embedding process, i.e., the efficiencies of VNE algorithms. An improved algorithm efficiency yields many advantages: it saves computational resources, it potentially leads to better embeddings of virtual networks, and it might speed-up the overall provisioning task.

1.2 Contributions

This section summarizes the contribution of this thesis to the research area of virtualizing software-defined networks. It overviews the content of the conducted studies and specifies their relations. Fig. 1.1 illustrates the structure of this thesis in the context of research areas and methodologies. Research has been conducted in three main areas: (1) measurements of SDN network hypervisor architectures and a system design mitigating identified shortcomings, (2) mathematical models and numerical analysis for the static and dynamic placement of SDN network hypervisors, (3) proposals of systems improving the efficiency of networking algorithms for provisioning virtual networks.

While the virtualization of software-defined networks provides advantages in terms of resource sharing and programmability, SDN network hypervisors introduce overhead. For example, they can cause higher control plane latencies due to additional processing, or even show unpredictable performance because of tenants interfering on the shared resources (Central Processing Unit (CPU), memory, network I/O) of the virtualization layer. A benchmark concept is thus proposed and used to measure the performance of SDN network hypervisors and their introduced overhead [BBK+17] in multi-tenant environments. In order to address identified resource interference problems of hypervisors, a system design for SDN-based networks towards adaptability [BBB+15] and predictability is presented and realized in a proof-of-concept implementation [BBR+16]. The newly designed system is analyzed with respect to interference mitigation and impact of reconfigurations due to adaptation. Measurements conducted in real SDN testbeds show that reconfigurations of the virtualization layer introduce additional control plane latency, which thus should be avoided by proper optimizations.

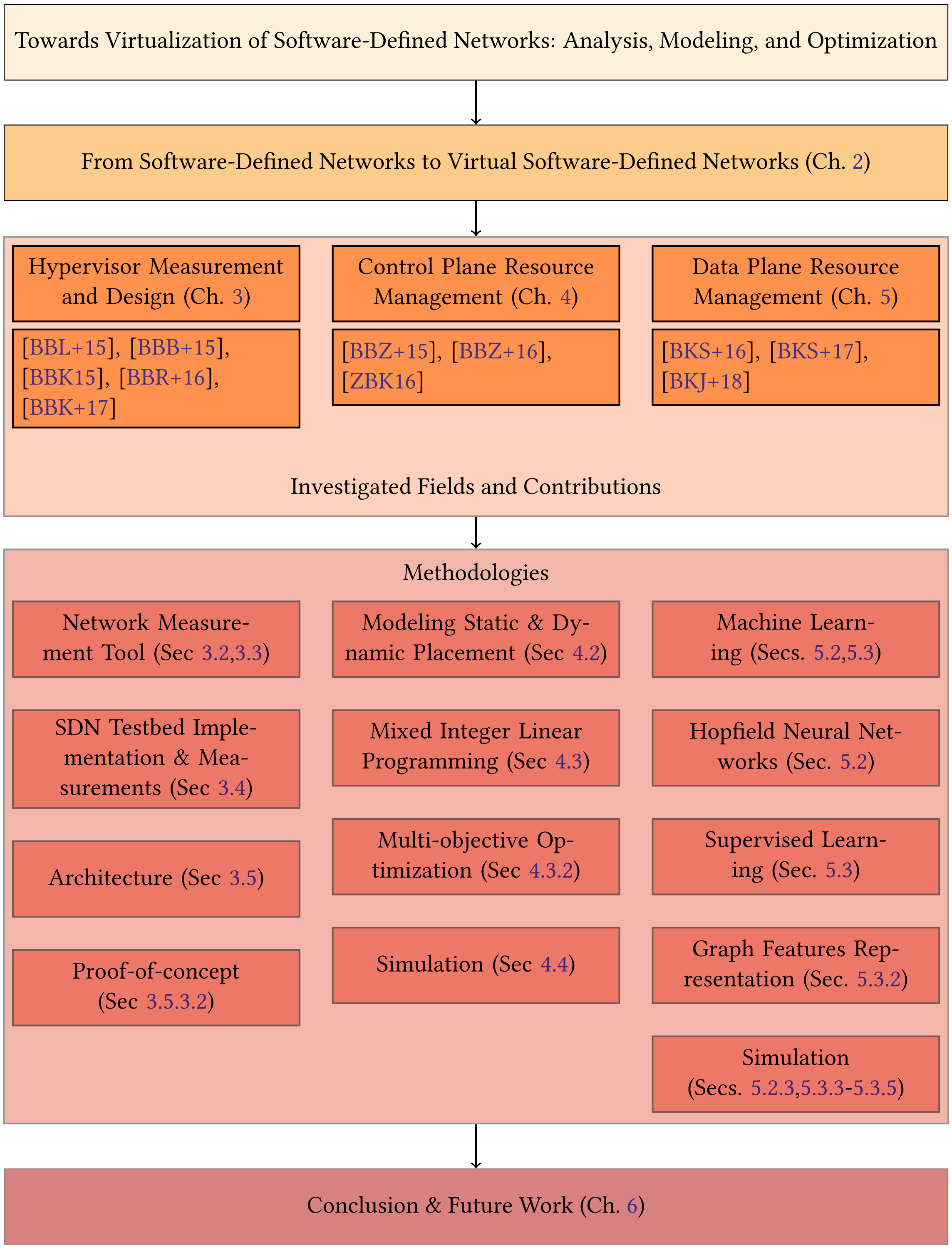

Towards Virtualization of Software-Defined Networks: Analysis, Modeling, and Optimization
From Software-Defined Networks to Virtual Software-Defined Networks (Ch. 2)
Hypervisor Measurement and Design (Ch. 3)
[BBL+15], [BBB+15], [BBK15], [BBR+16], [BBK+17]
Control Plane Resource Management (Ch. 4)
[BBZ+15], [BBZ+16], [ZBK16]
Data Plane Resource Management (Ch. 5)
[BKS+16], [BKS+17], [BKJ+18]
Investigated Fields and Contributions
Methodologies
Network Measurement Tool (Sec 3.2,3.3)
SDN Testbed Implementation & Measurements (Sec 3.4)
Architecture (Sec 3.5)
Proof-of-concept (Sec 3.5.3.2)
Modeling Static & Dynamic Placement (Sec 4.2)
Mixed Integer Linear Programming (Sec 4.3)
Multi-objective Optimization (Sec 4.3.2)
Simulation (Sec 4.4)
Machine Learning (Secs. 5.2,5.3)
Hopfield Neural Networks (Sec. 5.2)
Supervised Learning (Sec. 5.3)
Graph Features Representation (Sec. 5.3.2)
Simulation (Secs. 5.2.3,5.3.3-5.3.5)
Conclusion & Future Work (Ch. 6)

Figure 1.1: Thesis structure - the main investigated and contributed fields belong to three areas: hypervisor measurement and system design, control plane resource management, and data plane resource management. Different methodologies are applied in each area. Whereas the first content chapter focuses on practical methods, both resource management chapters focus on simulation-based studies of optimization problems related to combining SDN and NV.

The second major contributions target optimization models for analyzing the virtualization layer realized by SDN network hypervisors. Not only designing and dimensioning hypervisors, but also their placement inside the network has to be planned carefully; due to hypervisors' ability to be placed flexibly among a network, a thoroughly planned hypervisor placement might provide tremendous benefits, e.g, in terms of low control plane latencies. Correspondingly, we propose the Network Hypervisor Placement Problem (HPP) [BBZ+15] answering the questions of how many network hypervisor instances are needed and where they must be deployed in the network. The HPP is defined mathematically using Mixed Integer Linear Programming (MILP) according to the various architectural design possibilities of hypervisors. A simulation-based study compares the impact of hypervisor architectures. The study investigates how the number of hypervisors affects four control plane objectives [BBZ+16], where the metrics particularly account for virtualization. Furthermore, we analyze the overhead induced due to network virtualization. We propose new metrics to measure the control plane latency overhead when virtualizing SDN networks. In order to cope with the dynamic aspect of virtual networks, we model the Dynamic Hypervisor Placement Problem (DHPP), providing us a way to optimally solve this multi-objective problem via MILP [ZBK16]. An analysis of exact solutions is important for a careful planning of adaptive virtualization layers, which are inevitable to adapt to dynamically changing demands.

The third major contributions focus on the allocation of physical network resources towards virtual network demands. The problem of allocating virtual to physical network resources is well known in literature as the VNE problem. While many solutions exist that either solve the online or offline VNE problem, research has rarely focused on mechanisms that generally improve the optimization system performance. This includes methodologies from ML as well as their implementations and simulative analysis. We present system approaches applying ML in order to save computational resources, speed up the execution and even improving the outcome of networking algorithms. One system is *NeuroViNE* [BKJ+18], integrating Hopfield neural networks to improve solution qualities. *NeuroViNE* reduces the embedding cost by pruning the search space of embedding algorithms; as simulation results show, this can even improve solution qualities. Another suggestion is *o zapft'is* [BKS+17], a system that uses supervised learning to tap into the big data of networking algorithms: the problem-solution pairs of many solved instances. Knowledge extracted from the data of algorithms can be used in many flavors: to speed up the execution of networking algorithms, e.g., by predicting upper and lower bounds on solution values (either costs or benefits) or by predicting the feasibility of problem instances. Predicting feasibility of problems can avoid triggering algorithms solving hard or infeasible-to-solve problem instances [BKS+16], which can save system resources. Generally, we demonstrate that ML improves networking algorithm efficiencies.

1.3 Outline

The remainder of the thesis is structured as follows.

Chapter 2 gives background information on Network Virtualization (NV) and Software-Defined Networking (SDN), and puts them in relation with each other. It describes the relation between virtual networks, software-defined networks and virtual software-defined networks. Moreover, it provides a brief summary of SDN network hypervisors.

Chapter 3 addresses measurement methodologies for and the various designs of the network virtualization layer, i.e., SDN network hypervisors. Identified as a research gap, it introduces a benchmark system for virtual SDN environments and provides measurement results of existing virtualization layer implementations. It also outlines HyperFlex, a flexible SDN virtualization layer design towards improved predictability and adaptability.

Chapter 4 initiates the study of optimization problems which address the control plane of SDN virtualization layers, i.e., it introduces, models, and analyzes the hypervisor placement problem. In order to analyze the hypervisor placement problem, it introduces MILP-based mathematical models for virtualization layers targeting static and dynamic traffic scenarios. It then analyses the various models for different objectives, network topologies and varying input sets of virtual network requests.

Chapter 5 is mainly concerned with the optimization of the allocation of data plane resources to physical networks. As a new angle to improve costly executions of optimization algorithms, methods based on neural computation and ML are proposed. The methods reduce the search space or predict the feasibility and objective values before actually executing optimization algorithms.

Chapter 6 concludes this thesis and provides thoughts on future work.

Chapter 2

Combining Network Virtualization and Software-Defined Networking

In this chapter, we first elaborate on the general background that is related to this thesis, i.e., the concepts of Network Virtualization (NV) and Software-Defined Networking (SDN) (Sec. 2.1). Afterwards, Sec. 2.2 outlines the differences between software-defined networks and virtual software-defined networks. Sec. 2.3 introduces the SDN network hypervisor and its many tasks; the network hypervisor is the main entity capable of virtualizing SDN networks. The content of this chapter relies partly on the survey [BBR+16].

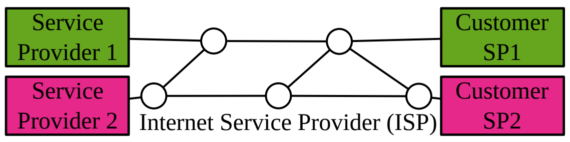
2.1 Paradigms and Definitions

This section introduces the relevant paradigms and definitions of NV and SDN.

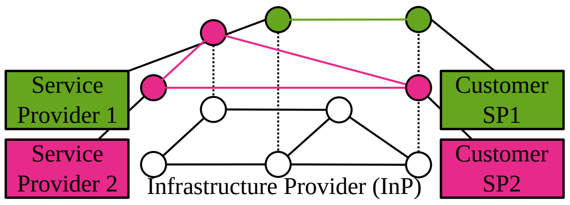
2.1.1 Network Virtualization (NV)

Virtualization of computers has been the main driver for the deployment of data centers and clouds [Gol74; LLJ10; SML10; DK13; SN05; ZLW+14]. Inspired by this successful development, Network Virtualization (NV) has initially been investigated for testbed deployments [APS+05; TT05; FGR07]. The idea of sharing physical networking resources among multiple tenants or customers has then been transferred to communication networks serving production network traffic: NV is seen as the key enabler for overcoming the ossification of the Internet [APS+05; FGR07; TT05]. As the idea of virtual networks is not new in general (e.g., Virtual Local Area Network (VLAN) defines layer 2 virtual networks), different network virtualization definitions and models have been proposed [CB08; BBE+13; CKR+10; KCI+15; KAB+14], e.g., based on the domain (data centers, wide area networks) they target.

NV has led to new business models, which are seen as main drivers for innovation for communication network technologies. In this thesis, we apply the business model for network virtualization roles as introduced in [CB09]. Fig. 2.1 compares the traditional roles with the NV roles. Traditionally, an Internet Service Provider (ISP) provides Internet access, i.e., connectivity, for its customers towards a Service Provider (SP), such as Google, Netflix, Amazon, etc., which host their services in data centers (Fig. 2.1a). In the business models of network virtualization, as illustrated in Fig. 2.1b,



(a) Traditional role: Internet Service Provider (ISP) providing access to service providers and customers. SPs cannot impact network decision and resource allocation.



(b) NV business model: SPs request virtual networks. Customers connect to services via virtual networks.

Figure 2.1: Traditional business model with ISP versus NV business model with InP.

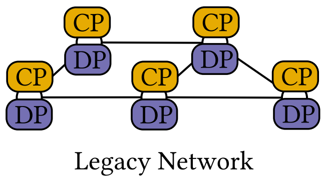
the traditional role of an ISP of managing and operating networks is split into an SP role and an InP role [FGR07]. The SP's role can be enriched with network control. They become the operators of virtual networks. Thus, SPs can use their knowledge about their services and applications to implement advanced network control algorithms, which are designed to meet the service and application requirements. It is then the task of the InP to provide virtual networks to the SPs. While SPs (tenants) might create virtual networks by requesting resources from multiple InPs, use cases in this thesis assume the existence of only one single InP.

Generally, virtualization is construed differently among networking domains [CB09]. Hence, different networking domains use varying technologies to realize virtual networks. For instance, VXLAN [MDD+14], GRE [FLH+00], or GRE's NV variant NVGRE [GW15] are used in data centers to interconnect virtual machines of tenants. Techniques such as Multiprotocol Label Switching (MPLS) [XHB+00; RVC01] create logically-isolated virtual networks on the Internet Protocol (IP) layer based on tunneling technologies, while potentially relying on specialized networking hardware.

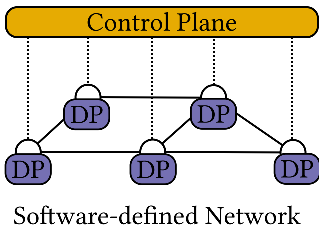
Full network virtualization comprises all physical resources that are needed to provide virtual networks with guaranteed and predictable performance (in this thesis: CPUs of network nodes, data rate and latency of network links.). The ability to program virtual networks, e.g., by using SDN, is a further important key aspect of (full) network virtualization [KAB+14]. Taking a look at VLAN-based virtualization without the ability of programming, tenants have no opportunity to instruct switches to make traffic steering decisions. However, to fully benefit from NV opportunities, tenants should obtain virtual network resources, including full views of network topologies and allocated networking resources, involving link data rates and network node resources, such as CPU or memory.

Providing isolated and programmable virtual networks has manifold advantages: first, network operators can design, develop, and test novel networking paradigms, without any constraints imposed by the currently deployed protocols or (Internet) architecture [APS+05]. Second, network systems that are designed to the demands of the served applications or users do not suffer from the overhead of unused network stacks or protocols. Furthermore, NV is seen as a key to provide predictable (guaranteed) network performance [BCK+11]. As a consequence, SPs should be enabled to offer new services over existing infrastructures much faster with higher flexibility, i.e., ease to adapt their networks to changing user and service demands [KAG+12].

Overall, this thesis targets NV architectures that aim at the following three functions: (1) the



(a) Legacy network where the control plane (CP) and the data plane (DP) are integrated into a device.



(b) Software-defined network where the control plane (CP) is decoupled from the data plane (DP) of the devices.

Figure 2.2: Comparison of legacy and software-defined network.

function to abstract the underlying infrastructure, thus to create virtual networks; (2) the function to program the virtual networks independently; (3) algorithms and functions guaranteeing isolated virtual networks.

2.1.2 Software-Defined Networking (SDN)

In legacy networks, the control plane is tightly coupled into the same device as the data plane. Fig. 2.2a shows an example network where the control plane is distributed among the devices. The control plane is responsible for control decisions, e.g., to populate the routing tables of IP routers for effective packet forwarding. Accordingly, in case of distributed control planes, an overall network control is established through the operation of distributed network operating systems. As distributed operating systems may belong to different stakeholders, a common agreement on the available functions and protocols is always needed in case of adaptations. This, however, may hinder the innovation of communication networks.

SDN decouples the control plane from the data plane, which allows a centralized logical control of distributed devices [MAB+08]. Fig 2.2b illustrates an example where the control is decoupled from the devices. The control plane logic is centralized in the SDN controller, which operates the SDN switches. The centralized control maintains the global network state, which is distributed across the data plane devices. While the data plane devices still carry out the forwarding of data packets, the centralized control now instructs the data plane devices how and where to forward data packets.

SDN controllers are written in software, which can be implemented through a variety of programming languages, e.g., C++ or Python. This hardware independence is expected to bring faster development and deployment of networking solutions. One of the first SDN controllers has been NOX [GKP+08]. Many SDN controllers have followed: e.g., Ryu [Ryu], ONOS [BGH+14], Beacon [Eri13], OpenDayLight [Ope13]. Being implemented in software, SDN further allows to freely design control plane architectures among the whole spectrum from one central entity to a completely distributed control plane, such as in traditional networks [TG10; KCG+10]. Accordingly, Tootoonchian et al. [TG10] distribute the SDN control plane for scalability and reliability reasons.

SDN defines the Data-Controller Plane Interface (D-CPI)[1] to program distributed networking devices. The D-CPI is used between the physical data plane and the (logically centralized) control plane.

[1] This interface has also been known as the *Southbound* interface earlier.

The connection between SDN devices and SDN controllers is referred as *control channel* in this thesis. To establish connections through control channels, SDN switches still host agents that receive and execute commands from the external control plane. To provide a common operation and control among heterogeneous SDN devices, instruction sets that abstract the physical data plane hardware are needed. The most most popular development is the OpenFlow (OF) protocol [MAB+08].

SDN applies a match and action paradigm to realize packet forwarding decisions. The SDN controller pushes instruction sets to the data plane, which include a match and action specification. In SDN, match specifications define a flow of packets, i.e., network traffic flow. A match is determined by the header values of packets of a network flow. For example, OF defines a set of header fields including, e.g., the Transmission Control Protocol (TCP) header and the IP header. An SDN controller instructs SDN switches to match on the specified fields and apply actions. The main actions are forward, drop, or modify network packets. Each version of the OF specification extended the set of header fields that can be matched on as well as the available actions.

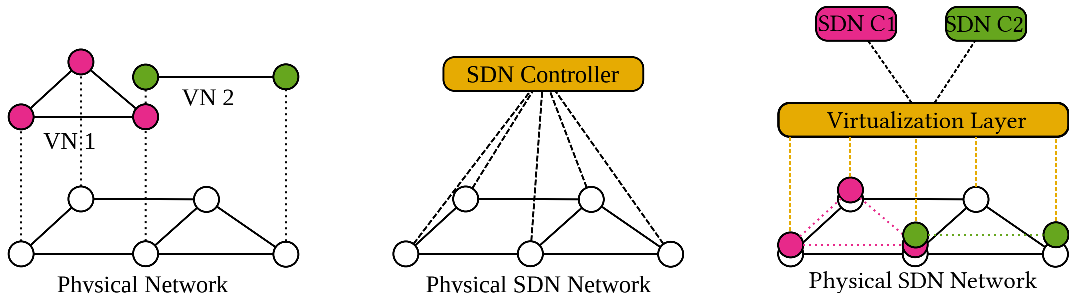
An SDN switch stores the commands, i.e., instructions, how to operate networks in one or multiple flow tables. The size for storing flow table entries is a defining characteristic of an SDN switch. Flow tables can be implemented either in hardware or software. Current OF switches use Ternary Content Addressable Memories (TCAMs) for hardware tables, which are fast but costly and limited. In contrast, software tables provide more space but are slower than hardware tables in storing and complex matching of network flows [KPK15]. Accordingly, for a predictable isolation, such resources also need to be allocated properly.

In order to ease the development of new networking applications as well as the control of software-defined networks, controllers provide Application-Controller Plane Interfaces (A-CPIs) [Ope14b; Ope14c]. Using A-CPIs, networking applications, like firewalls or load balancers, reside in the application control plane. Networking applications can be developed upon the provided functionality of the controllers' specific A-CPIs. Accordingly, while networking application designers and operators can again freely develop in any programming language, they are dependent on the A-CPI protocol of the respective controller, e.g., the REST API of ONOS [BGH+14] in OF-based networks. However, no common instruction set for the A-CPI has been defined yet.

2.2 From Software-Defined Networks to Virtual Software-Defined Networks

Combining NV and SDN provides tenants the advantages of both concepts, i.e., flexible resource sharing by acquiring virtual networks, and programmability of the network resources by using SDN. With the introduced programmability of SDN, NV offering virtual resource programmability towards tenants can now be put into effect [JP13]. Accordingly, NV is seen as one killer application of SDN [DKR13; FRZ14], that is, it provides the programmability of virtual network resources. The result of combining NV and SDN are vSDNs sharing the same infrastructure.

Figure 2.3 illustrates the differences between virtual networks, software-defined networks, and virtual software-defined networks. Figure 2.3a shows the traditional view of virtual networks. Two



(a) Virtual Networks (b) SDN Network (c) Virtualized SDN Network

Figure 2.3: Comparison of virtual networks, SDN network, and virtual SDN networks. Dotted lines on Fig. 2.3a indicate the embedding of the virtual nodes. Dashed lines in Fig. 2.3b illustrate the control connections between the SDN controller and the physical SDN network. Black dashed lines in Fig 2.3c show the connection between tenant controllers and the virtualization layer, while dashed colored lines show the connections between the virtualization layer and the physical SDN network. Dotted lines between virtual nodes on the physical network indicate the virtual paths between them.

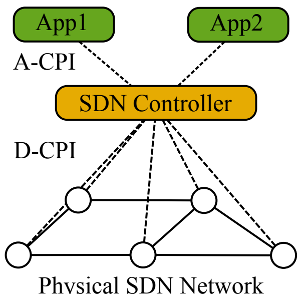
virtual networks are hosted on a substrate network. The dashed lines illustrate the location of the virtual resources. The interconnection of the virtual nodes is determined by the path embedding or routing concept of the InP. A clear way how a tenant can control and configure its virtual network is not given. SDN provides one option for providing virtual network resource control and even configuration to tenants. Figure 2.3b illustrates how an SDN controller operates on to top of a physical SDN network. The SDN controller is located outside of the network elements. It controls the network based on a logically centralized view. Figure 2.3c shows the combination of NV and SDN. A virtualization layer is responsible for managing the physical network. Besides, the virtualization layer orchestrates the control access among SDN controllers (here SDN C1 and C2) of tenants. As an example, tenant 1 (VN 1) has access to three network elements while tenant 2 (VN 2) has access to two network elements. Note the virtualization layer in the middle that is shared by both tenants.

2.3 SDN Network Hypervisors - An Introduction

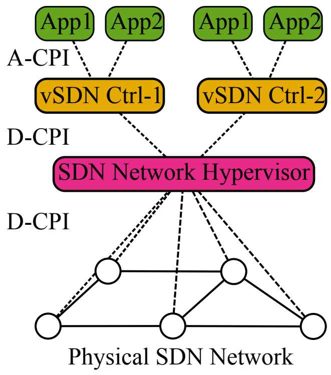
SDN network hypervisors implement the virtualization layer for virtualizing SDN networks. They provide the main network functions for virtualization of SDN networks. In this section, we explain how to virtualize SDN networks through a network hypervisor and its virtualization functions. We highlight the main functions that need to be implemented towards being compliant with the introduced abstraction demands of NV.

2.3.1 SDN Controllers versus SDN Network Hypervisors

By adding a virtualization layer, i.e., a network hypervisor, on top of the networking hardware, multiple vSDN operating systems are alleviated to control resources of the same substrate network. This concept has been proposed by [SGY+09; SNS+10]. The network hypervisor interacts with the networking hardware via the D-CPI through an SDN protocol, e.g., OF. In case of NV, the hypervisor provides on top the same D-CPI interface towards virtual network operators, i.e., SDN network tenants. This feature of the hypervisor, i.e., to interface through multiple D-CPI with multiple virtual



(a) Applications interact via A-CPI with SDN Controller. SDN controller interacts via D-CPI with physical SDN network.



(b) Comparison of SDN and vSDNs, and respective interfaces. The tenant controllers communicate through the SDN network hypervisor with their virtual switches.

Figure 2.4: Comparison of SDN and Virtual Software-Defined Network (vSDN), and respective interfaces.

SDN controllers, is seen as one of the defining features when virtualizing SDN networks.

Fig. 2.4 illustrates the difference between SDN networks and vSDNs in terms of their interfaces to tenants. In SDN networks (Fig. 2.4a) network applications are running on top of an SDN controller. The applications use the A-CPI of the controller to connect and communicate with the SDN controller.

For vSDNs, as depicted in Fig. 2.4b, the network hypervisor adds an additional layer of abstraction, the virtualization layer. The tenants now communicate again via a D-CPI with the network hypervisor. Still, on top of the tenant controllers, applications communicate via the controllers' A-CPI interfaces during runtime. The hypervisor acts as a proxy: it intercepts the control messages between tenants and the physical SDN network. The hypervisor acts as the SDN controller towards the physical SDN network. It translates the control plane messages between the tenant SDN controllers and the physical SDN network. Message translation is the main functional task a hypervisor has to accomplish.

Remark. In contrast to vSDN, some controllers such as ONOS [BGH+14] or OpenDayLight (ODL) [Ope13], might also provide NV in the sense that applications on top are operating isolated virtual network resources. However, the virtual network operators are bounded to the capabilities of the controllers' A-CPIs, e.g., the REST-API of ONOS [BGH+14]. Accordingly, tenants cannot bring their own SDN controllers, i.e., their fully fledged and individually adapted NOS. Thus, we do omit controllers such as ODL or ONOS for our analysis.

2.3.2 SDN Network Hypervisors: Virtualization Tasks and Functions

Next to translating messages, SDN network hypervisors face many tasks when virtualizing SDN networks: they need to grant access to tenants, isolate the virtual networks on the data plane, avoid interference on the control plane, guarantee predictable network operation, grant adaptation capabilities etc. In this thesis, we outline the tasks of a network hypervisor to abstract (virtualize) SDN networking resources and to create isolated virtual SDN networks.

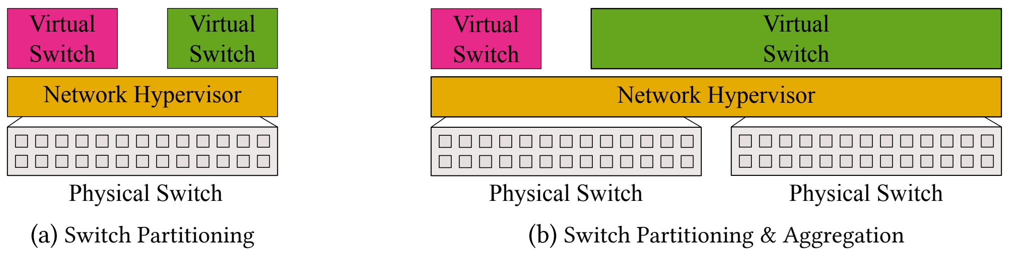


Figure 2.5: Comparison between switch partitioning and switch partitioning & aggregation. Fig. 2.5a shows a physical switch partitioned into two virtual switches. Fig. 2.5b illustrates partitioning and aggregation. Here, the right virtual switch represents an aggregation among two physical switches.

2.3.2.1 Abstraction

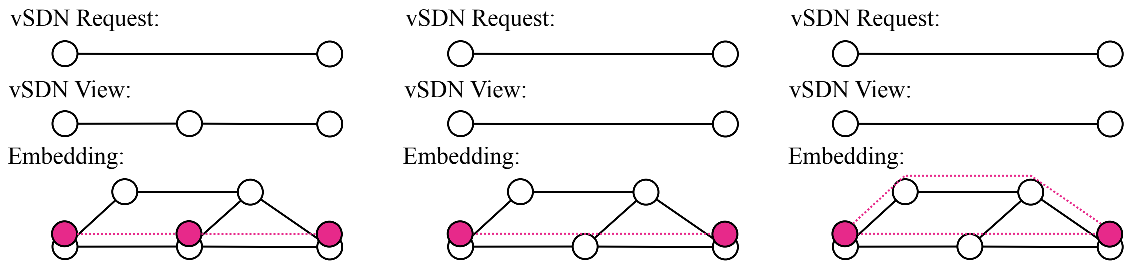
Overall, many tasks are affected by the realization of the main feature that hypervisors need to offer: abstraction. Abstraction means "the act of considering something as a general quality or characteristic, apart from concrete realities, specific objects, or actual instances" [Cor02]. As abstraction is seen as a fundamental advantage of NV and SDN [CFG14; CKR+10; DK13], an SDN network hypervisor should be able to abstract details of the physical SDN network. The degree of abstraction of the network representation determines also the *level of virtualization* [CB08], which is provided by a network hypervisor. The available features and capabilities are directly communicated by a hypervisor towards the tenants.

Three SDN network abstraction features are seen as the basic building blocks of a virtualization layer for SDN: topology abstraction, physical node resource abstraction, and physical link resource abstraction.

Topology Abstraction

Topology abstraction involves the abstraction of topology information, i.e., the information about the physical nodes and links that tenants receive as their view of the topology. The actual view of tenants is defined by the mapping of the requested nodes/links to the physical network and the abstraction level provided by the virtualization layer. Generally, we define the mapping of a virtual node/link to many physical nodes/links as a "1-to-N" mapping. A virtual node, for instance, can span across many physical nodes. In case a tenant receives a "1-to-N" mapping without abstraction, he has to do additional work; the tenant has to implement the forwarding of network packets on intermediate nodes by himself. When regarding links, while a tenant requests only a virtual link between two nodes, he receives a view also containing intermediate nodes to be managed by the tenant. In case nodes and links are mapped to only one physical instance, we call this a "1-to-1" mapping.

The provided information about nodes involves their locations and their interconnections through links. A virtual node can be realized on one ("1-to-1") or across many physical nodes ("1-to-N"). Fig. 2.5 illustrates the two cases. Fig. 2.5a shows an example where a switch is partitioned into multiple virtual instances. Each virtual switch is running on one physical switch only. As an example for node aggregation, i.e., where two physical instances are abstracted as one, a tenant operating a secure SDN network wants to operate incoming and outgoing nodes of a topology only. Thus, a physical topology consisting of many nodes might be represented via "one big switch" [MRF+13;



(a) View abstraction with a 1-to-N link mapping and 1-to-1 node mapping, which involves the intermediate node. No path abstraction and no path splitting.

(b) View abstraction with 1-to-N link mapping and 1-to-1 node mapping, which does not involve the intermediate node. Path abstraction and no path splitting.

(c) Link abstraction with 1-to-N link mapping and 1-to-1 node mapping, which does not involve the intermediate nodes. Path abstraction and path splitting.

Figure 2.6: Comparison between link abstraction procedures. On top the requested virtual network. In the middle the provided view based on the embedding on the bottom.

CB08; CKR+10; JGR+15]. As illustrated in Fig. 2.5b, the right green switch is spanning two physical instances. The network hypervisor aggregates the information of both physical switches into one virtual switch.

Similar, different mapping and abstraction options for virtual links or paths exist. Physical links and paths are abstracted as virtual links. Realizations of virtual links can consist of multiple hops, i.e., physical nodes. A tenant's view might contain the intermediate nodes or not. An example where intermediate nodes are not hidden from the view is shown by Fig. 2.6a. The request involves two nodes and a virtual path connecting them. As the physical path realization of the virtual link spans an intermediate node, the view depicts this node. Fig. 2.6b shows a view that hides the intermediate node. Besides, a virtual path can also be realized via multiple physical paths, which is illustrated by Fig. 2.6c. For this, the infrastructure needs to provide path splitting techniques [YRF+10].

Physical Node Resource Abstraction

In virtual networks, CPU and memory are usually considered as the main network node resources. For SDN networks, memory is additionally differentiated from the space to store matching entries for network flows, i.e., flow table space. While flow table space is only used for realizing the match-and-action paradigm, memory might be needed to realize a network stack (e.g., a tenant NOS on a device in case of a distributed virtualization architecture) or for realizing network management functionality (e.g., a byte counter for charging). CPU resource information can be abstracted in different ways. It can be represented by the number of available CPU cores or the amount of percentage of CPU. Similar, tenants can receive a concrete number or partitions of memories. Flow table resources involve, e.g., the number of flow tables or the number of TCAMs [PS06; PS02]. For instance, if switches provide multiple table types such as physical and software tables, network hypervisors need to reserve parts of these tables according to the demanded performance. Again, based on the level of abstraction, the different table types, i.e., software or hardware, are abstracted from tenants' views in order to lower operational complexity for them.

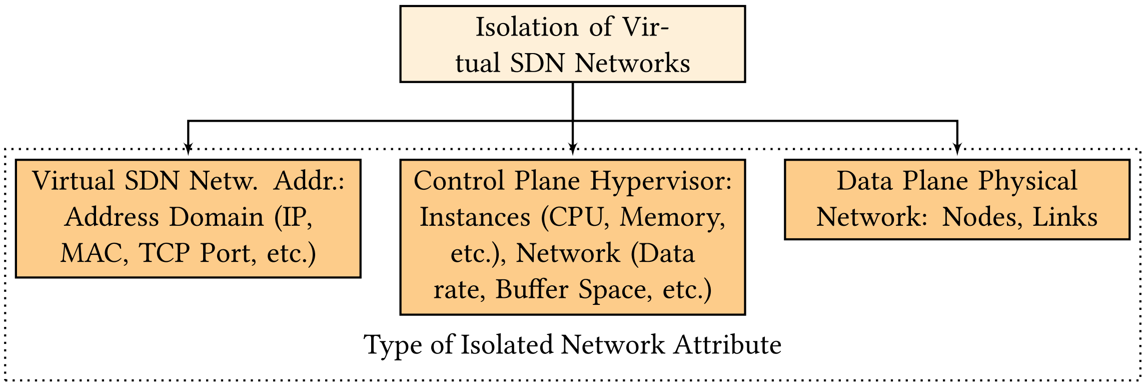

Isolation of Virtual SDN Networks
Virtual SDN Netw. Addr.: Address Domain (IP, MAC, TCP Port, etc.)
Control Plane Hypervisor: Instances (CPU, Memory, etc.), Network (Data rate, Buffer Space, etc.)
Data Plane Physical Network: Nodes, Links
Type of Isolated Network Attribute

Figure 2.7: Network hypervisors isolate three network virtualization attributes: control plane, data plane, and vSDN addressing. In particular the first and second attribute are addressed in this thesis.

Physical Link Resource Abstraction

Physical link resources involve the data rate, the available queues, the queue attributes such as different priorities in case of a priority queuing discipline, as well as the link buffers. Tenants might request virtual networks with delay or loss guarantees. To guarantee delay and upper loss bounds, substrate operators need to operate queues and buffers for the tenants. That is, the operation of queues and buffers is abstracted from the tenant. However, tenants might even request to operate queues and buffers themselves. For instance, the operation of meters, which rely on buffers, is a fundamental feature of recent OF versions. To provide tenants with their requested modes of operations, like metering, substrate operators need to carefully manage the physical resources.

2.3.2.2 Isolation

Network hypervisors should provide isolated virtual networks for tenants while trying to perceive the best possible resource efficiency out of the physical infrastructure. While tenants do not only demand physical resources for their operations, physical resources are also needed to put virtualization into effect, e.g., to realize isolation. Physical resources can be classified into three main categories as depicted in Fig 2.7: the provided addressing space, the control plane and the data plane resources.

vSDN Addressing Isolation

With virtualized SDN networks, tenants should receive the whole programmability of an SDN network. This means, tenants should be free to address flows according to their configurations and demands. Accordingly, techniques need to provide unique identification of the flows of different tenants. In a non-virtualized SDN network, the amount of addressable flow space is limited by the physical infrastructure attributes, i.e., the type of network (e.g., layer 2 only) and the used protocols (e.g., MPLS as the only tunneling protocol). The available headers and their configuration possibilities determine the amount of available flows. In virtualized SDN networks, more possibilities are available: e.g., if the vSDN topologies of two tenants do not overlap physically, the same address space in both vSDNs would be available. Or if tenants request lower OF versions than the deployed ones, extra header fields added by higher OF versions could also be used for differentiation. For instance, OF 1.1 introduced MPLS that can be used to distinguish tenants who request only OF 1.0.

Control Plane Isolation

In SDN networks, the control plane performance affects the data plane performance [TG10]. On the one hand, execution platforms (i.e., their CPU, memory, etc.), which host SDN controllers, directly influence the control plane performance [KPK14; RSU+12; TG10; TGG+12]. For instance, when an SDN controller is under heavy load, i.e., available CPUs run at high utilization, OF control packet processing might take longer. As a result, forwarding setups on switches might be delayed. On the other hand, also the resources of the control channels and switches can impact the data plane performance. In traditional routers, the routing processor running the control plane logic communicates with the data plane elements (e.g., Forwarding Information Base (FIB)) over a separate bus (e.g., a PCI bus). In SDN, controllers are running as external entities with their control channels at the mercy of the network. If a control channel is currently utilized by many OF packets, delay and even loss might occur, which can lead to delayed forwarding decisions. On switches, so called agents manage the connection towards the external control plane. Thus, switch resources consumed by agents (node CPU & memory, and link buffer & data rate) can also impact the performance of the control plane.

The overall performance perceived by tenant controllers is determined by many factors: the physical network, i.e., node and link capabilities; the processing speed of a hypervisor being determined by the CPU of its host; the control plane latency between tenant controllers and hypervisors is determined by the available data rate of the connecting network. What comes in addition is the potential drawback of sharing resources: performance degradation due to resource interference. Resource interference happens, for instance, when misconfigured controllers overwhelm virtualization infrastructures with too many control messages. Without isolation, the overload generated by a tenant can then degrade the perceived performance of other tenants, which degrades the data plane performance. To provide predictable and guaranteed performance, a resource isolation scheme for tenants needs to carefully allocate all involved resources at the data and the control plane.

Data Plane Isolation

The main resources of the data plane are node CPUs, node hardware accelerators, node flow table space, link buffers, link queues, and link data rates. Node resources need to be reserved and isolated between tenants for efficient forwarding and processing of the tenants' data plane traffic. On switches, for instance, different sources can utilize their CPU resources: (1) generation of SDN messages, (2) processing data plane packets on the switches' CPUs, i.e., their "slow paths", and (3) switch state monitoring and storing [SGY+09]. As there is an overhead of control plane processing due to involved networking operations, i.e., exchanging control messages with the network hypervisor, virtualizing SDN networks requires an even more thorough allocation of all involved resources. Besides, the utilization of the resources might change under varying workloads. Such variations need also be taken into account when allocating resources.

In order to successfully accomplish all these tasks, many challenges need to be solved in different research areas: e.g., architecture design of hypervisors providing predictable and guaranteed performance as well as solving algorithmically hard resource allocation problems for provisioning isolated virtual network resources.

Chapter 3

Measurements and Design for Virtual Software-Defined Networks

Whereas SDN itself introduces already noticeable changes on the data plane as well as the control plane, a deep understanding of the impact of all facets of NV in combination with SDN is missing. Many research work has been conducted to understand how to exploit the flexible resource allocation and to isolate traffic of virtual networks on the data plane. However, less attention has been given to the entity virtualizing the control plane, namely the network hypervisor, which is de-facto the most critical component in a virtual and programmable network environment. Accordingly, we in this chapter analyze and measure in detail existing virtualization layer (hypervisor) architectures.

Infrastructure providers need technologies for virtualization that provide a highly predictable network operation. Otherwise, it would be infeasible for infrastructure providers to negotiate Service Level Agreements (SLAs) with virtual network tenants. Precise and guaranteed SLAs are indispensable for tenants: they may need to operate time-critical applications on their virtual networks or they want to guarantee application execution times for jobs of big data tasks. Based on our measurement observations, we further propose a virtualization layer architecture targeting a more predictable operation even in dynamic environments.

Content and outline of this chapter. Section 3.1 first provides background on OF and existing SDN benchmarking tools. Second, it introduces related work with respect to the challenges of providing predictable and guaranteed performance in virtualized environments, i.e., shared servers and cloud infrastructures. The measurement procedure for virtual SDN environments, as introduced in Section 3.2, is mainly taken from [BBR+16]. The implementation of a benchmark tool based on the proposed measurement methodology is outlined in Section 3.3. Using the new tool, the conducted measurement studies, shown in Section 3.4, are mainly taken from [BBK+17]; in contrast to [BBK+17], the measurement data are reinterpreted from the predictability point of view. Section 3.5 introduces HyperFlex, a virtualization layer concept towards better network operation predictability. HyperFlex was originally proposed in [BBK15], while the control plane reconfiguration procedure measurement was contributed in [BBB+15]. The original measurement studies are significantly extended in this thesis to generally proof the concept and to gain much deeper insights into the operations of hypervisors.

3.1 Background and Related Work

This thesis targets network hypervisors for OpenFlow (OF)-based SDN networks. Hence, this section introduces background information on the OF protocol, its implementation aspects and message types, which is defined by the Open Networking Foundation (ONF) in different versions [Ope09; Ope11a; Ope11b; Ope12; Ope13; Ope14a]. Afterwards, it highlights details on the state-of-the-art of measuring SDN networks. Moreover, it overviews SDN network hypervisors. The related work focuses on measurements in SDN-based networks and on resource interference in shared environments, e.g., computers, networks, and clouds.

3.1.1 OpenFlow Protocol

As OF is so far the most prominent and accepted realization for SDN-based networks, many OF controllers, switches, benchmark tools and network hypervisors exist. Accordingly, when discussing the performance of OF-based vSDNs, background information on the OF protocol and existing benchmark tools should be provided for a basic understanding of OF-based network hypervisors.

OpenFlow components. In an SDN network, one *controller* manages multiple *OF switches*: controllers and switches build the end-points of the OF protocol. The switches are connected via multiple *OpenFlow (OF) control channels* with the controller.

An OF switch typically has one control channel for one controller; auxiliary (parallel) control channels are possible, e.g., to improve redundancy. The OF specification does not specify the control channel to be an out-band network, i.e., dedicated switches for the control traffic, or an in-band one, where control traffic is transmitted through the managed OF switches.

Literature sometimes calls the entity that implements the OF specification on the switch side the *OpenFlow (OF) agent* [KRV+15]. Given the IP address of the controller, OF agents initiate TCP connections to the controller via the OF control channel interface. The controller's IP address is normally pre-configured on switches before starting network control and operation. OF messages, i.e., commands, statistics, and notifications are sent through the control channels.

OpenFlow messages. OF defines three message types: *Controller-to-Switch*, *Asynchronous*, and *Symmetric*. The controller initiates *Controller-to-Switch* messages: e.g., to request features or to send a packet out on the data path of the switch. Only switches send asynchronous messages. Switches use these messages to report network events to controllers or changes of the their states. Both controllers or switches can send *Symmetric* messages. They are sent without solicitation. The following message types are used for measurements. Accordingly, they are briefly discussed in the following. *Asynchronous* messages:

- OFPT_PACKET_IN: Switches send OFPT_PACKET_IN messages to controllers to transfer the control of the packet. They are either triggered by a flow entry (a rule that specifies to send OFPT_PACKET_IN messages) or by a table-miss (when no rule can be found and the switch is then sending OFPT_PACKET_IN messages as its default behavior).

Controller-to-Switch messages:

- OFPT_FEATURES_REQUEST and OFPT_FEATURES_REPLY: This message request/reply pattern exchanges the main information on switch identities and on switch capabilities. A controller normally requests features once when a new control channel connection is established.
- OFPT_FLOW_MOD: This message modifies flow table entries; it adds, modifies, or removes flows from tables.
- OFMP_PORT_STATS: This is another message type that demands a switch to send a reply. The controller sends this message to request statistics about one or many ports of the switch. Statistics can be about received, transmitted packets and bytes etc.
- OFPT_PACKET_OUT: A controller uses this message type to send a packet out through the datapath of a switch. The controller sends this message, for instance, to discover topologies by using the Link Layer Discovery Protocol (LLDP).

3.1.2 SDN Network Hypervisors

In this section, we will briefly outline existing SDN network hypervisors. A comprehensive survey of existing SDN network hypervisors is given in [BBR+16].

3.1.2.1 FlowVisor (FV)

FlowVisor (FV) [SGY+09] has been the first hypervisor for virtualizing OF-based software-defined networks, enabling sharing of SDN networking resources between multiple SDN controllers.

- *Architecture.* FV is a software network hypervisor and can run stand-alone on any commodity server or inside a virtual machine. Sitting between the tenant SDN controllers and the SDN networking hardware, FV processes the control traffic between tenants from and to the SDN switches. FV further controls the view of the network towards the tenants, i.e., it can abstract switch resources. FV supports OF 1.0 [Ope09].
- *Flowspace.* FV defines the term *flowspace*. A flowspace for a tenant describes a possibly non-contiguous sub-space of the header field space of an OF-based network. Flowspaces between tenants should not overlap; therefore, FV guarantees isolated flowspaces for tenants. If tenants try to address flows outside their flowspaces, FV rewrites the packet headers. If packet headers cannot be rewritten, FV sends an OF error message. FV distinguishes shared from non-shared switches between tenants for flowspace isolation. For shared switches, FV ensures that tenants cannot share flowspaces, i.e., packet headers. For non-shared switches, flowspaces can be reused among tenants as those are physically isolated.
- *Bandwidth Isolation.* While OF in its original version has not provided any QoS techniques for data plane isolation, FV realized data plane isolation by using VLAN priority bits in data

packets. Switches are configured out-of-band by a network administrator to make use, if available, of priority queues. FV, then, rewrites tenant rules to further set the VLAN priorities of the tenants' data packets [Sof09]. The so called VLAN Priority Code Point (PCP) specifies the 3-bit VLAN PCP field for mapping to eight distinct priorities. As a result, data packets can be mapped to the configured queues; hence, they receive different priorities.

- *Topology Isolation.* FV isolates the topology in a way that tenants only see the ports and switches that are part of their slices. For this, FV edits and forwards OF messages related to a specific slice per tenant only.
- *Switch CPU Isolation.* In contrast to legacy switches, where control does not need an outside connection, OF switches can be overloaded by the amount of OF messages they need to process. The reason is that the processing needed for external connections adds overhead on switch CPUs. In detail, OF agents need to encapsulate and decapsulate control messages from and to TCP packets. Thus, in case a switch has to process too many OF messages, its CPU might become overloaded. In order to ensure that switch CPUs are shared efficiently among all slices, FV limits the rate of messages sent between SDN switches and tenant controllers. For this, FV implements a software-based message policing (briefly software isolation). This message policing needs to be specified and installed with the respective slice configuration. We will report in more detail on FV's software isolation in the measurement Section 3.4.
- *Flow Entries Isolation.* To efficiently realize the match-plus-action paradigm, SDN switches store match-plus-action instructions in flow tables. In case tenants demand Gigabit transmissions, this lookup needs to perform in nanoseconds. A special memory that operates in such timescales is TCAM [FKL04]. However, due to its cost, TCAM space is limited. FV reserves each tenant parts of the table space for operation. For this, FV keeps track of the used table space per tenant. Tenants cannot use table space of other tenants in case they demand more table space. FV sends tenants, which exceed their capacities, an indicator message telling that the flowspace is full.
- *Control Channel Isolation.* To distinguish between vSDNs, each vSDN is assigned its distinct transaction identifier. If tenants use the same identifier, FV rewrites the OF transaction identifiers to make them distinct again. It further modifies controller buffer accesses and status messages to put isolation into effect.

3.1.2.2 OpenVirteX (OVX)

OpenVirteX (OVX) [ADG+14; BGH+14; ALS14] builds up conceptually on FV. While FV only runs in software, OVX also makes use of networking elements to realize network virtualization functionalities. It extends address isolation and topology abstraction.

- *Architecture.* For virtualization of SDN networks, OVX makes use of general-purpose network elements, e.g., switches that are compliant with the OF specification 1.0 [Ope09]. While the main network virtualization logic still runs in software on a computing platform, such as an x86-based server, OVX makes use of rewriting and labeling features of switches for virtualization. It supports OF 1.0. As another feature, it supports node and link resilience.

- *Address Isolation.* Instead of using flow headers to differentiate tenants, OVX relies on a rewriting-based mechanism. To mimic a fully available flowspace, OVX rewrites IP and Media Access Control (MAC) addresses to virtually assigned IP and MAC addresses. This, however, introduces additional overhead on the data plane.
- *Topology Abstraction.* OVX is not working transparently. It intercepts the topology discovery process of switches. In case tenants use LLDP [802] for topology discovery, OVX answers as a stakeholder of the switches. Thus, it prevents intermediate switches of a physical path, which realize a virtual path, to show up in the tenant's topological view. Moreover, OVX provides resilience as a special feature. It can map a virtual path to multiple physical paths that connect the virtual endpoints.

3.1.2.3 Network Hypervisor Classification

Network hypervisors can be classified into centralized and distributed hypervisors.

Centralized hypervisors. Centralized hypervisors consist of one entity that is executed on a general-purpose computing platform, e.g., running in a virtual machine on a server in a data center. FV [SGY+09] has been the initial proposal of such hypervisor architecture for SDN-based networks. Many *General Hypervisors Building on FlowVisor* have followed. Other hypervisors are designed for special network types or use-cases like *Policy-based Hypervisors*.

- *General Hypervisors Building on FlowVisor.* Many network hypervisors have built up on the FV concept: *AdVisor* [SCB+11], *VeRTIGO* [CGR+12], *Enhanced FlowVisor* [MKL+12], and *Slices Isolator* [EABL+11]. Focusing on isolation, *Enhanced FlowVisor* extends FV's isolation capabilities. *Slices Isolator* focuses on improving switch resource isolation.
- *Policy-based Hypervisors.* Research on policy-based hypervisors is motivated by the fact that current SDN controllers do not provide (1) any topology abstractions for networking applications and (2) that control of network applications is not jointly optimized. For this reason, policy-based hypervisors, such as *CoVisor* [JGR+15], have been proposed to provide abstraction and joint optimization of networking application actions. Note that policy-based hypervisors are not particular designed to support multi-tenancy, but rather (1) to provide OF as a communication protocol on top of an SDN controller and (2) to optimize the joint operation of network applications.

Distributed hypervisors. We classify a hypervisor, like OVX, to be distributed if its functionality is distributed across the network elements. Network elements can either be computing platforms (like x86-based machines), general-purpose network elements, or special-purpose network elements. General-purpose network elements are, for instance, switches that are compliant with standard OF specifications, while special-purpose network elements implement special data/control plane functionalities for the purpose of network virtualization. So we further classify hypervisors by their execution platform they run on: *general computing platform*, *computing platform + general-purpose network elements*, and *computing platform + special-purpose network elements.*

- General Computing Platform: The hypervisors *FlowN* [DKR13], *Network Hypervisor* [HG13], *AutoSlice* [BP12; BP14] and *NVP* [KAB+14] run on general computing platforms, e.g., x86-based server platforms or inside virtual machines.
- Computing Platform + General-Purpose Network Elements: Two hypervisors that make use of general-purpose network elements are *OpenFlow-based Virtualization Framework for the Cloud (OF NV Cloud)* [MJS+11] and *AutoVFlow* [YKI+14a; YKI+14b].
- Computing Platform + Special-Purpose Network Elements: Hypervisors that rely on networking elements with special modifications are *Carrier-grade* [DJS12; SJ13], *Datapath Centric* [DSG+14], *Distributed FlowVisor (DFVisor)* [LLN14; LSL15], *OpenSlice* [LMC+13; LTM+11], and the *Advanced Capabilities OF virtualization framework* [SGN+12].

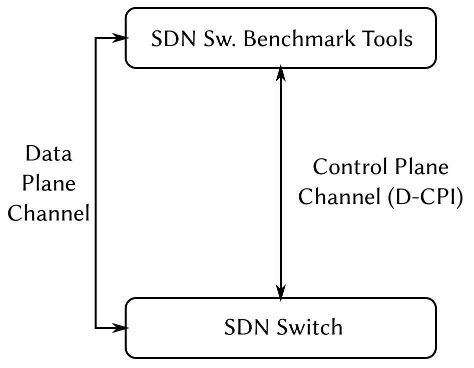
3.1.3 SDN Analysis and Benchmarking

In this section, the state-of-the-art on SDN switch and SDN controller benchmarking tools is highlighted first. The section also summarizes existing performance metrics. It also identifies components that affect SDN network operations, e.g., by adding latency overhead due to control plane traffic processing. Generally, benchmarking SDN networks can be split as follows: analyzing and benchmarking of SDN switches and analyzing and benchmarking of SDN controllers.

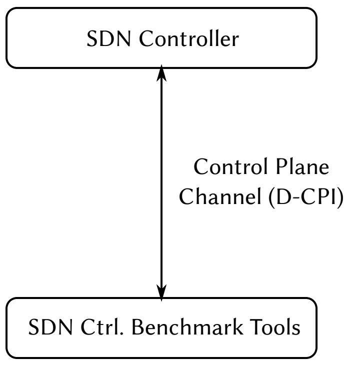
3.1.3.1 SDN Switch Analysis and Benchmarking

OF defines a common interface among switches, but it does not specify how switches need to perform under varying workloads. Switches from different vendors even behave differently given the same traffic events [LTH+14; DBK15], which leads to another source of unpredictable network performance. Accordingly, measuring and modeling SDN switches is needed to provide predictable network performance to accomplish complex networking tasks.

Concept. Fig. 3.1a shows the non-virtualized case of benchmarking an OF switch. The switch is connected with the benchmarking tool on the control plane through the D-CPI as well as on the data plane trough its physical ports. The measurement tool takes two roles: (1) the SDN controller role and (2) the role of injecting network traffic. This setup allows tools to benchmark the entire processing chain of network traffic flows. For instance, the tool can (1) send a network packet, (2) receive a notification from the switch for this packet, e.g., if no flow entry exists, (3) send an action, and (4) receive the effect of the action, e.g., the forwarded network packet. Whereas operation verification tools proof the correct implementation of OF specifications by a device [Oft], benchmarking tools try to reveal the performance of the device under varying network traffic conditions [RSU+12]. Many variations of traffic measurement scenarios should be considered to create knowledge about the behavior of switches.



(a) SDN switch benchmarking.



(b) SDN controller benchmarking.

Figure 3.1: Switch benchmarking and controller benchmarking setups in non-virtualized networks. The benchmarking tools directly connect to the SDN switches/SDN controllers.

Tools have been proposed to verify and benchmark OF-based SDN switches, where prominent representatives are OFTest [Oft], OFLOPS [RSU+12], and FLOPS-Turbo [RAB+14]. OFTest verifies the implementation of OF switches. OFLOPS benchmarks switches under varying workloads to reveal potential bottlenecks, i.e., unpredictable switch behaviors. Furthermore, OFLOPS can shed light on monitoring capabilities of OF switches and potential cross-effects that result from the simultaneous execution of OF operations (e.g., high data plane load while pulling switch statistics). FLOPS-Turbo is a hardware implementation of OFLOPS.

Performance indicators. Among all existing OF measurement studies, different performance indicators have been identified, which can be used to quantify the performance of OF switches. Generally, two performance aspects of switches can be measured: the control plane performance and the forwarding- or data plane performance.

- *OF Packet Processing.* OF defines several messages to be sent between controllers and switches. The throughput of these messages and their latency values are indicators for the overall OF packet processing performance. The processing of these messages might vary among switches. Hence, SDN measurements should study the performance among different OF messages.
- *OF Flow Table Updates.* SDN controllers trigger the update of flow tables. With OF, table operations are adding, deleting, or updating existing entries. As such operations might happen at runtime, they should be completed within small timeframes and in reliable manner. However, different measurement studies show unexpected behaviors for these operations for SDN switches [KPK14], e.g., the confirmation of table updates by the control plane even before the action is put into effect on the data plane.
- *OF Monitoring.* Precise and accurate monitoring of OF statistics is important, e.g., for charging or network state verification. Accordingly, knowledge about the temporal behavior of network statistic acquisition is important for predictable network operation.
- *OF Operations Cross-Effects.* Network operations have in common that they can happen simultaneously. While an SDN controller might trigger flow table updates, an SDN switch might

send OFPT_PACKET_IN messages towards the controller. As these network operations might cross-effect each others' performance, comprehensive measurement studies should particular focus on such scenarios.

- *Forwarding Throughput and Packet Latency.* All recently mentioned operations are put into relation to control plane operations. For quantifying the performance of forwarding operations, benchmarks should consider legacy performance indicators such as throughput and forwarding delay. The OF specifications define many data plane operations (re-writing, dropping, labeling) that might impact the data plane performance. Accordingly, benchmarks should measure the performance indicators for all possible data plane operations. This encompasses simple tasks (e.g., forwarding) to more complex tasks (e.g., labeling).

3.1.3.2 SDN Controller Analysis and Benchmarking

Operators will only choose SDN controllers as a credible alternative to control logic running on hardware if they deliver predictable performance. As operators should not suffer from unpredictability, they need detailed benchmarks on the performance of SDN controllers.

Concept. Fig. 3.1b shows the benchmarking for SDN controllers. The benchmark tool mimics the behavior of SDN networks, i.e., it should be capable to emulate one to many switches. It connects to the SDN controller through one to many D-CPI connections accordingly. By receiving, processing, and transmitting control messages through the control channel, operation of SDN controllers can add overhead.

The OF protocol specifies the way how SDN controllers need to behave to operate SDN networks. In order to quantify their behaviors, controller benchmark tools have been proposed, e.g., Cbench [TGG+12], OFCBenchmark [JLM+12], OFCProbe [JMZ+14].

Performance indicators. Mostly, these benchmark tools define the controller response time and the OF message throughput as performance indicators:

- *Controller Response Time.* The controller response time defines the time an SDN controller needs to respond to a network event, e.g., OFPT_PACKET_IN as a result of a table miss of a switch. The faster the response, the better the controller response time. Reliable network operations might also constrain the controller response time. For instance, in wide area network scenarios, the number of controllers might be determined by the targeted controller response time, which might be highly affected by the distance between controllers and switches. Accordingly, it can also serve as a scalability indicator for a controller architecture. Again, controller response times should be measured under varying control plane message loads and mixes. Only such measurements might reveal potential resource interference problems among controller operations.

- *Controller OF Message Throughput.* SDN controllers can be exhausted by the amount of messages they have to process. Accordingly, it is important to know such values in advance when assigning network tasks to controllers or when deciding for a controller architecture to accomplish a particular network operation scenario. The controller OF message rate determines the amount of messages a controller can either maximally accept before dropping a message due to buffer overflows; or it can be used to determine the rate at which a controller can operate while still running in a stable manner (i.e., without a continuous increase in response time). As SDN controllers have to serve a large amount of switches in large-scale networks or data centers, the message throughput should again be measured under varying workloads and a changing number of switches. When controllers need to react to simultaneous events, e.g., OFPT_PACKET_IN events from many switches, such workloads might show again potential controller resource interference issues. Accordingly, a high, fast, and reliable OF message processing is important for predictable performance of network operations.

3.1.4 Related Work

The related work consists of two parts. The first part reports on measurement studies in SDN-based networks. The second part briefly outlines work that addresses interferences in computer and network systems, such as clouds.

SDN Measurements

Performance and measurement aspects of OF have been studied before in the literature. Bianco et al. [BBG+10] initially benchmark OF switches; they introduce forwarding throughput and latency as performance indicators. Another comparison between OF switches and software implementations is conducted by [THS10]: a software switch outperforms existing hardware switches by achieving 25 % higher packet throughput. With respect to control plane performance, first studies revealed that OF can lead to performance problems due to high CPU loads on OF switches [CMT+11]. Other studies show that OF architectures may not handle burst arrivals of new flows [JOS+11]: e.g., eight switches can already overload controllers in data centers [PJG12].

Hendriks et al. [HSS+16] consider the suitability of OF as a traffic measurement tool (see [YRS15] for a survey on the topic), and show that the quality of actual measured data can be questionable. The authors demonstrate that inconsistencies and measurement artifacts can be found due to particularities of different OF implementations, making it impractical to deploy an OF measurement-based approach in a network consisting of devices from multiple vendors. In addition, they show that the accuracy of measured packet and byte counts and duration for flows vary among the tested devices. Also other authors observed inconsistencies between bandwidth measurement results and a packet-based ground truth [ADK14]. OF monitoring systems are implemented similarly to NetFlow, and accordingly, problems regarding insufficient timestamp resolution [Kog11; TTS+11], and device artifacts [CSO+09] also apply. Finally, Kuźniar et al. [KPK14; KPK15] report on the performance characteristics of flow table updates in different hardware OF switches, and highlight differences between the OF specification and its implementations, which may threaten operational correctness or even network security.

To the best knowledge, literature does not provide yet measurements that particularly target interference in multi-tenant scenarios for SDN-based networks.

Cloud Resource Interference

Clouds are realized by one or many data centers, which consist of tens of servers whose performance is shared among many tenants [AFG+10; JS15]. Performance unpredictability has been identified as one of the ten most critical obstacles to the success of cloud computing [AFG+10]. While resource sharing of virtual machines has been a well known issue, another source of unpredictability has been discovered in cloud environments over last years: the network [BCK+11; NVN+13]. In data centers, accordingly both resource interference problems are now coming together: virtual machines compete for resources on the servers and for resources on the network.

If we look into the cloud, physical resources are now shared among tenants: clusters consisting of many virtual machines are communicating via a shared network. This sharing, however, leads to severe performance unpredictabilities: the performance of the applications varies heavily dependent on, e.g., the daytime or the underlying server hypervisor [RNM+14] - tenants, however, always expect the same performance [SDQR10]. In such situations, tenants may even have to pay more for the same result. The main reasons for the unpredictability are the over-subscription of the cloud resources, but also the lack of inefficient resource management schemes, and the lack of explicit information exchange between tenants and providers about expected and achievable performance.

Accordingly, research focused on the understanding of the interactions between cloud applications and the hosting infrastructure: such information is actually important for both cloud infrastructure designers and software developers [TMV+11]. Models for better information exchange were also developed to improve application predictability [BCK+11].

With the occurrence of Network Function Virtualization (NFV), packet processing in software on commodity hardware perceives another significance. In particular to be an alternative to special-purpose hardware, general-purpose-based packet processing needs to provide predictable performance: operators simply cannot accept unintended behavior from their infrastructures [DAR12].

Various methods have been proposed to improve performance predictability. For instance, Novaković et al. [NVN+13] try to identify virtual machines suffering from performance variability, and then to adapt the placement to improve a VM's situation. Other concepts try to alleviate unpredictability by observing, learning, and predicting favorable and unfavorable placements [RNM+14]. This information is then included in resource placement decisions. To guarantee performance, concepts still rely on building isolated groups of links for each tenant [ZSR+16]. The concept relies on the separation between most demanding and less demanding tenants: the most demanding tenants then get isolated capacities to not interfere with less demanding tenants. HCloud makes its resource assignment based on the unpredictability factor of a service request, which is again based on measurements of how the underlying infrastructure interacts with the applications [DK16].

Interestingly, although a network hypervisor lies at the heart of any multi-tenant and network-virtualized system, the network hypervisor and especially its performance implications have received little attention so far.

3.2 Measurement Procedure for Network Hypervisors

Performance benchmarking of SDN network components is a basic prerequisite before deploying SDN networks. Since SDN introduces new levels of flexibility with respect to the implementation and geographical deployment of network functionality, also the potential sources of network operation interference change: for instance, control traffic between controllers and switches can be affected by the network; additional entities running on switches such as the OF agents are responsible for managing control plane traffic, while their execution might be affected by the processing power of switches. Such entities are potential sources of overhead and interference, which need to be identified first in order to take those into account for later operation.

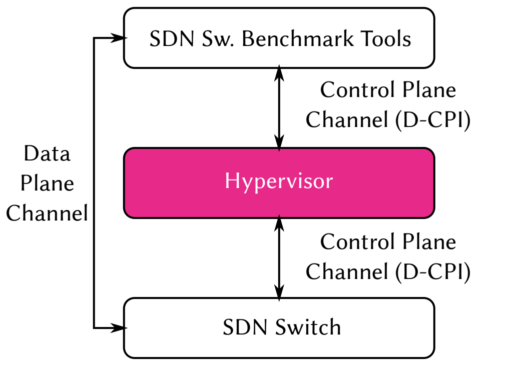
When looking at hypervisor architectures from top to bottom, a first source of interference is coming from the network connecting tenants with hypervisors. The tenant traffic might share the network resources, thus, their control plane traffic might interfere. Further, when hypervisors process tenants control traffic, the traffic might again share one hypervisor function and, accordingly, the available resources for this function (CPU, memory, network traffic). Besides, the hypervisor functions themselves share the available resources of a network hypervisor. Furthermore, like for the traffic between tenant controllers and hypervisor instances, network traffic from different tenants might interfere on their way towards the infrastructure. Finally, switches process again all hypervisor traffic, which might lead to interference due to the simultaneous processing of tenants' control and data plane traffic.

Two-step hypervisor benchmarking framework. A two-step procedure is proposed for benchmarking network hypervisors. The first step is to benchmark hypervisor functions, if possible, in an isolated manner. The second step is to quantify hypervisor performance as a whole, i.e., to benchmark the overall system performance of a virtualized SDN network. For varying workload setups, the system measurement provides insights into processing overhead, potential bottlenecks, and potential resource interferences. Generally, all OF-related measures can also be used to quantify hypervisor performance.

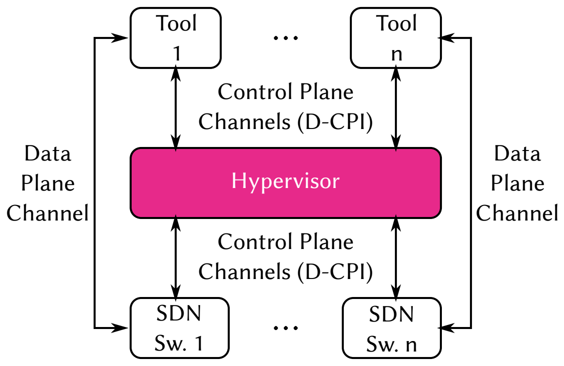
The operational performance quantification of hypervisors can be split into two parts: performance with respect to their operations towards switches and their performance with respect to how they behave towards SDN controllers. Furthermore, all performance measurements should be conducted for single and multi-tenant setups. Single-tenant setups, i.e., setups with only one tenant, can quantify the overhead per tenant. In such setup, overhead is expected to come from traffic processing only. Multi-tenant setups particularly try to reveal potential resource interferences due to sharing hypervisor functions (abstraction, isolation, etc.) and resources (CPU, network I/O, Memory, etc.) among tenants.

vSDN switch benchmarking. vSDN switch benchmarking differentiates between single-switch setups and multi-switch setups.

- *Single-Switch Setup.* To isolate overhead due to interference, hypervisor overhead should first be measured for a single-switch setup, as shown in Fig. 3.2a. Such setup should reveal the

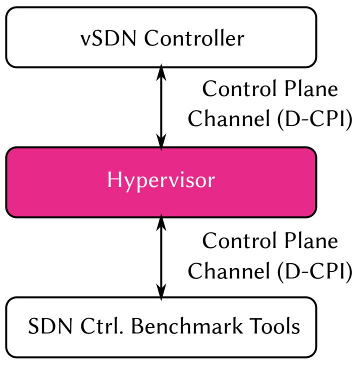


(a) Virtualized SDN switch benchmarking: single vSDN switch.

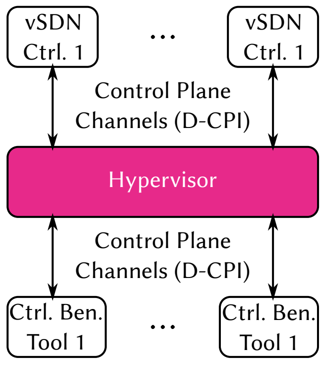


(b) Virtualized SDN switch benchmarking: multiple vSDN switches.

Figure 3.2: Two virtual switch benchmarking setups: left setup with a single switch; right setup with multiple switches and controller emulations. Tools emulate control plane or data plane traffic that needs to pass through the virtualization layer, i.e., the network hypervisor.


vSDN Controller
Control Plane Channel (D-CPI)
Hypervisor
Control Plane Channel (D-CPI)
SDN Ctrl. Benchmark Tools

(a) Virtualized SDN controller benchmarking: single vSDN controller.



(b) Virtualized SDN controller benchmarking: multiple vSDN controllers.

Figure 3.3: Two virtual controller benchmarking setups: left setup with a single vSDN controller; right setup with multiple switch emulations and vSDN controller emulations. Tools emulate control plane or data plane traffic that needs to pass through the virtualization layer, i.e., the network hypervisor.

performance capabilities of hypervisors to abstract a single vSDN switch. It should answer questions like how hypervisors manage switch resources, how they monitor them, and how they manage them in case of failures. To quantify the overhead, the performance results should be compared to a setup without a hypervisor. Single-switch measurement results serve as a baseline for the multi-switch setup.

- *Multi-Switch Setup.* Fig. 3.2b shows an exemplary multi-switch setup. The multi switch setup should again identify overhead, operational problems, or resource inefficiency. Handling multiple switches at the same time might add additional overhead due to message arbitration or resource interference. Furthermore, a hypervisor needs to make routing decisions or it has to decide on how many switches it should abstract to form a big switch representation.

vSDN controller benchmarking. Similar to benchmarking vSDN switches, hypervisor performance measurements should be conducted for single vSDN controller setups and multi vSDN controller setups.

- *Single-Controller Setup.* Like for SDN switches, SDN controller benchmark tools can measure the hypervisor performance in a single-controller setup. Fig. 3.3a shows a setup where one vSDN controller is connected to the hypervisor, which is connected to one controller benchmark tool. Again, measures as defined for SDN networks can be used to quantify the performance and induced overhead of network hypervisors.
- *Multi-Controller Setup.* Also similar to the multi-vSDN-switch setup, hypervisor performance models have to be quantified for scenarios with multiple controllers. Fig. 3.3b shows the setup where one or many controller benchmark tools are connected to the hypervisor, which connects to multiple tenant controllers. This setup should again provide insights into how the hypervisor can process potentially simultaneous requests from different tenants. Here, the tenants may send OF messages of the same or different types, at different rates, but at the same time. In such setup, resource interference might even lead to higher processing times and variations than compared with a setup with a single controller.

Benchmarking of individual hypervisor functions. For hypervisors, the individual function benchmarking concerns the isolation and the abstraction functionality. All OF-related metrics from non-virtualized SDN environments can be applied to evaluate the functions in a virtualized scenario.

Performance implication due to abstraction. Abstraction benchmarking should involve the benchmarking of topology abstraction, node resource abstraction, and link resource abstraction. Abstraction benchmarking involves the evaluation of the resources that are needed to realize abstraction but also the potential overhead that is introduced due to abstraction. For instance, abstraction might demand additional processing, e.g., for rewriting control plane packets, from hypervisors or SDN switches, e.g., CPU resources. Furthermore, abstracting might increase control plane latency as its realization requires synchronization among distributed physical resources.

Performance implication due to isolation. Isolation benchmarking involves benchmarking three parts: control plane isolation, data plane isolation, and vSDN addressing isolation. Isolating resources (queuing, schedulers, etc.) might also introduce additional resource overhead, which has to be measured and quantified. Furthermore, isolation could be implemented in a work conserving manner, i.e., resources that are currently not used by tenants are freed to tenants demanding more resources. All mechanisms for abstraction and isolation need to be evaluated for different scenarios: underutilization, multi-tenancy, overutilization etc.

3.3 Measurement Tool for Benchmarking Network Hypervisors: *perfbench*

A hypervisor benchmark needs to cover a wide spectrum of possible workloads. For example, data centers may face traffic loads [BAM10; GHJ+09] that vary significantly in the number of network flows; from a few thousand to tens of thousands of flows; reports count 2×10^3 to 10×10^3 of flows for Amazon's EC2 and Microsoft's Azure clouds [HFW+13]. For a 1500-server cluster, measurements show that the median of flows arriving in the network is around 1×10^6/s [KSG+09], whereas networks with 100 switches can have peak flow arrivals of 1×10^9/s [BAM10]. It can be expected that the number of flows scales with the number of tenants running on top of the clouds, i.e., on top of the network hypervisors. Hence, the measurement studies in this thesis focus on message rates in the range of tens of thousands of messages per second.

For the purpose of this study, a novel benchmarking tool called *perfbench* is presented. *perfbench* is designed to measure the performance of network hypervisors for OF-based vSDNs. This section overviews *perfbench* and puts it into perspective with existing SDN tools, and with focus on the use case of this thesis: multi-tenant virtualized SDN networks.

3.3.1 Architecture and Implementation

perfbench is tailored toward precise and high throughput performance benchmarks for OF-based SDN networks to meet the requirements, e.g., of data center workload emulations [BAM10; GHJ+09]. It can emulate high OF control plane message rates, while it can be used for both non-virtualized SDN networks and for vSDNs. To meet the high demanding performance aspects, it builds on top of libfluid [VRV14], a C++ library providing basic implementations and interfaces for OF message processing and generation. As libfluid supports OF versions 1.0 and 1.3, *perfbench* can benchmark OF networks with the respective versions. *perfbench* can generate the following OF messages:

- OFPT_PACKET_IN
- OFPT_PACKET_OUT
- OFPT_ECHO_REQUEST & OFPT_ECHO_REPLY
- OFPT_FEATURES_REQUEST & OFPT_FEATURES_REPLY
- OFPT_FLOW_MOD and OFMP_PORT_STATS

Fig. 3.4 gives a conceptual view of *perfbench*'s design and how it operates in multi-tenant SDN networks. As the figure shows, *perfbench* consists of two parts: a control plane part called *perfbenchCP* and a data plane part called *perfbenchDP*. The measurement study in Section 3.4 deploys *perfbench* in both modes:

- *perfbenchCP* runs processes that can emulate multiple SDN controllers simultaneously. To avoid resource interference among those processes, leading to unpredictable measurements, each emulated controller is assigned its own thread. Further, to emulate a realistic multi-tenant setup, each controller process connects through a unique TCP connection to the network hypervisor under test. Every emulated controller can generate its own workload, following a traffic distribution with a given distribution of message bursts parameter and a given distribution for the inter-arrival times of bursts. The distributions can be modeled either deterministically, exponentially, or normally distributed. Note that for the bursts the equivalent discrete distributions are achieved by simple rounding. Currently, *perfbench* can distribute the inter-arrival times mean values ranging from 1 ms to 1 s. The send intervals trigger the sending

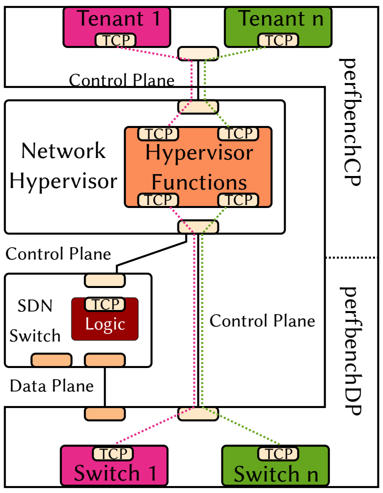


Figure 3.4: Conceptual view and architecture of *perfbench*. *perfbench* is split into two parts: *perfbenchCP* (top) and *perfbenchDP* (bottom). *perfbenchCP* controls the tenant processes. *perfbenchDP* either connects to the *Network Hypervisor* under test or to n SDN switches.

of bursts. Accordingly, the average burst size is then determined by the overall demanded average message rate per second.

- *perfbenchDP* emulates either data plane traffic or data plane switches. Hence, it has two modes of operation. In the first operation mode, it connects to data plane switches that are connected to the network hypervisor (or SDN switches). In this mode, *perfbench* generates User Datagram Protocol (UDP) packets to emulate data plane traffic. In the second operation mode, it emulates data plane switches, i.e., it directly connects to network hypervisors (or SDN controllers). To reduce interference, each emulated switch is assigned its own thread. Further, in this mode, *perfbench* generates the needed OF traffic to emulate a functional OF control channel interface.

A scheduler is responsible to manage the sending of the messages of the controller process(es) and switch process(es). The message rates are determined by the average demanded messages per second, whose inter-arrival times can again be generated following one of three distributions: uniformly distributed, exponentially distributed (or Poisson distributed per send interval), or based on a discrete Weibull distribution. The generation process can be stationary or non-stationary, i.e., the mean values might not need to be constant over time. This provides capabilities to generate more realistic network traffic behaviors.

3.3.2 Procedure for Latency Estimation

For latency estimation, two message types need to be differentiated: messages that require a reply and messages that do not explicitly need a reply. Messages like OFMP_PORT_STATS, OFPT_FEATURES_REQUEST, or OFPT_ECHO_REQUEST work in a request and reply manner. The estimated control latency is the time from sending the request until receiving the reply.

The latency estimation for messages such as OFPT_FLOW_MOD, OFPT_PACKET_IN or OFPT-_PACKET_OUT works different: for instance, there are two modes for OFPT_PACKET_IN. In the first mode, *perfbenchDP* sends UDP packets for each tenant on the data plane. The switches have no pre-installed rules for the UDP packets. As a result, the switches generate OFPT_PACKET_IN messages to ask the hypervisor (and subsequently the tenant controllers) how to handle the packets. The latency is then the time it takes from sending UDP packets until receiving their OFPT_PACKET_IN messages at *perfbenchCP*. In the second mode, *perfbenchDP* emulates switches. It directly sends OFPT_PACKET_IN messages per tenant to the hypervisor, which forwards them to *perfbenchCP*. In this case, the latency is the difference between *perfbenchCP* receiving the OFPT_PACKET_IN messages and *perfbenchDP* sending them.

Similarly, two cases exist for OFPT_PACKET_OUT messages. In both cases, *perfbenchCP* sends OFPT_PACKET_OUT messages. The hypervisor forwards these messages either to a real soft- or hardware switch (first case), or it forwards it directly to *perfbenchDP* (second case). In the first case, *perfbenchDP* would be connected to the switch to receive the data plane packets that were encapsulated in the OFPT_PACKET_OUT messages. When emulating the switch, *perfbenchDP* directly receives the data plane packets. In both cases, the latency is calculated as the time difference between *perfbenchCP* sending the OFPT_PACKET_OUT messages and *perfbenchDP* receiving the de-capsulated data plane messages.

3.3.3 Conceptual Comparison to SDN Benchmarking Tools

Tab. 3.1 compares *perfbench* to existing tools with focus on the four supported features: multi-tenant, control plane benchmark (CB), data plane (switch) benchmark (DB), and supported distributions for OF traffic generation. As described in Sec.3.1.3, SDN benchmark tools can be classified into two categories: switch and controller benchmark tools.

While *OFtest* [Oft] has originally been designed to verify switch implementations of OF 1.0, it can also be used for basic switch and controller performance measurements [BBK15; BBL+15]. Due to its Python-based implementation, it supports, however, only rates with up to 1 000 messages per second.

OFLOPS [RSU+12] is another tool to benchmark and verify switch implementations. It introduced further data plane metrics such as flow table update rates and flow insertion latencies. However, *OFLOPS* is designed for a single-switch setup, i.e., it cannot emulate multiple controllers simultaneously. Furthermore, it generates traffic in a best-effort manner.

CBench [TGG+12] and *OFCProbe* [JMZ+14] are designed to benchmark SDN controllers. Both tools can emulate multiple OF switches to measure controllers in terms of control plane throughput and message response latency. While *CBench* and *OFCProbe* can emulate multiple switches, they demand the operation of additional SDN controllers. Besides, *CBench* emulates switch-to-controller traffic only in a best-effort manner, which does not provide means to quantify varying traffic distributions.

Only *perfbench* and *hvbench* are designed for performance measurements of vSDNs, i.e., supporting the simultaneous emulation of multiple SDN controllers and switches. Like *perfbench*, *hvbench* uses libfluid and provides several OF message types for OF 1.0 and OF 1.3. However, *hvbench* al-

Table 3.1: Feature comparison with existing OF benchmarking tools. ✓indicates whether a tool supports a given feature. *Multi-tenant:* can interact with hypervisors. *CB:* controller benchmark. *SB:* switch benchmark. *OF traffic generation rate:* OF message sending behavior (best-effort, distribution)

Tool	Multi-tenant	CB	SB	OF traffic generation rate
OFTest [Oft]		✓	✓	best-effort
OFLOPS [RSU+12]			✓	best-effort: traces
CBench [TGG+12]		✓		best-effort
OFCProbe [JMZ+14]		✓		best-effort, distribution: pre-defined (Normal, ChiSquared, Exp., Poisson)
hvbench [SBB+16b]	✓			distribution: pre-defined (Exp.)
perfbench	✓	✓	✓	pre-defined distribution: pre-defined (Uniform, Exp., Weibull) and custom

ways demands the existence of a network hypervisor, i.e., it has not been designed for the purpose of measuring SDN networks. *hvbench* can only generate OF traffic whose message inter-arrival times follow an exponential distribution.

3.4 Measurement Evaluation of Network Hypervisors

This section focuses on benchmarking two hypervisors under varying network load: FlowVisor (FV) and OpenVirteX (OVX). It starts by briefly summarizing the existing hypervisor measurement results. Then, it proceeds to show the performance benchmark results.

3.4.1 Existing Hypervisor Measurements

As this thesis investigates control plane isolation mechanisms of network hypervisors, measurements falling under this category are briefly explained in the following paragraphs.

FlowVisor. Initial experiments on FV in real testbeds analyzed how much overhead FV adds and how efficient its isolation mechanisms are (bandwidth on data plane, flowspace, and switch CPU). In [SGY+09], an initial experiment quantifies how much latency overhead FV adds for a setup consisting of one switch. The switch connects through two physical interfaces to a measurement machine, one connection for data plane traffic and one connection for control plane traffic. The machine sends 51 packets per second. The flow setup time is measured from sending the packet until receiving the new flow message (OFPT_PACKET_IN) on the measurement machine. FV's performance is compared to a reference setup without virtualization. The reference setup shows an average latency of 12 ms. With FV, the latency is 16 ms on average. Hence, FV adds an overhead of 4 ms on average in this setup. To quantify the overhead for OF messages waiting for replies, a special-purpose controller is used. The special-purpose controller sends 200 OFMP_PORT_STATS messages on average; this was the maximum rate supported by the switch. The measured latency overhead is 0.48 ms on average. The authors argue that the better performance for OFMP_PORT_STATS is due to an optimized handling of this message type.

OpenVirteX. Evaluation results [ADG+14; BGH+14; ALS14] compare control plane latency overhead between OVX, FV, FlowN and a reference case without virtualization. Cbench [TGG+12] emulates five switches with a specific number of hosts for benchmarking. Each switch emulates one virtual network. OVX shows the lowest latency overhead of only 0.2 ms taking the reference use case as a basis.

VeRTIGO. VeRTIGO increases the flexibility of provisioning vSDNs, however, it comes with increased complexity. The evaluations [CGR+12] show an increase in average latencies for new flow requests by roughly 35 % when compared to FV.

FlowN. FlowN is compared to FV in terms of hypervisor latency overhead in [DKR13]. For an increasing number of virtual networks from 1 to 100, FV latency overhead also increases while FlowN's induced latency stays constant. Note, however, that the added latency overhead of FV is lower for 1 to 80 virtual networks.

AutoVFlow. For AutoVFlow, a multi-domain use case is benchmarked. Different OF message types were evaluated: OFPT_PACKET_IN, OFPT_PACKET_OUT, and OFPT_FLOW_MOD. OFPT_-FLOW_MOD faced the highest latency overhead of 5.85 ms.

Carrier-grade. Carrier-grade [DJS12; SJ13] implements network functions directly on switches. A basic evaluation measured an additional per hop delay of 11 % for one virtual network. More virtual networks were not measured, thus, no conclusions about potential interference can be given.

Datapath Centric. The latency overhead induced by Datapath Centric's VA agents is conducted in [DSG+14]. Compared to a reference case, the VA agents add a latency overhead of 18 %. When compared to FV, the average overhead is 0.429 ms. A second evaluation examines Datapath Centric's scalability for an increasing number of rules from 0.5×10^3 to 10×10^3. The added overhead stays constant from 100 to 500 rules, while it increases linearly up to 3 ms after 500 rules.

In summary, all results are not representative for production environments, like data centers where 100 switches can have peak flow arrivals of 1×10^9/s [BAM10]. Hence, the small message rates and topologies cannot reveal the true performance of hypervisors. As a consequence, resource interference in larger networking environments might have not yet been studied in detail for hypervisors.

3.4.2 Benchmark Settings for FlowVisor (FV) and OpenVirteX (OVX)

The following measurements systematically analyze potential factors and influences of the two network hypervisors FlowVisor (FV) [SGY+09] and OpenVirteX (OVX) [ADG+14] on the performance of vSDNs. Both hypervisors are publicly available. The measurements are structured into two parts: workload benchmarks with a single tenant and experiments with multiple tenants and multiple switches.

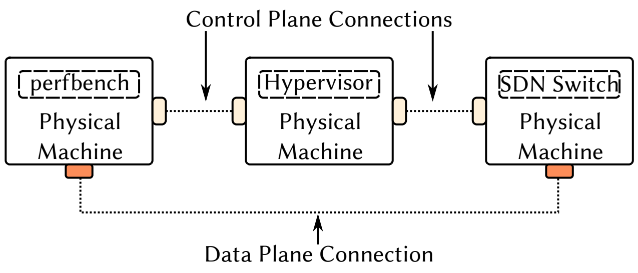


Figure 3.5: Network hypervisor benchmarking and measurement setup.

3.4.2.1 Measurement Setup and Test Cases

perfbench generates the workload to explore the performance implications of the network hypervisors. Fig. 3.5 shows the measurement setup. The setup consist of three PCs. Each PC has 16 GiB of RAM and 4 physical CPU cores (8 with Hyper-Threading): Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz. All PCs run Ubuntu 14.04.5 LTS with the kernel 3.19.0-26-generic x86_64. Throughout the measurements, all CPU policies are setup to ”performance” mode to avoid side effects due to CPU energy saving.

The left PC runs *perfbenchCP* and *perfbenchDP*, the middle PC runs the SDN hypervisor under test, and the right PC runs an Open vSwitch (OvS) instance if *perfbenchDP* does not emulate the data plane. *perfbenchCP* connects to the hypervisor PC, and the hypervisor PC connects to the PC hosting the OvS switch. *perfbenchDP* is either connected through a dedicated line to the data plane part of the OvS PC or directly to the hypervisor PC.

The latest versions of FV[1] and OVX[2] are used. *perfbench* is always configured to be capable to operate with the respective hypervisor. In case of OVX, *perfbenchDP* uses artificial unique MAC addresses per tenant: a pre-requisite for the operation of OVX. In order to study OVX for OFPT_FLOW_MOD messages, the OVX source code has to be modified. A flow table lookup inside OVX is disabled[3], as it is irrelevant for the investigated use cases. The flow table lookup would dramatically impact OVX; the results would be useless. FV does not demand special settings beside the configuration of the slices.

Table 3.2 provides an overview of all conducted measurements. Measurements are examined for different rates of different message types, single-tenant as well as multi-tenancy setups, and TCP_ND on/off settings. Every setup is repeated at least 10 times for a minimum duration of 30 seconds. The analyses cut off the first 10 and last 10 seconds as both underly transient effects: e.g., when starting a multi-tenant measurement, not all tenants are started exactly at the same time; hence, they might also stop at different points in time. The remaining durations are expected to provide meaningful insights into the performance under stable behavior.

[1] See https://github.com/opennetworkinglab/flowvisor/tree/1.4-MAINT last accessed 14. November 2017.
[2] See https://github.com/opennetworkinglab/OpenVirteX/tree/0.0-MAINT last accessed 14. November 2017.
[3] See https://github.com/opennetworkinglab/OpenVirteX/blob/master/src/test/java/net/onrc/openvirtex/elements/datapath/FlowTableTest.java last accessed 14. November 2017.

Table 3.2: Measurement configurations for single-tenant/switch and multi-tenant/multi-switch setups. The tenants column specifies the number of tenants. 5:20 tenants with a step size of 5, and 25:100 tenants with a step size of 25. The switch column shows whether an OvS switch was used or *perfbenchDP* (=1). Message rate is message per second with a message inter-arrival time of 1 ms. The messages are uniformly distributed among the inter-arrival times: e.g., for 10k, 10 messages are generated every 1 ms.

Hypervisor	OF Message Type	Tenants	Switch	Msg. Rate	TCP_ND
FV/OVX	OFPT_PACKET_IN	1	OvS	10k:10k:40k	0
FV/OVX	OFPT_PACKET_OUT	1	OvS	20k:10k:50k	0
FV/OVX	OFMP_PORT_STATS	1	OvS	5k:1k:8k	0
FV/OVX	OFPT_FLOW_MOD	1	1	1k-30k	0/1
FV	OFPT_FLOW_MOD	5:20	1	100 per tenant	1
OVX	OFPT_FLOW_MOD	25:100	1	100 per tenant	1
FV	OFPT_PACKET_IN	1	5:20	100 per switch	1
OVX	OFPT_PACKET_IN	1	25:100	100 per switch	1

3.4.3 Systematic and Exploratory Evaluation

This section targets to systematically answer important questions when virtualizing SDN networks. The control plane latency and the CPU utilization of the respective hypervisor are the performance indicators in this section.

The measurement studies are structured into two parts. First, Sec. 3.4.3.1 reports on the measurement results of FV and OVX for single-tenant and single-switch scenarios. Second, Sec. 3.4.3.2 deepens the measurement study towards multi-tenant and multi-switch setups.

3.4.3.1 Evaluation Results of Single-Tenant and Single-Switch Measurements

The original measurement results of FV and OVX show that both network hypervisors yield overhead due to adding an indirection layer. However, the original studies lack detailed investigations for different message types, rates, and tenant configurations. Hence, this section focuses on answering the following questions:

- What is the impact of network hypervisor implementations?
- What is the impact of OF message types and rates?
- What is the impact of tenant controller implementations?

How does the control plane throughput depend on the specific network hypervisors? The first experiment studies the control plane throughput (maximum rate per second) of FV and OVX with OvS as the data plane switch. It provides guidelines for the settings of the next experiments. As cross effects due to multiple tenants interfering on the resources should be excluded, the setup

Table 3.3: Hypervisor control plane messages throughput (maximum OF message rate per second) with a single tenant and a single switch. **Note these measurement settings are not listed in Table 3.2. All measurements are conducted with OvS.**

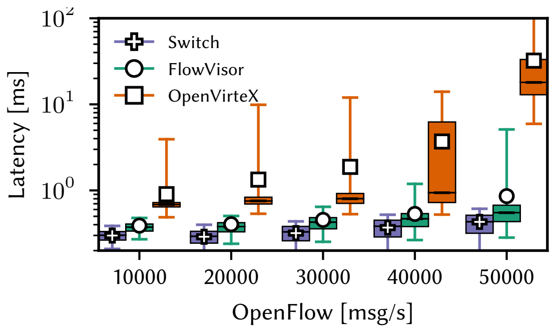
OF Message Type	FV	OVX
OFPT_PACKET_IN	58,170 ± 123	51,941 ± 579
OFPT_PACKET_OUT	57,980 ± 247	51,899 ± 301
OFPT_FLOW_MOD	39,975 ± 138	31,936 ± 402
OFMP_PORT_STATS	7,993 ± 22	199,937 ± 34

consists of a single switch and a single controller. *perfbench* generates OF messages that are steadily increasing during a measurement run: when hypervisor CPUs are starting to be overloaded, the control plane latency first steadily increases (the buffers bloat) until losses start (the buffers overflow). The maximum OF message throughput is determined as the rate before reaching the buffer overflow.

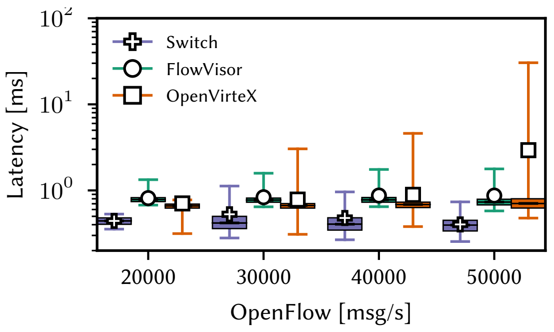
Table 3.3 shows that FV can provide a higher throughput for asynchronous OFPT_PACKET_IN, OFPT_PACKET_OUT and OFPT_FLOW_MOD messages, e.g., FV can support $\sim 7 \times 10^3$ msg/s of OFPT_PACKET_IN more than OVX. This can be explained by OVX's data message translation process: OVX includes data plane packet header re-writing from a given virtual IP address, specified for each tenant, to a physical IP address used in the network. This is done in addition to control message translation. Note that for both hypervisors, the supported rate of OFPT_FLOW_MOD messages is lower than the received OFPT_PACKET_IN rate, which might set the upper bound for the flow setup (new connections) rate if all OFPT_PACKET_IN messages demand reactions (OFPT_FLOW_MOD).

For synchronous OFMP_PORT_STATS messages, OVX shows much higher throughput ($\sim 200 \times 10^3$ msg/s) compared to FV (only $\sim 8 \times 10^3$ msg/s). Since FV transparently forwards all messages to the switch, the switch becomes the bottleneck for OFMP_PORT_STATS throughput. OVX uses a different implementation for synchronous messages: it does not forward all requests for port statistics to a switch transparently, but rather *pulls* statistics from the switch, given a pre-configured number of times per second. The default pulling rate of OVX is 1 OFMP_PORT_STATS message per second. OVX replies on behalf of a switch to all other requests (using the same port statistics); hence, OVX increases throughput in receiving OFMP_PORT_STATS messages. However, all tenants are limited by the OFMP_PORT_STATS pulling rate set by OVX. In fact, the "factual" OFMP_PORT_STATS throughput of OVX is equal to its statistics pulling rate.

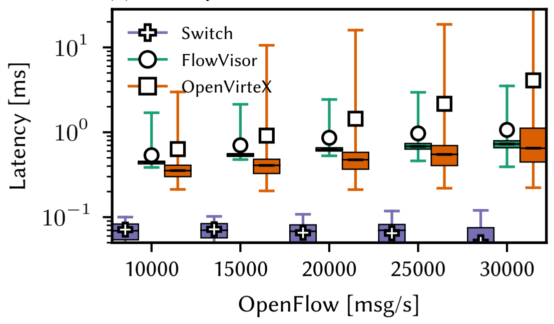
How much overhead do network hypervisors add to the performance? The following evaluation compares FV and OVX for four message types: OFPT_PACKET_IN, OFPT_PACKET_OUT, OFPT_FLOW_MOD and OFMP_PORT_STATS. Fig. 3.6 shows the performance overhead induced by the indirection of the control plane traffic through a hypervisor. For instance, Fig. 3.6a showcases the latency for OFPT_PACKET_IN messages arriving at rates between 10×10^3 msg/s and 50×10^3 msg/s (message per second).



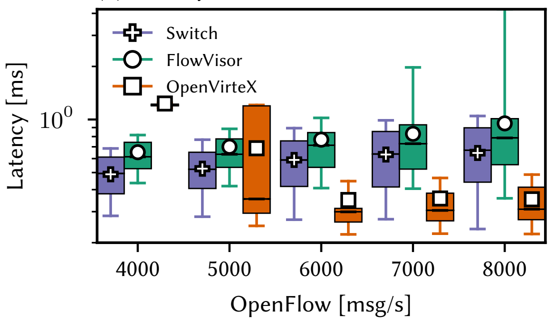
(a) Latency: OFPT_PACKET_IN.



(b) Latency: OFPT_PACKET_OUT.



(c) Latency: OFPT_FLOW_MOD (with *perfbenchDP*).



(d) Latency: OFMP_PORT_STATS.

Figure 3.6: Boxplots of OF control plane latency in milliseconds [ms] for four message types: OFPT_PACKET_IN, OFPT_PACKET_OUT, OFMP_PORT_STATS, OFPT_FLOW_MOD. Note the different message rates. Message rate is always given in messages per second. The boxplots show the mean values via markers and the median values via black lines. The whiskers illustrate the lower 2.5 % and the upper 97.5 % control plane latency values of all measurement runs. TCP_ND is always enabled. Switch measurements show the best performance for OFPT_PACKET_IN, OFPT_PACKET_OUT, OFPT_FLOW_MOD (*perfbenchDP*). FV shows the second best performance for these message types. OVX achieves the best performance for OFMP_PORT_STATS. Generally, with increasing message rates, hypervisors linearly increase the achieved latencies. In contrast, switch latencies remain stable and show only a small increase.

The switch achieves average latencies from 0.3 ms increasing to 0.5 ms for the highest rate. FV and OVX add significant latency overheads due to their network processing: FV results in average latencies between 0.4 ms and 1 ms; OVX adds even more overhead with average values from 1 ms up to 30 ms. Moreover, OVX shows significantly higher latency variations, which is due to its multi-threading-based implementation. Whereas more threads can make use of free CPUs or show advantages in case of waiting for I/O operations, they potentially introduce overhead due to blocking and freeing resources. Such observations need to be considered when making different performance guarantees, like average or maximum latency guarantees.

Although the latency increases by an order of magnitude, the latency still seems small in absolute terms for OFPT_PACKET_IN. However, especially in latency critical environments, such latency values may be unacceptable, and also introduce unacceptable uncertainties. For instance, latency communication is a primary metric for building data center and rack-scale networks [AKE+12; SCS+15]. Outside of data centers, according to [VGM+13], even slightly higher web page load times can significantly reduce visits from users and directly impact revenue.

How do hypervisor implementations affect the control plane latency for different message types and rates? As Fig. 3.6a-Fig. 3.6d illustrate, no virtualization gains the lowest latency values, whereas virtualization always adds latency overhead again. Looking at the hypervisor results, FV features again a lower latency than OVX, especially at high message rates. OVX results in higher-varying latency values and more outliers with varying rates in general: OVX's performance is less predictable. For instance, OFPT_PACKET_IN messages have an average of 1 ms for 10×10^3 messages, up to an average of 30 ms for 50×10^3 messages (Fig. 3.6a).

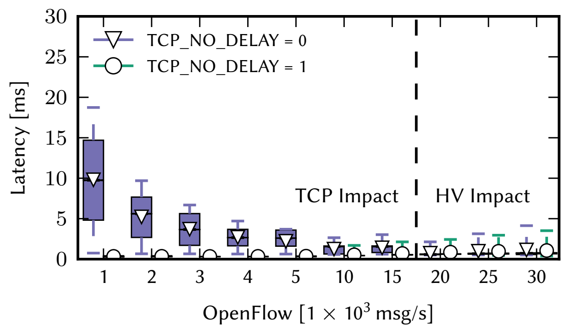
As already explained in Sec. 3.4.3.1, the translation mechanism of OVX leads to processing overhead on switches. Also the high variances with OVX for 40×10^3 and 50×10^3 messages, as also indicated by Fig. 3.6b, make predictability even harder. In contrast, FV operates in a transparent manner as it does not instruct switches to change data plane packet headers. It even operates on a microsecond precision on average for all evaluated rates. The OFPT_PACKET_IN and OFPT_PACKET_OUT handling at FV results in lower control latency and exhibits less variability even under varying control rates when compared to OVX.

Fig. 3.6c reports on the behavior for OFPT_FLOW_MOD messages. Again, the switch shows the best latency performance. Interestingly, while FV shows a better average latency, 75 % of its latency values are worse than the ones with OVX. OVX's multi-threading-based implementation can efficiently improve the latency for the majority of the messages (75 %), however, it introduces higher variance.

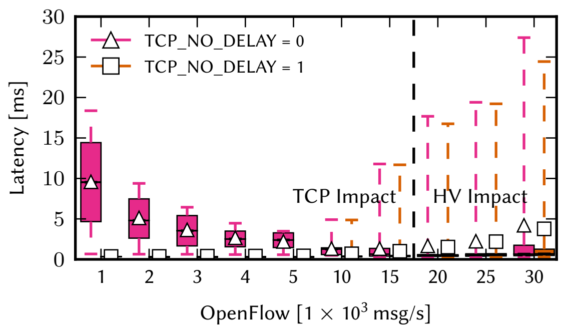
Fig. 3.6d shows the results for OFMP_PORT_STATS for a rate from 5×10^3 to 8×10^3 messages per second. The maximum rate is set to 8×10^3 messages, as a rate higher 8×10^3 overloads OvS, thus latency increases and even becomes instable. As Fig. 3.6d shows, OVX now outperforms FV due to its implementation as a proxy which does not transparently forward all messages. FV's design relies on transparency, i.e., it forwards all messages to the switch, which overloads the switch. In contrast, OVX replies on behalf of the switch; hence, OVX avoids an overload of the switch: this significantly improves control plane latency, but of course, might lead to outdated information about switch statistics.

As a conclusion, the best achievable performance per message type depends on the chosen network hypervisor: for different message types each hypervisor shows better performance in terms of predictability, e.g., in terms of average and maximum latency. Such observations need to be taken into account when a network hypervisor needs to be chosen for a specific network scenario. This means that based on the tenants' demands, e.g., whether a tenant requests more flow modifications or wants to have a more precise network monitoring, the virtualization layer should be composed of varying implementations of virtualization functions.

How does the tenant's controller impact the hypervisor performance? The next investigation focuses on the impact of the implementation aspects of SDN controllers on network hypervisor performance, and vSDN performance in general. For this, the controller operating system configures its TCP connection with TCP_ND set to 0 or 1: TCP_ND $= 1$ disables Nagle's algorithm. The controller socket is forced to send its buffered data, which leads to more network traffic as more smaller packets are sent. This configuration possibly improves TCP performance in terms of latency. The



(a) Hypervisor: FV Latency.



(b) Hypervisor: OVX Latency.

Figure 3.7: Boxplots showing control plane latency impact of the TCP_ND feature on the hypervisor performance. The message type is OFPT_FLOW_MOD. Left figure shows FV, right figure OVX. Unit scales on x-axis are 10×10^3 msg/s. TCP impact can be observed for rates up to 15×10^3 msg/s. Message rate and hypervisor (HV) impact latency for rates higher than 15×10^3 msg/s.

performance of both hypervisors is evaluated for OFPT_FLOW_MOD messages. The measurements are carried out at message rates between 1×10^3 and 30×10^3 messages per second.

Fig. 3.7 illustrates that for small rates (1×10^3 to 15×10^3), with TCP_ND $= 0$, the TCP aggregation behavior determines the control plane latency: this holds for both FV and OVX. Disabling Nagle's algorithm leads to a significant latency improvement for both hypervisors. When sending more than 20×10^3 messages per second, there is no clear difference for the latency between the TCP settings. The reason is that the TCP_ND flag has no impact on the sending behavior of the tenant anymore as its socket buffer becomes saturated. The socket process can always completely fill TCP packets (up to 1500 bytes) and does not wait artificially until a packet is completely filled.

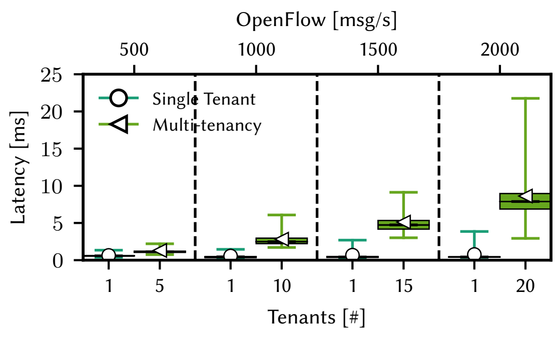
When the transmission behavior becomes independent of the TCP_ND setting, the way how hypervisors process messages determines control plane latencies. Whereas FV only shows a small latency increase, OVX shows a significant latency variation. Again, the reason for the latency increase is due to OVX's packet re-writing procedure and its multi-threaded processing implementation.

3.4.3.2 Evaluation Results of Multi-Tenant and Multi-Switch Measurements

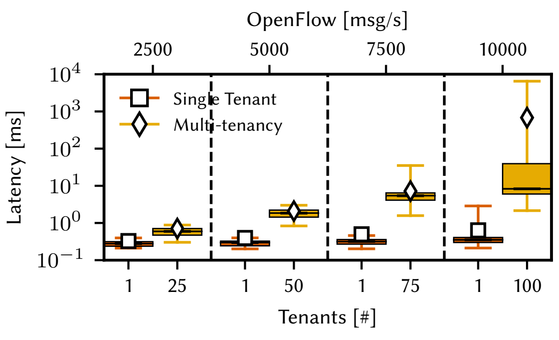
Hypervisors show varying control plane latencies for different OF message types for different rates. This section focuses on scenarios with multiple tenants and multiple switches. The following general questions should be answered:

- Do multiple tenants lead to resource interference?
- Do multiple switches lead to resource interference?
- If interference happens, how severe is it?

How does the control latency and CPU utilization depend on the number of tenants? The next measurements evaluate the impact of the number of tenants on the hypervisor performance: ideally, the performance provided for a single virtual network should scale (transparently) to multiple tenants. As an example: the performance (e.g., in terms of control latency) of two tenants with

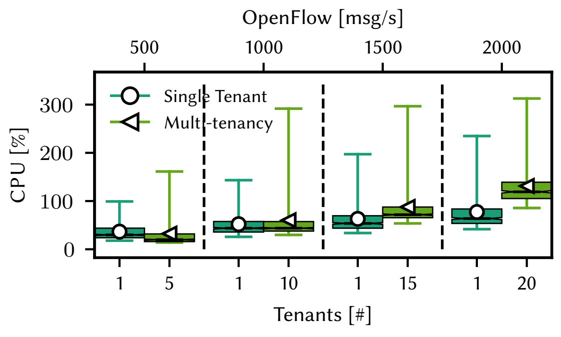


(a) Hypervisor: FV Latency.

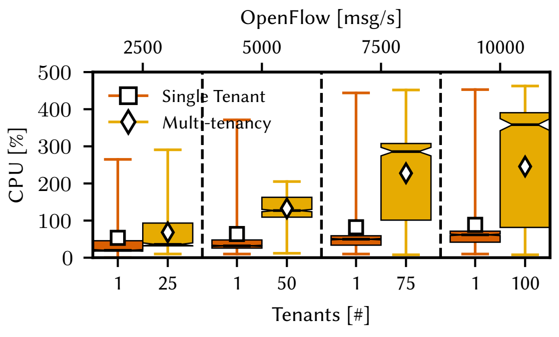


(b) Hypervisor: OVX Latency.

Figure 3.8: Comparison between the impact of a single tenant and multi-tenancy on the control plane latency provided by FV and OVX. The message type is OFPT_FLOW_MOD. Note the different message rates for FV and OVX. The number of tenants is provided on the lower x-axis, while the message rate is given on the axis on top of the figures. The dashed lines indicate the different measurement setups. Multi-tenancy is increasing the control plane latency in contrast to the single-tenant setup.



(a) Hypervisor: FV CPU.



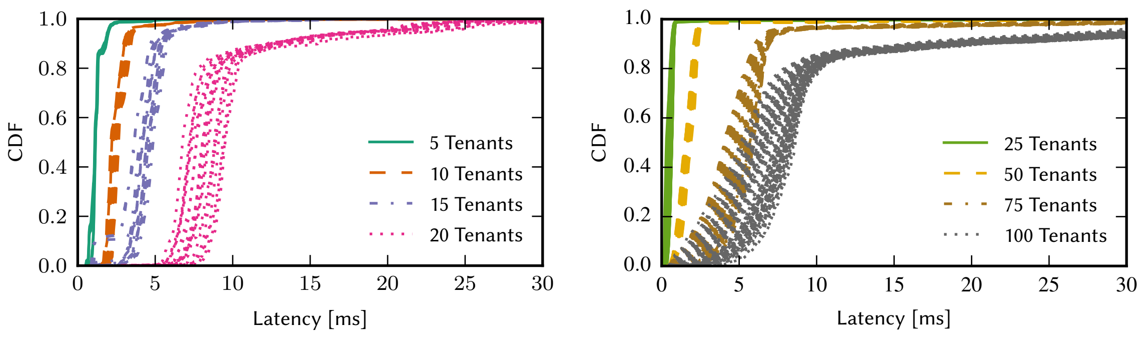
(b) Hypervisor: OVX CPU.

Figure 3.9: Comparison between the impact of a single tenant and multi-tenancy on CPU consumption by FV and OVX. The message type is OFPT_FLOW_MOD. Note the different message rates for FV and OVX. The number of tenants is provided on the lower x-axis, while the message rate is given on the axis on top of the figures. The dashed lines indicate the different measurement setups.

a given traffic rate (in total 200) should be equal to a setup with one tenant having the same total traffic rate (the sum of the rates of the two tenants, e.g., 200) - the control latencies should be in the same range for both single and multi-tenancy setups. Hence, increasing the number of tenants also increases the workload on the hypervisor under test. TCP_ND is enabled so that the controllers do not add waiting times. In case of multi-tenancy, each tenant generates 100 OFPT_FLOW_MOD messages per second.

Fig. 3.8a shows the impact of multiple tenants on FV's performance. With an increasing number of tenants, multiple tenants generally lead to higher latency when compared to the single-tenant setup. For instance, with 10 tenants producing a total rate of 1×10^3 messages, the control latency is 3 ms higher than the control latency of a single tenant with 1 ms. The measured CPU consumption confirms that FV has to accomplish extra work for multiple tenants, which can particularly be seen for 15 and more tenants in Fig. 3.9a. This justifies the obligation of conducting multi-tenant measurements when quantifying the performance of hypervisors.

Due to load constraints, FV cannot support more than 20 tenants: the implementation of how



(a) Hypervisor: FV Latency. (b) Hypervisor: OVX Latency.

Figure 3.10: Cumulative Distribution Functions (CDFs) of latency values of tenants for four tenant settings. The message type is OFPT_FLOW_MOD. CDFs show the latency distribution per tenant per varying setups for FV and OVX. Note the different number of tenants for each setup. The higher the number of tenants, the higher the distribution of latencies among individual tenants — an indicator for unfair latency distribution.

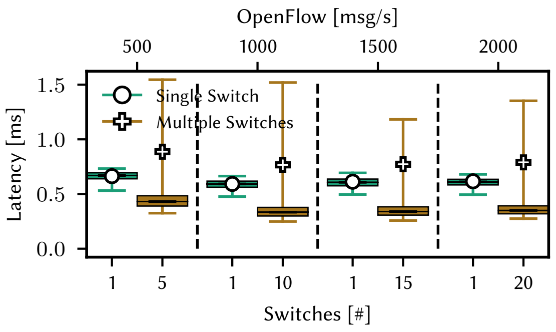
FV processes OFPT_FLOW_MOD messages leads to a switch overload; thus, the performance becomes even unpredictable. In detail, FV implements a protection method that replicates the OFPT_FLOW_MOD message of one tenant to all other tenants, using a ”forward to controller” action; however, this mechanism puts high loads on the control channel between FV and the switch, as well as on the switch CPU.

OVX shows similar effects when increasing the number of tenants, as seen in Fig. 3.8b. OVX can support up to 100 tenants given the setup. Because OVX is multi-threaded, it can efficiently utilize more than 1 CPU core. Fig. 3.8b shows again a significant increase in control plane latency for multiple tenants; with 100 tenants and 10×10^3 messages per second, the control latency increases by more than 10 times up to 10 ms when compared to the single-tenant setup. OVX also consumes drastically more CPU with an average of 380 % with 100 tenants compared to 70 % in case of a single tenant; a five-fold increase, as shown in Fig. 3.9b. Note also that OVX is already overloaded in these settings; some measurement runs are not usable as OVX even shows failures — OVX cannot handle more than 75 tenants reliably. All results represented here are for runs where no failure could be observed at least for the 30 sec duration of the experiments.

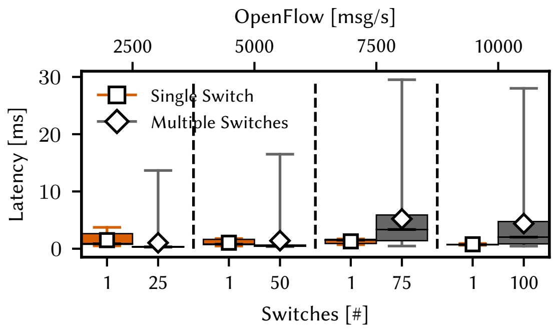
Similar performance and behavior are observed for other OF message types, e.g., OFPT_-PACKET_IN messages originating from the switches.

Is the control plane latency distributed fairly across the tenants? Even if tenants are sending messages of the same type at the same rate, side-effects due to resource interference might lead to unfair situations. The control plane latency per tenant is now used as an indicator for fairness. The fairness among 5 to 20 tenants is compared for FV and among 25 to 100 for OVX.

Fig. 3.10 shows the latency distribution among all tenant setups for FV and OVX. For 5 or 10 tenants, there is no clear difference among all tenants: both FV and OVX do not clearly advantage one tenant. This holds generally for all conducted runs. However, when comparing 5 tenants with 10 or 20 tenants, it can be noted that the variation of latency values increases with the number of tenants independent of the hypervisor. The tenants can even perceive latency values that differ on 4 ms on

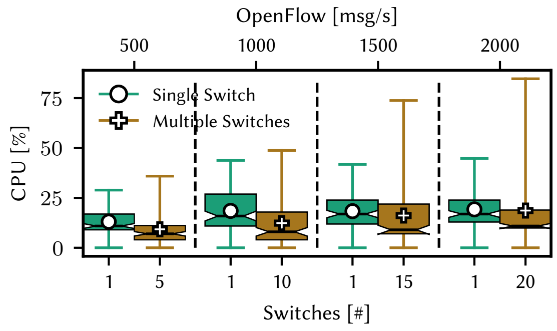


(a) Hypervisor: FV Latency.

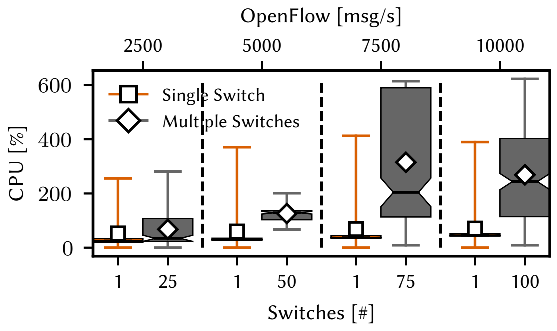


(b) Hypervisor: OVX Latency.

Figure 3.11: Comparison of the control plane latency provided by FV and OVX between single-switch and multi-switch setups. Note the different number of switches for FV and OVX. The number of switches is provided on the lower x-axis, while the message rates are given on the axis on top of the figures. The dashed lines indicate the different measurement setups.



(a) Hypervisor: FV CPU.



(b) Hypervisor: OVX CPU.

Figure 3.12: Comparison of the CPU consumption by FV and OVX between single-switch and multi-switch setups. Note the different number of switches for FV and OVX. The number of switches is provided on the lower x-axis, while the message rates are given on the axis on top of the figures. The dashed lines indicate the different measurement setups.

average; a clear unfairness when considering that the absolute values are not higher than 10 ms for 90 % of all values. Note further that an increasing number of tenants even drastically worsens the absolute latency values. For both hypervisors, 5 % of all latency values are larger than 15 ms already. In summary, the performance in a multi-tenant setup becomes less predictable with both hypervisors - it is harder to provide absolute latency guarantees.

The observed latency values can be used to determine upper bounds for the control plane latency that can be guaranteed in a multi-tenant setup. Beyond any doubt for small traffic rates, the hypervisors share the resources fairly among the tenants based on their portion of network control traffic. All tenants are equally affected given the number of tenants, the message type, and the control plane traffic. However, increasing the number of tenants leads to higher and more variable latency performance: more tenants lead to more resource interference - the hypervisors do not fairly share the resources anymore.

How does the control latency depend on the number of switches? This section quantifies the ability of hypervisors to operate multiple switches. Such information is important to argue about a hypervisor's scalability performance. Intuitively, a hypervisor performance should scale with the number of switches (switch connections): the performance of tenants should not be affected by the number of switches. For this purpose, *perfbenchDP* emulates switches and sends OFPT_PACKET_IN messages as traffic workload. FV is benchmarked for 5 to 20 switches; OVX is benchmarked for 25:100 switches. Each switch generates 100 OFPT_PACKET_IN messages per second; therefore, the workload increases with the number of switches. The performance of the multi-switch setup is compared to the single-switch setup. The setup consists of one controller with TCP_ND enabled to eliminate potential cross effects due to multiple controllers in combination with artificial waiting times.

Fig. 3.11 and Fig. 3.12 show the effect of the number of switches on the hypervisors' performance. Controlling many switches does not severely affect FV's CPU consumption and its provided control plane latency on average (Fig. 3.11a and Fig. 3.12a). However, more switches introduce a higher performance variability: while the average latency and the average CPU consumption are only marginally affected, the performance metrics generally show a higher variance. For instance, when comparing 1 with 20 switches (i.e., a total message rate of 2×10^3 msg/s), 20 switches increase the average control latency by 200 µs only; however, maximum latency values that are two times higher can be observed. However, the latency overhead due to multiple switches is negligible when looking at the absolute values.

In contrast to FV, OVX noticeably affects the control plane latency: the average latency and the maximum latency are doubled for 75 and 100. This is also shown by the CPU consumption in Fig. 3.12b: OVX consumes more CPU due its multi-threaded implementation to handle multiple switches; the CPU actually increases from 44 % (single-switch setup) to 395 % (multi-switch setup). This means that OVX exploits multiple threads to share its resources (CPU) among switches. Note also that OVX is already overloaded in these settings; some measurement runs are not usable as OVX even shows failures. All results represented here are for runs where no failure could be observed at least for the 30 sec duration of the experiments.

In summary, the number of switches and the number of tenants affect hypervisor performance; in particular the number of tenant controllers shows a higher impact than the number of switches. This suggests that handling the abstraction of virtual networks (and the involved context switching of the hypervisors) has a significantly higher impact than handling switch abstractions. As an identified example, adding tenants increases the workload of FV quadratically with the number of tenants, whereas when adding a switch the relation between number of switches and workload is linear.

3.5 Network Virtualization Layer Architecture Towards Predictable Control Plane Performance: *HyperFlex*

As the previous measurements reveal, hypervisor implementations offer significantly different performance for the same networking tasks, i.e., abstracting and isolating networking resources. Accordingly, operators must choose between different implementations and their performance trade-offs when planning to virtualize SDN networks. In case operators demand adaptation, they should not limit themselves by using a one-fits-all approach.

Generally speaking, the existing architectures cannot adapt towards the given use cases and requirements. For instance, they cannot change their operations from software (e.g., for low cost) to hardware (e.g., for low latency), even if the current setup would allow it. Hypervisor proposals further do no elaborate on their deployment in different network scenarios; rather they invoke the design of special hypervisors for special setups, which is per se limiting their utility for different scenarios. Accordingly, we introduce HyperFlex: an architecture for SDN networks that decomposes the virtualization layer into functions that are needed for the virtualization of SDN networks.

HyperFlex's concept relies on the idea to simultaneously operate network functions either in software or in hardware, by making use of the whole flexibility as provided by today's communication network infrastructures. The number and the operation of the software and hardware functions are determined by the current requirements of the virtual network tenants. For instance, tenants that require a precise control plane latency might be hosted on hardware functions, which satisfy strict processing time constraints. Functions for virtual networks that demand a flexible adaptation can be hosted on software functions: nevertheless software functions might be prone to higher performance variability.

However, to realize such flexible virtualization architecture, the operation and execution of vSDN needs to be highly predictable: network functions should show performance values that fall into known ranges. Accordingly, mechanisms are needed that (1) provide a predictable performance and that (2) provide a high flexibility in terms of resource operation. In this section, we present virtualization layer functionalities that increase the predictability and adaptability of virtualization layers. We particularly focus on the design of the control plane part of the virtualization layer.

3.5.1 Architecture Overview

This section briefly introduces HyperFlex's main architecture concept. Later on it discusses the main advantages of HyperFlex's architecture. According to the given classification introduced in Sec. 3.1.2, HyperFlex is the first hypervisor network architecture that operates on general computing platforms and can make use of special-purpose and general-purpose networking hardware.

Concept. HyperFlex [BBK15] realizes the hypervisor layer via the orchestration of multiple heterogeneously implemented virtualization functions: the virtualization layer has been decomposed into functions required to virtualize SDN networks. The functions themselves can be realized through heterogeneous implementations: a function is either realized in software or in hardware, or it can be realized in a distributed fashion, i.e., via both software and hardware. Besides, any entity implementing one functionality or more, e.g., existing hypervisors as FV or OVX, can be integrated in order to accomplish the virtualization tasks.

HyperFlex operates a mix of all possible realizations of functions, which means also a mix of their implementations. As control plane traffic of tenants may need to pass through different functions (isolation and abstraction), HyperFlex operates and interconnects the functions accordingly, which as a whole realizes the virtualization layer.

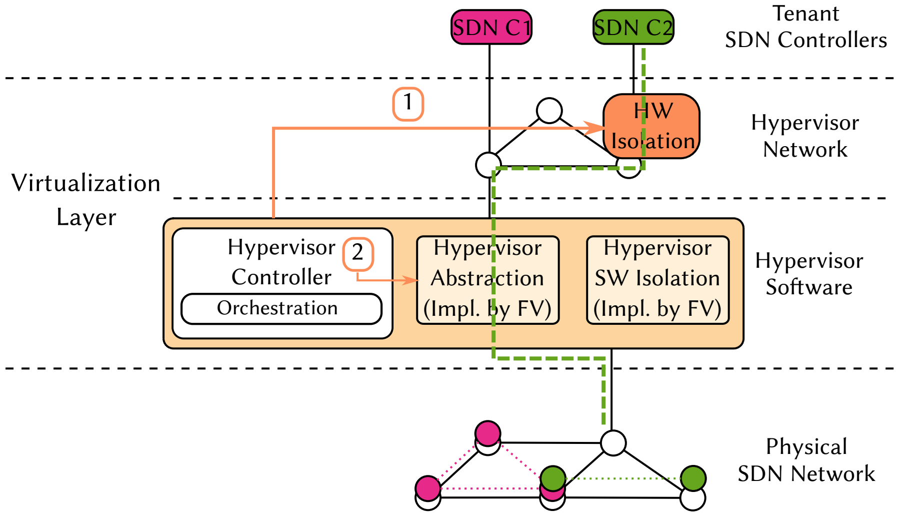


Figure 3.13: HyperFlex architecture: the *virtualization layer* is built of tow main components: the *hypervisor network* and the *hypervisor software*. The *hypervisor network* nodes host the hypervisor functions realized in hardware. The *hypervisor software* consists of the *hypervisor controller* and the software-based functions (e.g., abstraction and isolation). The *hypervisor controller* controls and orchestrates all functions, i.e., hardware functions (see 1) and software functions (see 2). The example illustrates how the traffic of the second tenant passes through the hardware function and then through the abstraction function towards the physical SDN network.

Components. Fig. 3.13 shows an overview of the architecture. Two main components are shown: the *hypervisor network* and the *hypervisor software*. The hypervisor network connects the tenants' SDN controllers with the hypervisor software, e.g., FV realizing abstraction and software isolation. FV's abstraction function is responsible for controlling the physical SDN network; it establishes the control connections with the SDN network. The network nodes of the hypervisor network host the hardware-based virtualization functions. The *hypervisor controller* manages and controls the hypervisor network. It also configures the hardware functions. Any computing platform, like data centers, can host the hypervisor SDN controller (*hypervisor controller*) and software functions. To summarize, HyperFlex differentiates the following two function types:

- *Hypervisor Software Functions.* These are the virtualization functions implemented in software and hosted on computing platforms. In its current version, HyperFlex can deploy here any existing software-based hypervisor, e.g., FV: in its current realization, hypervisors also establish the connections with the physical SDN network. Additional functional granularity can be achieved by using FV or OVX in a hierarchical manner [SGY+09; SNS+10; ADG+14; ALS14].
- *Hypervisor Hardware Functions.* Hypervisor hardware functions consist of two components: the logic residing in software and the processing realized via hardware. An SDN controller manages the hardware parts of these functions. The functions can only be realized by hardware components offering the targeted functionality, e.g., isolation through hardware schedulers and queues.

As an example, Fig. 3.13 shows how the control traffic of the second green tenant passes through the hypervisor network. Here, the control traffic is processed by a hardware-based isolation function.

The software-based abstraction functions then processes the control traffic: e.g., it sets the correct identifier of the tenant. Then, it forwards the control traffic towards the physical SDN network.

Design goal advantages. HyperFlex's design goals are higher flexibility, scalability and predictability. In the following, it is discussed how the different design choices contribute towards these targets.

- *Software versus Hardware Functions.* Making use of the realization of functions either in software, hardware, or both software and hardware, increases the flexibility to realize the virtualization layer. For instance, the realization in software allows to offload the functionality to cloud infrastructures. This utilizes actually the advantages of using cloud resources: flexible requesting of resources on demand. Furthermore, software implementations benefit from their independence of the underlying hardware. Hardware implementations require special capabilities from the hardware, e.g., traffic shapers or limiters for isolation. In case hardware provides the capabilities, it is expected that hardware processing provides higher predictability, i.e., less variations. However, the use of hardware is usually limited by the available resources when compared to software, e.g., fast matching needs expensive TCAM space.
- *Flexible Function Placement.* HyperFlex makes it possible to flexibly deploy software functions, i.e., they can be placed and run on commodity servers, or to use hardware, i.e., functions can be realized via the available capabilities of the physical networking hardware. Hence, HyperFlex potentially increases placement possibilities. The advantages are manifold; avoiding a single point of failure due to operations at multiple locations; lower control plane latency due to deploying functions closer to tenants.
- *Improved Resource Utilization.* Decomposing and distributing the virtualization layer into functions provides better demand adaptation capabilities. For instance, in case of high load scenarios, processing may be offloaded from hardware to software functions or vice versa. A hypervisor architecture realized as a single centralized entity may always demand a high resource over-provisioning, as the architecture is limited by the scalability of the hosting platform.
- *Control Plane Isolation.* Differentiating between tenant control plane traffic introduces better control plane isolation capabilities. Isolating control plane traffic leads to more predictable network operation: e.g., the traffic of a malicious controller is hindered to affect the operation of non-malicious controllers of other tenants. Only by isolation, operators can guarantee correct control plane operation with acceptable latency, which eventually results even in a more predictable behavior of the data plane.

3.5.2 Virtualization Layer Control Plane Isolation

This section introduces HyperFlex's control plane isolation concept towards predictable and guaranteed control plane operations for tenants.

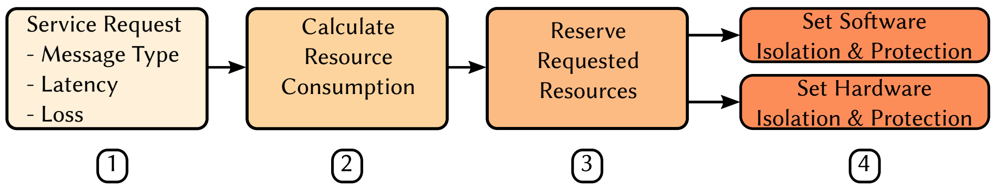

Service Request
- Message Type
- Latency
- Loss
Calculate Resource Consumption
Reserve Requested Resources
Set Software Isolation & Protection
Set Hardware Isolation & Protection
1
2
3
4

Figure 3.14: Resource reservation pipeline: steps from requesting to setting isolation and protection of resources. First: service requests specified with OF message type, latency and loss; second: calculation of resource consumption based on request; third: needed resources (CPU and network data rate) are reserved; fourth: hardware and software isolation functions are configured for isolation and protection.

3.5.2.1 Concept

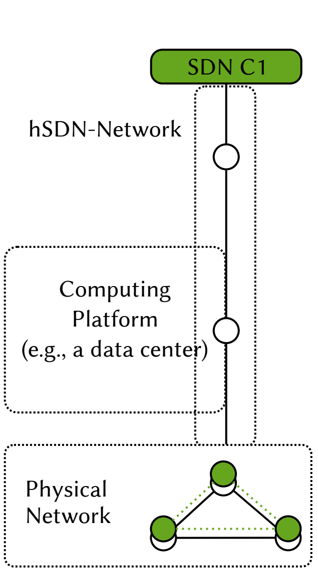
Two resources might affect the control plane latency of tenants: the available data rate on the hypervisor network and the processing capabilities of the virtualization functions. In contrast to the impact of the message rate on the CPU consumption, the needed data rate for transmitting messages is rather small. For instance, even sending OFPT_FLOW_MOD messages at a rate of 50×10^3 msg/s results in a data rate of 4 Mbit/s only; prioritizing and reserving such data rate comes at a low cost. As the measurements, however, revealed, even with multiple tenants sending at low data rates, the CPUs of hypervisors can be easily overutilized. As a consequence, HyperFlex mainly focuses on isolation techniques to prevent CPUs of platforms hosting the virtualization functions from over-utilization. It targets at the implementation of two isolation techniques: a software-based and a hardware-based isolation.

Figure 3.14 illustrates the overall configuration process for control plane isolation. Firstly, service requests are defined in terms of OF message types, rates, latency and loss metrics (maximum, average, etc.). Secondly, based on the requests, resource consumptions are calculated; resources include the CPUs of the virtualization functions as well as the resources of the hypervisor network and the processing resources of the networking elements. Thirdly, the virtualization layer then reserves the needed resources. Finally, software and/or hardware isolation functions are configured and deployed among the infrastructure.

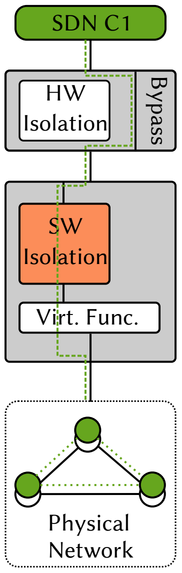
The CPUs of the network nodes (physical SDN switches) and the computing platforms of the infrastructure can mainly affect the processing time of OF messages. Highly-utilized CPUs might lead to longer message processing times. As many tenants are sharing these CPUs, even a single overloaded CPU may degrade the performance of many vSDNs. The source of over-utilization can already be a single vSDN only. Accordingly, all involved CPUs need to be sliced and allocated for all tenants for isolation purpose.

In order to slice CPUs for tenants, benchmarks are needed which quantify the relationships between CPU consumptions and control plane message rates of different control plane message types. With benchmarks (as demonstrated in Sec.3.4), functions can be modeled that map requested OF message rates to CPU consumptions of network hypervisors. Based on identified relationships, isolation functions are then configured to support requested services in terms of OF message type, loss and latency.

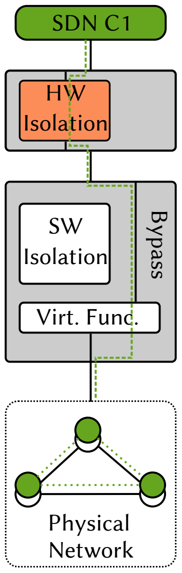
Figure 3.15 provides an overview about an exemplary physical network underlying the isolation



(a) Physical view of network.



(b) Function chain with software (SW) isolation.



(c) Function chain with hardware (HW) isolation.

Figure 3.15: HyperFlex's deployment of isolation functions: physical network underlying chains, function chain with hardware versus function chain with software.

function chains (Fig. 3.15a), an exemplary software function chain (Fig. 3.15b) and a hardware function chain setup (Fig. 3.15c). Fig. 3.15a shows the hypervisor network and the physical SDN network. The hypervisor network consists of a network node; the software functions are operated on a computing platform, as provided by data centers. The network node hosts the hardware isolation function (HW Isolation) and the data center hosts the software isolation function (SW Isolation).

Software Isolation

Figure 3.15b shows a realization of the software function chain. While the control plane traffic bypasses the hardware isolation, it passes the software isolation. The software isolation function operates on the application layer (Open Systems Interconnection (OSI) Layer 4); it drops OF messages that exceed a prescribed vSDN message rate per second. Dropping OF messages is intended to lower CPU consumptions of hypervisor functions and switches: hence, it should either decrease interference or protect CPUs from over-utilization.

The reasons for savings are expected to be twofold: (1) hypervisor software functions, which succeed the isolation, process less application layer messages received from the tenants and (2) they forward less messages towards the switches, which involves again both message (application layer) and packet (network layer) processing. As a drawback, the tenant controllers have to compensate for the lost OF messages: e.g., controllers should not wait endless for replies of request messages. By sending OF error messages, hypervisors can also implement active notification mechanisms; however, such mechanisms might again demand resources, e.g., CPU. The measurement studies in Section 3.5.2.2 reveal that the benefits of the software isolation depend on several aspects like the OF message type: e.g., dropping synchronous messages, i.e., requests, avoids the overhead of processing the replies.

Configuration. The configuration parameter of the software isolation is the number of total OF messages allowed per second. The software isolation function only counts the perceived OF messages per second; within a second, a tenant can send messages with any inter-arrival time and at different burst-sizes as long as the messages do not exceed the limit per second. The configuration parameter of the software isolation depends on the relationship between OF message type rates and the CPU consumption of the computing platform. The CPU consumption involves the operation of the isolation function and the operation of all other hypervisor functions.

Hardware Isolation

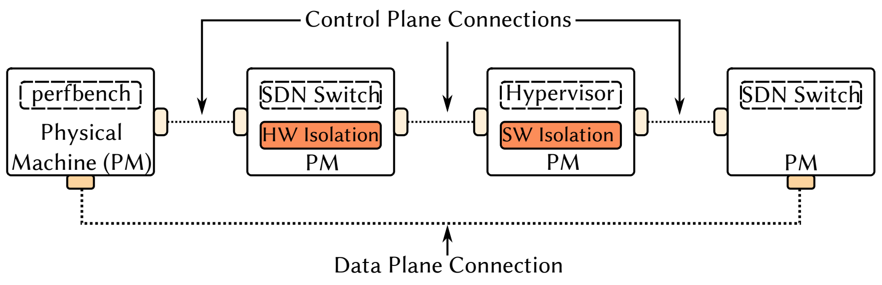
Figure 3.15c illustrates how HyperFlex realizes the hardware isolation; the control plane traffic passes through the hardware isolation function while it bypasses the software isolation. The hardware isolation function operates on the OSI layers 2–4, while it polices (limits) the forwarded vSDN control messages.

More specifically, a hardware solution polices (limits) OF control traffic on the network layer through shapers (egress) or policers (ingress) of general-purpose or special-purpose networking hardware. However, hardware isolation does not specifically drop OF messages, which are application layer messages: current OF switches cannot match and drop application layer messages. Besides, OF can use encrypted data transmissions, which makes it impossible to match on specific OF headers in the content of the TCP packets. Rather, hardware solutions drop whole network packets (containing TCP segments) based on matches up to layer 4 (transport layer). Dropping TCP segments causes TCP packet retransmissions. As a result, the backlogs of buffers increase; larger buffers lead to higher control plane latencies for tenants.

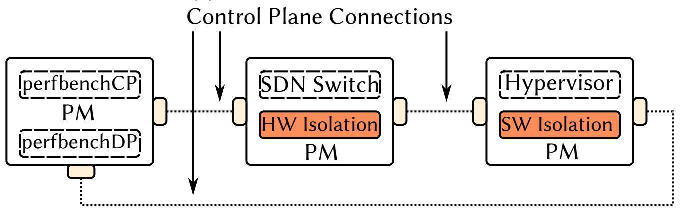
Configuration. The configuration parameter of the hardware isolation is the data rate per second (e.g., kbit/s) that is needed to forward control plane messages at stated inter-arrival times (e.g., 1 ms). For configuring the data rate, an additional mapping between OF message rate and network data rate is needed, which needs also to consider the inter-arrival times of messages. Hence, the mapping has to consider the individual sizes of OF packet headers and the headers of the used transmission protocol (e.g., TCP) when calculating and setting the data rate.

For instance, let us assume that a tenant requests to send 1 OF message, e.g., OFPT_FEATURES-_REQUEST having a size of 8 Byte, every 500 ms. Taking a TCP header of 66 Bytes into account, this demands a policer to accept a sending rate of $2 \cdot (66 + 8) = 148$ Bytes per second (66 Byte including TCP, IP and Ethernet header). However, if the tenant now sends only every second, he can send $(148 - 66)/8 = 10.25 \approx 10$ messages per second: the tenant sends more messages at a higher aggregation rate; however, he has to accept a longer waiting time between sending messages. Hence, the configured data rate determines the OF message throughput in a given time interval. Note further that the exemplary calculation provides only a lower bound of minimum data rate that needs to be reserved. Experiments, which are not shown for brevity, revealed that always a data rate headroom needs to be reserved to achieve the desired message rate with stated inter-arrival times. More details on configuring the hardware isolation function are provided in [BBK15; BBL+15].

As we are interested in the general effects of isolation, we omit more detailed explanations of the configurations of hardware and software isolation. In particular the configurations of the hardware



(a) Without Switch Emulation.



(b) With Switch Emulation.

Figure 3.16: Two isolation measurement setups: without switch emulation (top) and with switch emulation (bottom). When *perfbench* emulates the switch, it connects actively to the hypervisor under test.

and software isolation offer many further optimization opportunities, like using different schedulers on switches or advanced shaping mechanisms [Lar].

3.5.2.2 Measurement Evaluation of Control Plane Isolation

While the preliminary study of HyperFlex [BBK15] proofs its potential to isolate performance, this thesis extends the preliminary study: it focuses on more message types and different utilization scenarios. The following questions are investigated:

- Can both isolation mechanisms always avoid resource interference?
- Does isolation depend on the message type?
- What is the cost of isolation?

Figure 3.16 gives an overview of the measurement setup. The measurement setup introduced in Section 3.4.2.1 is extended by an additional switch between the hypervisor under test and the SDN controllers. The additional PC runs an instance of the OvS software switch for realizing the hardware isolation function.

Implementation details. The same measurement infrastructure as presented in Sec. 3.4.2.1 is used: the same PCs and interconnections. HyperFlex's isolation capabilities are studied for FV, as only FV implements a software isolation function. Linux traffic control is used to demonstrate the hardware isolation [Lar].

Procedure. The benchmarks rely also on the same measurement procedure as introduced in Section 3.4.2.1. Either a single or two tenants send OF messages of one type at the same or different

Table 3.4: Isolation measurement configurations. Since TCP_ND is alway set to true, it is omitted from the table. A hardware isolation of 100 kB/s corresponds to a message rate of $\approx 2.5 \times 10^3$ msg/s. OF message inter-arrival times are 1 ms. In the first two settings, Tenant-1 is increasing its message rate, which is in accordance with its SLA. Tenant-2 is sending always 5×10^3 msg/s with 1 ms inter-arrival times, which is not in accordance with its SLA. In the third setting, the needed headroom of software isolation for one tenant is studied. The fourth setting demonstrates the infrastructure protection capabilities: Tenant-2 is always sending 4.5×10^3 msg/s while the software isolation rate is increased from 1×10^3 msg/s to 7×10^3 msg/s.

Iso.	OF Message Type	Tot. Ten.	Tenant	Msg. per sec.	Iso. Conf.
HW	OFPT_-FEATURES_-REQUEST	2	1	2.5×10^3-12.5×10^3	-
			2	5×10^3	100 kB/s $\approx$ 2.5 messages every 1 ms
SW	OFPT_-FEATURES_-REQUEST	2	1	3×10^3-6×10^3	-
			2	5×10^3	2.5×10^3 msg/s
SW	OFPT_-PACKET_OUT	1	1	20×10^3 - 50×10^3	20×10^3
SW	OFMP_PORT_-STATS	2	1	4×10^3	-
			2	4.5×10^3	1×10^3 - 7×10^3

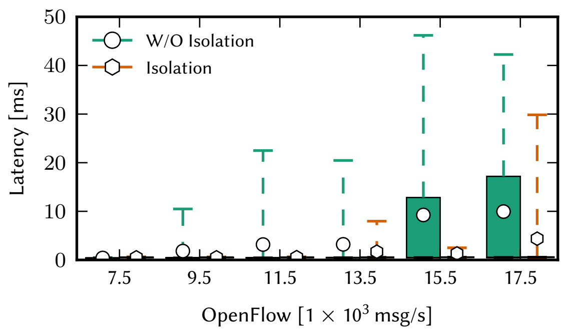
rates. The inter-arrival times of messages is always 1 ms. The messages are processed either by a hardware isolation or by software isolation.

Data. All results represent at least 3 runs per message rate and type. The durations of all runs vary between 30 s and 60 s. The analysis relies on the following performance indicators: the control plane latency and the CPU consumption of the virtualization layer and an OvS switch. All measurement setups and configurations are summarized in Table 3.4.

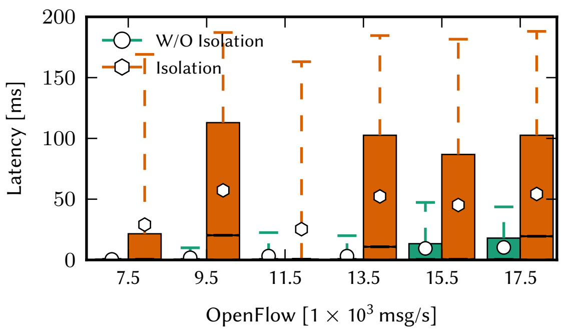
3.5.2.3 Measurement Results

In a first setup, the general capabilities of the isolation functions to guarantee network performance for tenants are quantified. The used message type is OFPT_FEATURES_REQUEST; the message type is a good representative for request/reply messages, as it benchmarks the whole processing cycle - virtualization layer, switch, virtualization layer. Two tenants are operating in this setup: Tenant-1 and Tenant-2. Tenant-1 is constantly increasing its message rate; this is in conformity with its service request - the tenant should not be affected by interference. While Tenant-1 increases its message rate, Tenant-2 will always send a constant but exceeding number of messages; this is not in accordance with its service request - isolation should prevent the tenant from exceeding its rate. Exceeding the rate, however, will increase the CPU load of the hypervisor, which might lead to interference. The CPU of the hypervisor process is limited to 15 % to better illustrate the overload impact.

Can hardware isolation protect virtualization layer resources? Figure 3.17 shows the results of the hardware isolation for both tenants. In this setup, the hardware isolation is configured to



(a) Latency of Tenant-1 (normal behavior). Tenant-1 is increasing its message rate, which is in agreement with its service request. x-axis gives the total message rate.



(b) Latency of Tenant-2 (not conform). Tenant-2 is always sending 5×10^3 msg/s, which is **not** in agreement with its service request. Network isolation rate: 100 kB corresponding to 2.5×10^3 msg/s with 1 ms inter-arrival times.

Figure 3.17: Latency boxplots with and without **hardware** isolation for two tenants: Tenant-1 is behaving normal whereas Tenant-2 is exceeding its agreed message rate. Comparison between the tenants' control plane latencies with hardware isolation and without isolation (W/O Isolation). Fig. 3.17a shows the normal behaving tenant. Fig. 3.17b shows the tenant that exceeds the rate. OF message type: OFPT_FEATURES_REQUEST. The x-axis shows always the total message rate per experiment. For instance, 7.5×10^3 msg/s contain 2.5×10^3 msg/s of Tenant-1 and 5×10^3 msg/s of Tenant-2. Isolation reduces interference; hence, latency is reduced for Tenant-1.

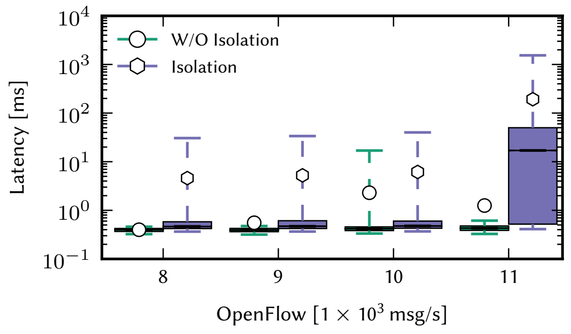
allow a message rate of roughly 2.5×10^3 msg/s. Without isolation, the tenants' operations interfere. This can be seen for Tenant-1 in Figure 3.17a: without isolation (W/O Isolation), the control plane latency is affected by the control traffic of Tenant-2. When looking at the total message rates 9.5×10^3 to 17.5×10^3, the control plane latency is significantly increasing in comparison to the isolation setup. For instance, the average control plane latency is ten times higher for a total traffic rate of 15.5×10^3 msg/s. With isolation, the function protects the performance of Tenant-1; it keeps the working latency low for traffic rates between 7.5×10^3 to 11.5×10^3.

The hardware isolation increases the maximum message throughput of FV. It forces the exceeding tenant controller to aggregate its OF messages; aggregating messages leads to less network packets, which decreases CPU utilization. As a result, FV can process more OF messages, which is also indicated by the low latency for Tenant-1 in Figure 3.17a for 15.5×10^3 msg/s. Although FV operates under high utilization, it processes more messages with low latency for Tenant-1; however, this comes at the cost of higher control plane latency for Tenant-2.

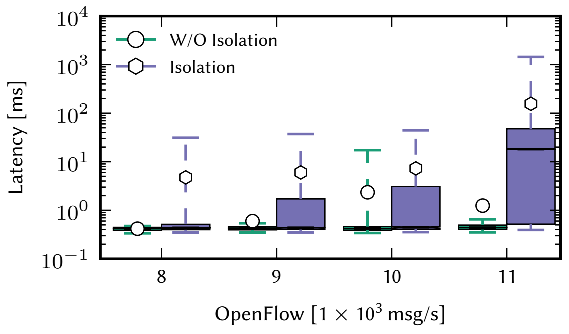
Figure 3.17b shows the control plane latency of the exceeding tenant (Tenant-2): the latency values with isolation are all significantly higher than without isolation. The reasons are that when Tenant-2 exceeds its message rate, TCP retransmits the dropped packets. Because its data rate budget of control channel resources is already spent, even less data rate is available for the whole transmission process, which results in significant higher latencies.

We conclude that hardware isolation can protect hypervisor CPUs; hence, it isolates tenants' control plane traffics and reduces interference.

Does software isolation depend on the utilization and message type? Figure 3.18 reports on the results of the software isolation function for OFPT_FEATURES_REQUEST messages. For



(a) Latency of Tenant-1 (normal behavior). Tenant-1 is increasing its message rate, which is in agreement with its service request. x-axis gives the total message rate.



(b) Latency of Tenant-2 (not conform). Tenant-2 is always sending 5×10^3 msg/s, which is **not** in agreement with its service request. Software isolation rate: 2.5×10^3 msg/s.

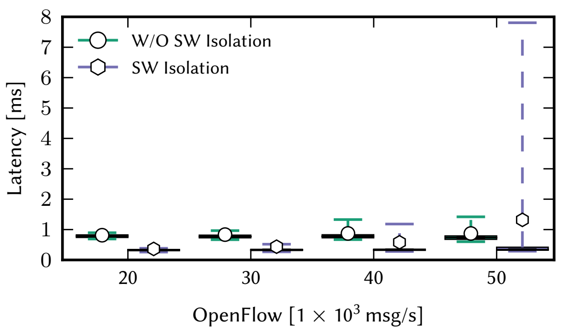
Figure 3.18: Latency boxplots with and without **software** isolation for two tenants: Tenant-1 is behaving normal whereas Tenant-2 is exceeding its agreed message rate. Comparison between a tenant's control plane latency with software isolation and without (W/O Isolation). Fig. 3.18a shows the tenant that behaves normally. Fig. 3.18b shows the tenant that exceeds its rate. OF message type: OFPT_FEATURES_REQUEST. The x-axis shows always the total message rate per experiment. For instance, a total rate of 8×10^3 msg/s contains 3×10^3 msg/s of Tenant-1 and 5×10^3 msg/s of Tenant-2.

Tenant-1, as shown in Figure 3.18a, the software isolation increases the control plane latency on average and also the maximum values. There is one main reason for this behavior: while FV is already operating under high CPU consumption for the investigated message rates, software isolation additionally consumes CPU resources. Accordingly, the software isolation cannot protect and isolate the control plane latency of Tenant-1. A message rate of 11×10^3 msg/s clearly manifests this observation: the control plane latency is significantly worse than without an active isolation.

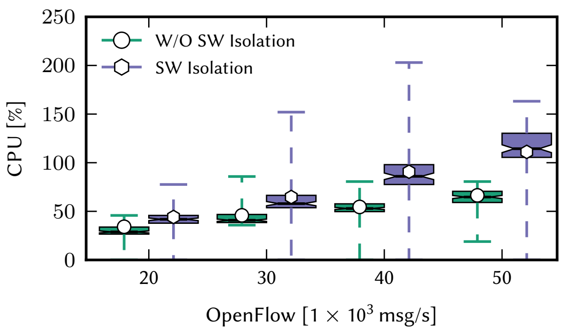
The same observations hold for the second tenant (Fig. 3.18b): the trends of the latency values are similar to the ones of Tenant-1. Among all message rates, software isolation again increases the control plane latency: a counterintuitive observation. When elaborating the implementation of the software isolation in depth, the effect can be explained: before dropping application layer messages, FV still processes all network packets. This packet processing, however, is already the bottleneck in the whole process. Thus, as isolation adds processing, even less CPU resources are available for the network packet processing, which increases the latency.

To summarize, when FV is already highly loaded, software isolation cannot decrease its CPU consumption; therefore, it does not avoid interference among tenants in high load scenarios.

Does the efficiency of software isolation depend on the current utilization? Although the software isolation cannot protect CPUs of hypervisors or guarantee latency in an over-utilization scenario, it can improve, i.e., guarantee, latency when the CPU is not overloaded. Figure 3.19 shows the results: one tenant sends OFPT_PACKET_OUT messages at constant rates ranging from 20×10^3 msg/s to 50×10^3 msg/s. Note again that a rate of 50×10^3 msg/s slightly overloads FV, i.e., the available CPU resources. Figure 3.19a depicts that software isolation now decreases the control plane latency, in contrast to the previous measurements with OFPT_FEATURES_REQUEST. Only for 50×10^3 msg/s, the upper bound of the control plane latency increases due to the over-utilization



(a) Control plane latency for two tenants when increasing software rate limit of exceeding tenant (Tenant-2).



(b) CPU consumption with and without activated software isolation. Software isolation demands a CPU headroom.

Figure 3.19: Software isolation-based infrastructure overload protection - OpenFlow message: OFPT_-PACKET_OUT. Latency boxplots over message rate (Fig. 3.19a) and CPU boxplots over message rates (Fig. 3.19b). Software isolation for OFPT_PACKET_OUT protects control plane latency and decreases CPU consumption. FV is overloaded for 5×10^4 msg/s.

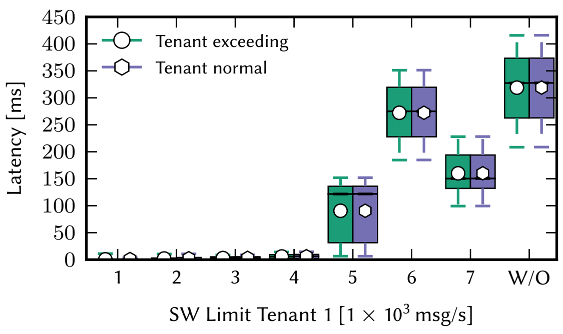


Figure 3.20: Software isolation protecting a switch. Latency boxplots over software isolation message rate with and without (W/O) isolation over total message rate. Between rates from 1×10^3 msg/s-4×10^3 msg/s, software isolation protects the switch CPU. The latency is low and stable. At 5×10^3 msg/s, the software isolation message rate is too high. The switch becomes overloaded as all messages of the second tenant are forwarded towards the switch — the latencies of the tenants are up to 400 times higher than with isolation.

of the CPU. Now dropping messages has a positive effect on the latency: less messages are queued and processed efficiently.

Figure 3.19b reports on the CPU consumption; it illustrates that software isolation demands additional CPU resources. Further, an increasing message rate does not only increase the average CPU, it also leads to higher CPU variations. When configuring the software isolation, the additional CPU consumption needs to be considered: a headroom needs to be reserved.

Can software isolation protect the network nodes? While hardware isolation can generally protect FV and isolate control plane latency, and software isolation similar when FV is not overloaded, there is yet another benefit from software isolation: the protection of the infrastructure. Two tenants are involved in the next setup: one tenant sends OFMP_PORT_STATS at constant rates of 4×10^3 while the other tenant sends 4.8×10^3 msg/s. The software isolation rate for the second tenant is increasing. When the second tenant rate exceeds the isolation rate, i.e., when the dropping rate is higher than the sending rate (5×10^3 msg/s and more), the OvS switch will be overloaded. It can only handle up to 8×10^3 msg/s of OFMP_PORT_STATS messages.

Figure 3.20 reports on the results for the conducted measurements. Dropping rates between 1×10^3 msg/s and 4×10^3 msg/s efficiently isolate the control plane latency; the latency stays low. When the dropping rates exceed 5×10^3 msg/s, the switch is overloaded; hence, the switch processing cannot handle the message rates anymore and induces significant latency. This even affects both tenants: their latencies are both equally increased due the overload of the switch, although they send at different message rates. Furthermore, the latency values even become less predictable, which is shown by the variations of the boxplot values. Note also the different variations of the rates 5×10^3 msg/s to 7×10^3 msg/s. The reasons here are different aggregation levels of TCP: again, network packet aggregation is one main impact factor on CPU consumption. Without isolation (W/O), both tenants obtain the worst latency performance.

3.5.3 Virtualization Layer Adaptation

One main goal of HyperFlex is to better adapt to changing requirements, e.g., changing virtual network demands or occasional over-utilization of computing platforms. Therefore, a virtualization layer adaptation mechanism was introduced in [BBB+15] to realize a dynamic virtualization layer. The adaptation mechanism reassigns vSDN controllers to different hypervisors functions at runtime. For instance, controllers connected to overloaded virtualization functions or hypervisors can be reallocated to less-loaded entities, or hypervisor instances can even be turned off for energy saving.

OF 1.2 [Ope11b] introduced and defined the multiple controllers feature. Implementing the multiple controllers feature, switches can simultaneously connect to multiple SDN controllers. OF version 1.2 adds the OFPCR_ROLE_EQUAL mode. In this mode, all controllers connected to a switch can fully access and control the switch and its resources. Note, however, that OFPCR_ROLE_EQUAL requires controllers to synchronize. In this thesis, we analyze how the multiple controllers feature can be used for adaptation support and to reduce the control plane latency of vSDNs (see Chapter 4).

3.5.3.1 Concept

Fig. 3.21 gives an overview of HyperFlex's adaptation support. The overall goal of the concept is to avoid side-effects as much as possible and to work transparently towards the tenants. The tenant controllers should not need to interact with the virtualization layer to realize the adaptation support: the virtualization layer should work transparently in terms of control actions. Consequently, the virtualization layer operator should try to minimize side-effects such as increased latency values during the migration as much as possible.

Figure 3.21 illustrates that the hypervisor controller takes over three tasks: configuration of the hypervisor proxy - where to forward the control plane traffic (indicated by box 1); configuring the hypervisor functions (box 2); and configuring the data plane switches, i.e., instruct the switches to which functions they should connect (box 3). During the migration, the hypervisor controller triggers the migration of the control plane forwarding between the hypervisor proxy and the functions as well as between the functions and the physical network.

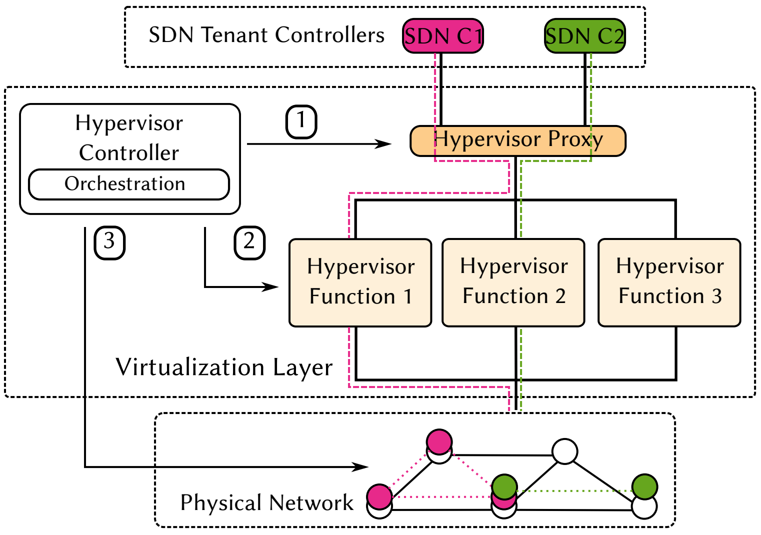


Figure 3.21: HyperFlex's adaptation support. Two controllers (SDN C1 and SDN C2) are connected to the hypervisor proxy. The hypervisor proxy is connected to three hypervisor functions. The hypervisor functions connect to the physical SDN network. The hypervisor controller is responsible for the control and orchestration of the hypervisor proxy (see box 1), the hypervisor functions (box 2), and for configuring the physical network (box 3).

Virtualization layer proxy. The virtualization layer is extended by the hypervisor proxy to realize the transparent working behavior. The hypervisor proxy acts as an interface between the hypervisor functions and the tenant controllers. It demultiplexes the incoming OF control messages towards the tenants. Without the hypervisor proxy, the tenant controllers would connect directly to the hypervisor functions. In case a hypervisor function would be adapted, e.g., turned off, the TCP connection on the controller side would be interrupted. The tenant controller would have to renegotiate a new TCP connection initiated by the virtualization layer. This would lead to an operational interruption on the controller side. For instance, current bytes (messages) in-flight would be lost and the TCP stack of the connection would have to be reestablished. The controller might even react with countermeasures, which, however, are erroneously triggered by a desired adaptation.

Introducing the proxy avoids such unpredictable network operation; the proxy is intended to always keep the operation with controllers up and running while the TCP connections between switches, functions and the proxy are changed at runtime. A special network protocol was developed to guarantee a seamless handover of the control traffic between the switches, functions, and the proxy. The protocol is explained in more detail in [BBB+15].

3.5.3.2 Measurement Evaluation of Adaptation Support

The focus of the measurement evaluation of the adaptation support is to identify control plane latency overheads due to reconfigurations.

Fig. 3.22 shows the evaluation setup of the virtualization system operating on real SDN hardware. The setup includes five virtual machines: one for emulating the SDN controller, one for hosting the hypervisor proxy, two for hosting the hypervisor functions, and one to run an SDN software switch. The latency between proxy and hypervisor software is set to 5 ms and the latency between software

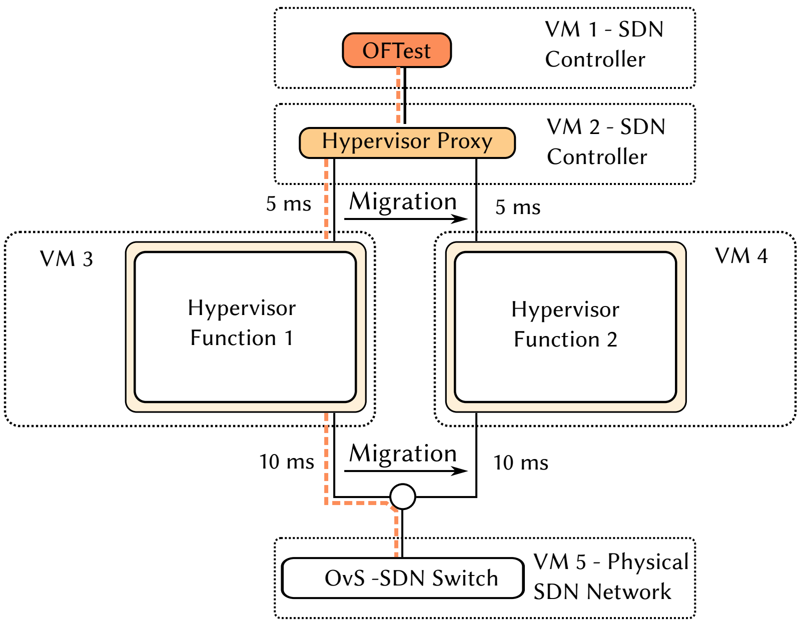


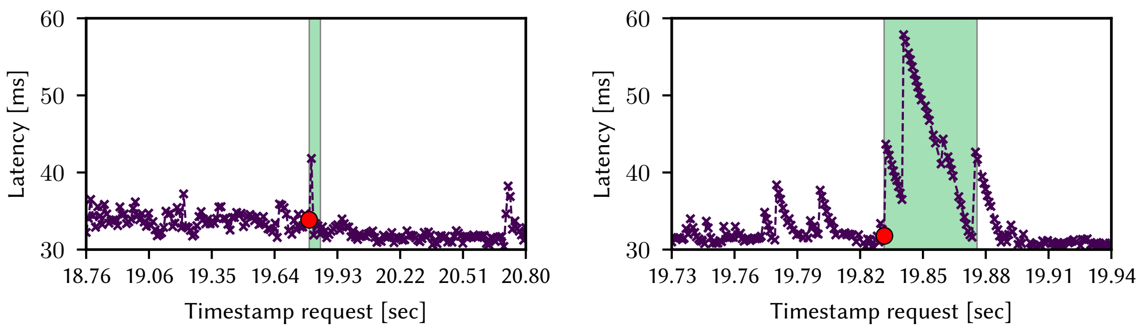
Figure 3.22: Evaluation setup for the control path migration protocol.

and switch to 10 ms. Hence, the end-to-end latency between the proxy and the switch is above 30 ms. This setting should reflect more realistic use-cases.

Implementation details. The SDN switch is an Open vSwitch (OvS) [PPA+09], running OF 1.2. The SDN controller is an *OFtest* instance [Oft]. It is adapted for this setup to be able to support sending a constant OF message rate. The virtualization layer is implemented by hypervisor instances in Python. The implementations use the open source OF library OpenflowJ LoxiGen [Lox]; the library provides message parsing and generation for OF 1.0 and OF 1.3. The proxy is also implemented in Python. It only contains the forward mappings, i.e., the lookup for forwarding the tenant messages to the correct hypervisor instance.

Procedure. The measurement procedure is as follows: the SDN controller sends a constant message rate, which is forwarded through the left hypervisor function towards the SDN switch. Then, the handover is triggered to move the control plane connection to the second hypervisor function. The control plane latency is monitored to quantify the reconfiguration overhead. Furthermore, the hypervisor instances log the messages they process during execution.

Data. The result data contains at least 10 runs per message rate, each with a duration of 40 s. Each run shows the effect of one handover event. The used OF message type is OFPT_FEATURES-_REQUEST: it is a valuable representative to benchmark the whole processing loop from sending via the controller until receiving the reply from the virtualization layer.



(a) OF rate 100 msg/s.

(b) OF 1000 msg/s.

Figure 3.23: Time-series of control plane latencies for two runs with OF message rates 100 msg/s and 1000 msg/s. Red dots mark the start of the handover. The green-highlighted areas show the duration of handover and mark the affected messages.

3.5.3.3 Measurement Results

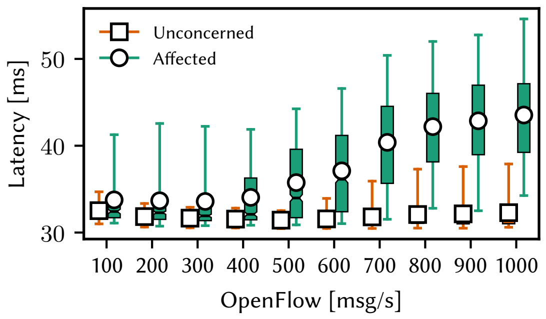
Fig. 3.23 shows the control plane latency over time for two control plane message rates: 100 and 1 000. The start of the handover procedure is identified by the `xid`[4] of the last message that is processed by the left hypervisor instance. When the right hypervisor instance processes the first message, the handover procedure has finished: the `xid` of this message marks the end of the procedure. The time between the start and the finish of the handover procedure determines its duration. The OF messages that are transmitted during the handover procedure are called *affected*. Analogous, messages that are not affected by a reconfiguration process are called *unconcerned*.

Do reconfigurations affect control plane latency? The control plane latency over time indicates that the reconfiguration affects the control plane latency. For all message rates, the control plane latency increases during the handover process. Moreover, a larger message rate can increase the control plane latency more: for 100 requests, Fig. 3.23a shows a peak of 43 ms, whereas 1 000 requests have the highest peak latency of 57 ms (Fig. 3.23b).

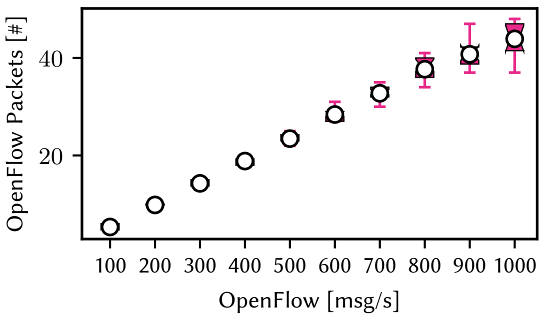
An increasing message rate also affects more OF messages: for 100 requests, the number of affected messages is 3 in contrast to 45 affected messages for 1 000 requests per second. The reason is that a higher rate leads to more packets (messages) in-flight; thus more messages are buffered at the right hypervisor instance due to the protocol's procedure: the right hypervisor buffers all OFPT_FEATURES_REQUEST messages till the migration is completed — then it sends all buffered control messages to the switch. It can be noted that no OF messages are lost during the handover.

The higher the rates, the higher the impact? Fig. 3.24a shows boxplots for both message groups: the average latency of the *unconcerned* messages via square boxes, and the average latency of the *affected* ones via circles. The whiskers range from the lower 5 % to the upper 95 % of measured control plane latency. The average latency of unconcerned messages is constant over small message rates. In contrast, higher message rates increase the average latency and also the maximum 95 %. For

[4] Part of the OF header. `xid` is a transaction id associated with a message. Message replies should use the same `xid` as requests.



(a) Latency: unconcerned vs affected.



(b) Number of affected messages.

Figure 3.24: Latency over number of OF messages (Fig. 3.24a) and affected messages due to reconfigurations (Fig. 3.24b). Comparison between the control plane latency during operation without reconfigurations and with reconfigurations. Reconfigurations and higher message rates increase the operational latency and the number of affected OF messages.

the total number of affected messages, Fig. 3.24b illustrates that this number also increases with the message rate.

We conclude: the higher the rate, the more messages are affected; hence, frequently reconfiguring the virtualization layer can significantly increase the overall control plane latency, with already up to 25 % for 1×10^3 msg/s. This can become very critical, for instance in virtualized environments, such as data centers. Here, it has been shown that switches see up to 100 new flows arriving every millisecond [BAM10; KSG+09]. In the worst case, every flow demands an interaction from a tenant SDN controller. Assuming a linear dependency between OF message rate and affected packets, one reconfiguration can worsen the latency of up to 4 to 5 thousand flows. Accordingly, optimization solutions are needed that can generally try to avoid reconfigurations; virtualization layers should rather make use of reconfigurations in a planned manner.

3.6 Summary

In this chapter, we analyze the existing implementations of state-of-the-art SDN network hypervisors, which realize the virtualization layer in vSDNs. Although there exists a wide range of hypervisor designs, we observed the following shortcomings: the lack of a comprehensive benchmarking framework to quantify performance and the lack of a virtualization layer design that provides flexible and adaptable mechanisms with guaranteed network performance.

Accordingly, this chapter proposes a new benchmarking tool - *perfbench*. The tool satisfies the requirements for realistic hypervisor benchmarks: controllable-high, stable, and variable OF message rates as well as the emulation of multiple tenants and switches simultaneously. With *perfbench*, we find that hypervisors differently impact on the vSDN performance under varying workloads. Whereas FV adds less latency overhead, OVX can scale to larger networking scenarios with more tenants and larger topologies. The results and the identified performance criteria can help researchers to develop refined performance models for multi-tenant software-defined networks.

Based on the measurements, we identify the lack of a virtualization layer design that provides more predictable network operation. Hence, this chapter proposes HyperFlex - an adaptable virtualization layer architecture with improved predictable network control plane performance guarantees for tenants. HyperFlex decomposes the virtualization layer into software and hardware-based functions; the functions can be flexibly placed among servers or networking hardware. In order to provide predictable network operation even for the new flexible virtualization layer design, two mechanisms are implemented: an adaptation mechanism and a control plane isolation concept.

The measurement studies on the control plane isolation reveal trade-offs between software and hardware implementations: their usage depends on the OF message type and current infrastructure utilization, as already illustrated for the hypervisor measurements. Measurements of the adaptation mechanism show that the control plane assignments of switches to hypervisor functions can be changed at runtime. This comes, however, at the cost of increased control plane latency during the migration process; accordingly, the next chapter will also focus on analyzing and optimizing reconfigurations in dynamic traffic scenarios.

Chapter 4

Modeling and Optimization of Network Virtualization Layer Placement Problems

The previous chapter has demonstrated that hypervisor architectures and implementations can show significantly varying performance in terms of control plane latency. Beside adding latency due to implementation choices, there is another factor impacting the achievable control plane latency: the physical locations of the hypervisor (virtualization) functions. Only by knowing the impact of hypervisor locations on the control plane latency, network operators can offer and guarantee their tenants predictable control plane latencies - an indispensable feature for predictable network operation of SDN networks. This chapter focuses on modeling and optimization of the placement of network hypervisor instances.

SDN controllers connect directly to switches; in contrast, the virtualization layer adds another layer of indirection: the control traffic of tenants has to pass trough the virtualization layer. Hence, tenant controllers may experience higher control plane latencies than in non-virtualized SDN networks, which we call *the cost of virtualization*. As a consequence, the virtualization layer demands an even more sophisticated planning to make vSDNs a credible alternative to non-virtualized SDNs. This chapter models and analyzes the k-Network Hypervisor Placement Problem (HPP): the k-HPP answers the fundamental questions of how many hypervisors (the number k) are needed and where to place them in the network.

Whereas some hypervisor architectures rely only on basic SDN features, some hypervisors can make use of special switch functionalities, e.g., the *multiple controllers* feature [BBB+15]. Using the multiple controllers feature, switches can simultaneously connect to multiple SDN controllers, i.e., hypervisor instances. Multi-controller switches may improve control plane performance, e.g., reduce control plane latency as they can balance the load among the available connections. However, multi-controller switches demand additional synchronization between distributed hypervisor instances; hypervisor instances may need to synchronize flow table access or to carefully plan the allocation of available flow table space. Thus, the placement of multi-controller switches needs to be carefully planned. We refer to this planning problem as the Multi-controller Switch Deployment Problem (McSDP) in this chapter.

SDN network hypervisors provide a more flexible way of adaptation: locations of hypervisor functions or whole hypervisor instances can be changed at runtime as they do not rely on special hardware at fixed locations anymore. However, adapting the virtualization layer introduces reconfigurations - which we call *the cost of adaptation*. Taking reconfigurations not into account can result in severe networking problems: network outages, service interruptions, long unacceptable downtimes of services, or additional data plane latency [PZH+11; ICM+02; GJN11; GMK+16]. Planning the virtualization layer requires new optimization models and methods to analyze and optimize for reconfigurations, which do not yet exist in literature.

Our main contribution in this chapter is the in-depth study of the k-HPP for four SDN network hypervisor architectures with respect to control plane latency in cases of static and dynamic use. We provide MILP-based models solving the placement problems; our models jointly solve the McSDP and the k-HPP. We determine and investigate the best locations of hypervisor instances and multi-controller switches with our models for real network topologies and a wide range of vSDN settings. Considering also the dynamic use, we analyze the trade-offs between four hypervisor latency objective metrics and the number of hypervisor reconfigurations. We also closely examine how virtualization affects the individual vSDN requests. Furthermore, we analyze the benefits of a priori optimization of the locations of the vSDN controllers. Specifically, we investigate the impacts of three different controller placement strategies on the k-HPP and McSDP for static and dynamic traffic use-cases.

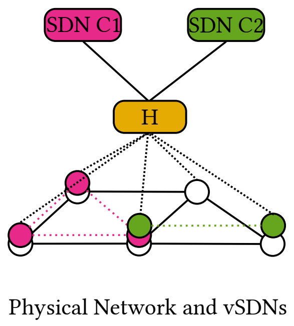
Content and outline of this chapter. Sec. 4.1 introduces background, where four hypervisor architectures are classified, and related work, which summarizes research on the facility location problem, the SDN controller placement problem, and the network function placement problem. The hypervisor problem settings and placement models for static and dynamic use are introduced in Sec. 4.2 and Sec. 4.3, which rely in parts on content from [BBZ+15; BBZ+16; ZBK16]. Sec. 4.4 presents the results of the conducted simulations. Finally, Sec. 4.5 concludes this chapter.

4.1 Background and Related Work

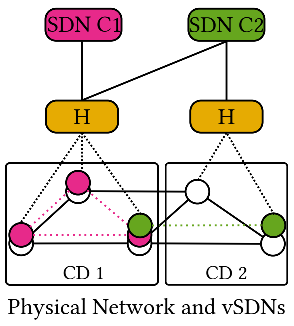
This section first introduces background on hypervisor architectures and second reports on placement and optimization work related to this chapter.

4.1.1 Background on SDN Network Hypervisor Architectures

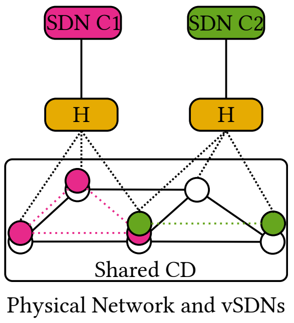
In this section, we introduce four hypervisor architecture categories. We categorize the architectures into *centralized architectures* and *distributed architectures*. We further sub-classify the distributed architectures into architectures operating with *single-controller SDN switches* or with *multi-controller SDN switches*. In addition, we consider distributed *hybrid architectures* that combine single- and multi-controller SDN switches. A single centralized hypervisor instance (at a single location) provides the virtualization functionality in a centralized architecture. In contrast, in a distributed hypervisor architecture, multiple hypervisor instances that are distributed over multiple locations realize the virtualization functionality. We denote the number of hypervisor instances by k and the number of multi-controller switches by M.



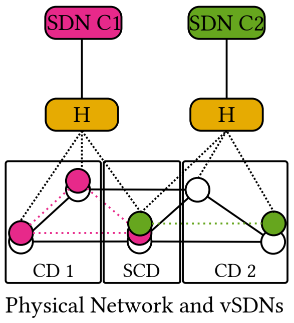
(a) Centralized architecture, $k = 1$.



(b) Distributed architecture + single-controller SDN switches, $k > 1, M = 0$.



(c) Distributed architecture + multi-controller SDN switches, $k > 1, M = |\mathcal{V}|$.



(d) Distr. hybrid arch. with single- and multi-controller SDN switches, $k > 1, 0 < M < |\mathcal{V}|$.

Figure 4.1: Illustration of four hypervisor architecture categories (characterized by number of hypervisor instances k and number of multi-controller switches M in relation to the number of physical switches $|\mathcal{V}|$) for an example SDN network with two vSDNs. The purple and green color differentiate the two vSDNs. A hypervisor instance (location) is represented by a squared box labeled with H. The square boxes represent the non-shared control domains (CDs) and the shared control domains (SCDs or Shared CD) in case of multiple hypervisor instances. A colored and filled circle is a vSDN switch (node) hosted on a non-filled circle, which represents a physical SDN switch (node). The solid lines between these boxes represent the data plane connections, i.e., the edges of the physical SDN network. A solid line represents a connection between an SDN controller (SDN-C) and a hypervisor instance "H". A dashed line represents a physical connection between a hypervisor and a physical SDN switch. Dotted lines either represent the connection between the vSDN switches (filled-colored circles) or between the hypervisor and the physical nodes (non-filled circles).

Centralized network hypervisor architecture. The centralized SDN network hypervisor architecture ($k = 1$) deploys only a single hypervisor instance (at a single location) for SDN network virtualization. vSDNs can be provided by running this single hypervisor instance at one physical network location. FV [SNS+10] is an example of a centralized hypervisor architecture. The centralized hypervisor architecture works with SDN switches compliant with the OF specification [Ope14a]. OF specification compliant switches do not provide any specialized functionalities supporting virtualization [Ope14a].

Fig. 4.1a shows an exemplary centralized hypervisor architecture setup. The hypervisor connects down to five physical SDN switches (nodes, network elements) and up to two vSDN controllers. Two vSDNs (purple and green) are sharing the infrastructure: the purple controller controls three virtual switch instances and the green controller two. All control traffic of the vSDNs has to pass through this single hypervisor instance. The hypervisor forwards the control traffic towards the corresponding vSDN controller.

Distributed network hypervisor architecture for single-controller SDN switches. A hypervisor can be distributed into multiple ($k > 1$) hypervisor instances that are distributed over multiple (k) locations in the network. Suppose that the SDN switches can only connect to one hypervisor instance at a time ($M = 0$). Accordingly, the physical SDN network is split into multiple control domains, whereby one hypervisor instance is responsible for a given domain. An example for a distributed SDN hypervisor architecture operating with single-controller SDN switches is FlowN [DKR13].

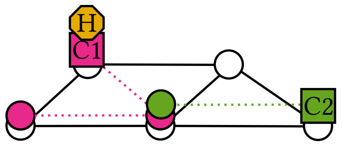
Fig. 4.1b shows an example with two hypervisor instances; the switches are now controlled by $k = 2$ hypervisors. Each SDN switch connects to either one of the $k = 2$ hypervisor instances. Note that one hypervisor connects to multiple controllers (as illustrated for the left hypervisor instance). For instance, as the left hypervisor controls switches hosting virtual instances for both tenant controllers, the hypervisor connects to both tenant controllers. Vice versa, the right SDN controller connects with two hypervisor instances.

Distributed network hypervisor architecture for multi-controller SDN switches. The distributed network hypervisor architecture for multi-controller switches realizes the SDN virtualization layer via multiple separated hypervisor instances ($k > 1$). However, all $|\mathcal{V}|$ physical SDN switches can now simultaneously connect to multiple hypervisor instances as it is assumed that all switches support the multiple controllers feature, i.e., $M = |\mathcal{V}|$ (See Section 3.5.3 for the introduction and use of the multiple controllers feature). As a result, there is no separation of the control domain of the SDN switches as each switch can be simultaneously controlled by multiple hypervisor instances. The adaptation support as introduced in Chapter 3 makes use of the multi-controller feature.

While each physical SDN switch is only connected to a single hypervisor instance in Fig. 4.1b, Fig. 4.1c shows two hypervisor control connections for one physical SDN switch. The multi-controller feature makes it possible that an SDN switch connects to multiple hypervisor instances during operation. The hypervisor instances need to coordinate their resource management on the switches, as switch resources, e.g., switch CPU and flow tables, are shared and not strictly isolated.

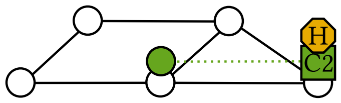
Distributed hybrid network hypervisor architecture. In general, only some SDN switches need to support the multi-controller features to achieve an optimization goal. This yields the option to reduce the number of multi-controller switches, which reduces coordination overhead and controllers interfering on shared switches. This leads to the distributed hybrid architecture operating on hybrid SDN networks. In this chapter, a hybrid SDN network is defined as an SDN network consisting of switches supporting single-controller and multi-controller switches.

Fig. 4.1d illustrates an example: while the two left switches connect only to the left hypervisor and the two right switches connect to the right hypervisor, the middle switch ($M = 1$) connects to both hypervisors. Both hypervisor instances share the control domain of the middle switch. The network is now divided into shared (SCD) and non-shared control domains (CD). The switches of the non-shared control domains operate in single-controller mode. The proposed models of this chapter can prescribe a maximum number M of multi-controller SDN switches when optimizing for a given



Physical Network

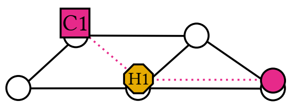
(a) Initial Hypervisor Placement. Hypervisor placement serves two virtual networks.



Physical Network

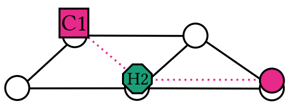
(b) Hypervisor placement after adapting to changed number of virtual network requests. Hypervisor location changed compared to initial placement.

Figure 4.2: Motivation: From HPP to DHPP.



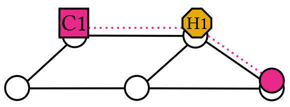
Physical Network

(a) Initial Placement.



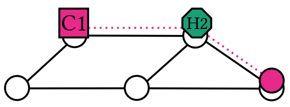
Physical Network

(b) Hypervisor instance changes.



Physical Network

(c) Hypervisor location changes.



Physical Network

(d) Hypervisor location and instance change.

Figure 4.3: Different types of reconfigurations: hypervisor instance change, hypervisor location change, and hypervisor instance & location change.

substrate network. The outcomes of optimization solutions provide the locations and the numbers of needed multi-controller switches to achieve an optimization objective.

4.1.1.1 Background on Reconfigurations in SDN Network Virtualization Layers

Solving the HPP provides not only the locations of the hypervisor instances, it also generates the routing of the connections between tenant controllers and their virtual switches. We call a single controller-hypervisor-switch connection a Virtual Control Path (VCP); a VCP can also be seen as a chain where the control packets need to traverse one hypervisor or virtualization function. Fig. 4.2a shows an initial hypervisor placement: one hypervisor is located at a single node in the center of the network. Network dynamics, such as link failures or changes of the network requests, may require an adaptation of the virtualization layer, i.e., the hypervisor location. Whereas a specific placement provides the optimal latency for one traffic state, a virtualization layer may need to adapt the placement to react to a change in the network traffic demands, such as a leaving vSDN. In Fig. 4.2b, one virtual network leaves the substrate network; the new optimal hypervisor placement is now close to the controller of the remaining virtual network - the hypervisor location changed.

Adapting a placement of hypervisor instances of a virtualization layer introduces reconfigurations: hypervisor locations may change, the number of hypervisors changes, or the routing of the VCPs. Taking the view of a tenant, three reconfiguration types affect its VCP, which are modeled and analyzed in this thesis: a change of the hypervisor instance, a change of the hypervisor location, or a change of both the hypervisor instance and location. Fig. 4.3 provides an overview of the different placements after a preceding reconfiguration. Fig. 4.3a illustrates the initial placement: controller C1 (purple square) connects through the hypervisor instance one (orange hexagon) to its virtual network, i.e., virtual switch (purple circle).

- **Hypervisor instance change.** In Fig. 4.3b, the instance responsible for managing the control traffic of the VCP changed. In such scenario, the underlying TCP establishing the control connection between both hypervisor and controller as well as hypervisor and switch may need to re-initiate; this may add latency as shown in Chapter 3. Moreover, network state information concerning the VCP may need to be exchanged between the old and the new hypervisor instance.
- **Hypervisor location change.** In this scenario, as illustrated in Fig. 4.3c, the location of the hypervisor instance changes. Consequently, the routing of the VCP needs to be updated to the new hypervisor location. Such operation involves the transmission of the hypervisor memory, which stores, e.g., the current state of its connected virtual networks.
- **Hypervisor instance & location change.** This operation might introduce the most overhead: a change of both the hypervisor instance and the location (Fig. 4.3d). The operation induces all the overhead of the previous changes.

The last two changes are seen as the ones that mostly affect the operation of the tenants and are most costly for the network operator (state copies, path updates, control plane interruptions etc.). Because of this reason, this thesis covers the minimization of hypervisor location changes including also potential instance changes.

4.1.2 Related Work

We review main research on optimization and placement related to the virtualization of SDN networks in this section. The related work is classified into three problem categories: facility location problems, controller placement problems, and network function placement problems.

Facility Location Problem

As indicated by Heller et al. [HSM12], the general Facility Location Problem (FLP) underlies the SDN Controller Placement Problem (CPP). Originally, the Weber problem is one of the first simple facility location problems: the solution to the Weber problem is a point (node) in a network (graph) that minimizes the sum of the transportation costs from this node to n destination nodes. The k-HPP can be related to the hierarchical facility location problem [FHF+14]. The task of the hierarchical facility location problem is to find the best facility locations in a multi-level network. The facilities at higher levels have to serve the facilities at lower levels, while customers need to be served at the lowest level. A similar layering can be applied to the k-HPP: tenant controllers need to connect to hypervisor instances, while hypervisor instances need to connect to SDN switches at the lowest level. Different variations, adaptations to real problems, and overviews of the FLP are provided in [ALL+96; KD05; PJ98; GMM00; FHF+14]. The grouping of demands has initially been investigated in [DW00; BDW01]. One unique feature of the k-HPP is the differentiation of groups of customers, i.e., individual vSDNs, which need to be specifically operated by their corresponding tenant controllers, which can also be seen as a combination of demand grouping and hierarchy of facilities. Another unique feature comes from the multiple controllers feature; one switch can connect to multiple hypervisors.

SDN Controller Placement Problem

The SDN CPP for non-virtualized SDN networks has been initiated in [HSM12]. The CPP targets the question of how many controllers are needed and where to place them. Using a brute-force method, Heller et al. [HSM12] evaluate the impact of controller placement on average and maximum latency metrics for real network topologies. The authors conclude that five controllers are sufficient to achieve an acceptable control plane latency for most topologies. As different optimization objectives, e.g., load and delay, are critical for the operation of SDN networks, Lange et al. [LGZ+15] apply multi-objective optimization approaches. Their framework uses simulated annealing to analyze the CPP for different network topologies with respect to multiple objectives, e.g., latency and resilience. As real SDN networks have node and link capacity constraints, mathematical models for solving the CPP with node and link capacity have been studied in [YBL+14; SSH15]. Considering capacity constraints during planning protects SDN controllers from overload situations. Distributed SDN controllers can be organized in a hierarchy to achieve resilience [JCPG14]. Jimenez et al. [JCPG14] provide an algorithm and performance comparisons for k-center and k-median-based algorithms. Additional CPP research either considers different metrics, e.g., resilience or load balancing [LGS+15; HWG+14; MOL+14], or incorporates different methodologies, e.g., clustering, solving the CPP. A dynamic version of the CPP, where the rate of flow setups varies over time, has been studied in [BRC+13].

In virtual SDN environments, each vSDN uses its own controller, which needs to be placed for each vSDN individually. The present study solves the CPP a priori for maximum or average latency objectives. It then uses the tenant controller locations as an input when optimizing the placement of the hypervisor instances. This two step optimization makes it possible to analyze the impact of the vSDN controller placement on the placement of hypervisors.

Network Function Placement Problem

Luizelli et al. [LBB+15] propose a formulation for the embedding of Virtual Network Function (VNF) chains on a network infrastructure. The proposed model targets a minimum number of virtual network functions to be mapped on the substrate. Their evaluation metrics mainly considered infrastructure resource utilization, e.g., CPU, as well as end-to-end latency of the embedded function chains. However, the proposed model does not incorporate the characteristics of SDN networks, moreover virtual SDN networks.

Network functions like virtualization functions face dynamic network traffic. Accordingly, operators might adapt the placement of VNFs to always achieve the highest resource efficiency: for instance, paths between functions should always be short to save network resources or functions should always be accommodated for energy efficiency. Clayman et al. [CMG+14] introduce an architecture managing and orchestrating network functions for dynamic use. Their approach includes different policies, for instance, placing new VNFs on the least loaded server. They neglect, however, the dynamics induced by adaptations potentially leading to service interruptions: they do not minimize the number of reconfigurations resulting in potential service interruptions.

Ghaznavi et al. [GKS+15] also aim at a dynamic use when placing VNFs; they consider QoS requirements and operational costs while paying attention to reconfigurations. They consider different reconfiguration types: reassignment of load between VNFs and the migration of VNFs. As they claim that such optimization problem is NP-hard, Ghaznavi et al. [GKS+15] design heuristics for the whole management process including installation, migration, and general task handling. Their conducted simulations show that their non-greedy heuristic improves the overall performance when compared to a greedy algorithm; however, they do not provide a detailed study of the mathematical model; hence, a general quantification of the heuristic is missing.

Kawashima et al. [KOO+16] take a different approach: their goal is to avoid peak reconfigurations induced by the simultaneous migration of multiple VNF instances. In order to mitigate the migration of many VNFs, they try to predict future demands within a given time window. A MILP-based model integrates the predictions of demands while it minimizes the number of active physical nodes and reconfigurations. Since the demand prediction may be imprecise, the mathematical model might not always provide the optimal solution for the network traffic of the predicted time.

In contrast to the VNF placement, the hypervisor placement considers groups of virtual switches that need to be connected to tenant controllers. To our best knowledge, neither latency as a measure nor optimizing the sum of distances of individual groups has been investigated in the context of VNF placement yet.

4.2 Placement Problems in Virtual Software-Defined Networks

Network operators need to solve a variety of placement problems when virtualizing SDN environments: the placement of the virtual SDN controller, the embedding of the virtual networks, the placement of the hypervisor functions, the determination of the switches supporting special virtualization features, like the multi-controller feature, etc. The overall goal of an operator might be to know the impact of different placements in order to offer his tenants acceptable levels of control plane latencies. Consequently, an operator needs to jointly solve the preceding optimization problems to find the best trade-off among the various design choices. Accordingly, this thesis models and optimizes parts of the problems jointly (e.g., the hypervisor placement and the multi-controller switch deployment).

4.2.1 Problem Setting for Network Hypervisor Placement Problem (HPP) and the Multi-controller Switch Deployment Problem (McSDP)

This section introduces the problem settings for the Network Hypervisor Placement Problem (HPP) and the Multi-controller Switch Deployment Problem (McSDP). It first defines the notation for the physical SDN network and the vSDN requests. Then, it introduces the mathematical definition of the k-HPP and the McSDP.

4.2.1.1 Network Models

The input of the k-HPP is given by the set of vSDN requests $\mathcal{R}$, which are to be hosted on a physical SDN network graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$.

Table 4.1: Notation of sets for physical SDN network $\mathcal{G}$ for static use.

Notation	Description
$\mathcal{G}(\mathcal{V}, \mathcal{E})$	Physical SDN network graph
$\mathcal{V}$	Set of physical SDN switches (network nodes), i.e., node locations
i	Physical SDN switch (network node) $i \in \mathcal{V}$
$\mathcal{E}$	Set of physical network edges
e	Physical edge $e \in \mathcal{E}$
$\mathcal{H}$	Set of potential hypervisor nodes (locations) with $\mathcal{H} \subseteq \mathcal{V}$
$\mathcal{P}$	Set of pre-calculated shortest paths between all network node pairs

Table 4.2: Notation of helper functions for physical SDN network $\mathcal{G}$.

Notation	Description
$\lambda : \mathcal{E} \to \mathbb{R}_{>0}$	Latency of edge e, with $\lambda(e) \in \mathbb{R}^+$
$d : \mathcal{V}^2 \to \mathbb{R}_{\geq 0}$	Latency (distance) of shortest path $(s,t) \in \mathcal{P}$, with $d(s,t) \in \mathbb{R}_{\geq 0}$
$d_h : \mathcal{V}^3 \to \mathbb{R}_{\geq 0}$	Latency (distance) of path connecting nodes s and t while traversing (passing) node h, with $d_h(s,h,t) \in \mathbb{R}_{\geq 0}$

Physical SDN network specification. Tables 4.1 and 4.2 summarize the notation of sets for the physical SDN network and the helper functions. The network is modeled as a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ with physical SDN switches (network nodes) $i \in \mathcal{V}$ connected by undirected edges $e \in \mathcal{E}$. The potential hypervisor nodes (locations) are given by the set $\mathcal{H}$. They are a subset of $\mathcal{V}$, i.e., $\mathcal{H} \subseteq \mathcal{V}$. The set $\mathcal{P}$ contains the shortest paths of the network between any network node pair. A shortest path is denoted as $(s,t) \in \mathcal{P}$.

The function $\lambda : \mathcal{E} \to \mathbb{R}_{>0}$ computes for an edge e the latency from the geographical distance between two network nodes that are connected via edge e. Note that the function does not consider the transmission bit rate (edge capacity). The latency $\lambda(e)$ of an edge $e \in \mathcal{E}$ is used for evaluating the latency of network paths. The distances, i.e., latencies of shortest paths are given by the function $d : \mathcal{V}^2 \to \mathbb{R}_{\geq 0}$. Accordingly, the distance of a shortest path $(s,t) \in \mathcal{P}$ can be calculated by $d(s,t)$. Furthermore, the function $d_h : \mathcal{V}^3 \to \mathbb{R}_{\geq 0}$ gives the latency of the shortest path connecting two nodes traversing an intermediate node; $d_h(s,h,t)$ gives the latency of the shortest path connection between nodes s and t via node h. This value is calculated as the sum of $d(s,h)$ and $d(h,t)$.

Virtual SDN Network (vSDN) request. Tables 4.3 and 4.4 summarize the notation for the vSDN requests $\mathcal{R}$ and the helper functions regarding the vSDN requests. A vSDN request $r \in \mathcal{R}$ is defined by the set of virtual SDN network nodes $\mathcal{V}^r$ and the vSDN controller c^r. All vSDN network nodes $v^r \in \mathcal{V}^r$ of a request r need to be connected to their controller instance c^r. Note that we assume a vSDN to operate only one SDN controller; multi-controller vSDNs are not considered in

Table 4.3: Notation of sets and constants for virtual SDN network (vSDN) requests $\mathcal{R}$.

Notation	Description
$\mathcal{R}$	Set of vSDN requests
r	Virtual network request $r \in \mathcal{R}$
$\mathcal{V}^r$	Set of virtual nodes of vSDN request $r \in \mathcal{R}$
v^r	Virtual network node $v^r \in \mathcal{V}^r$, with $\pi(v^r) \in \mathcal{V}$
c^r	Virtual controller node of vSDN request r, with $\pi(c^r) \in \mathcal{V}$

Table 4.4: Notation of helper functions for virtual SDN network (vSDN) requests $\mathcal{R}$.

Notation	Description
$\forall r \in \mathcal{R}, \forall v^r \in \mathcal{V}^r \cup \{c^r\} : \pi : \mathcal{V}^r \cup \{c^r\} \to \mathcal{V}$	Mapping from virtual node v^r or controller c^r to their physical host switch (network node) i: function is defined for both virtual switches and controllers.

the evaluations.

The physical SDN switch (location) of a vSDN network node is given by the function $\pi(v^r)$, i.e., $\pi(v^r) \in \mathcal{V}$. The location of the controller is also chosen among the available network node locations, i.e., $\pi(c^r) \in \mathcal{V}$.

4.2.1.2 k-Network Hypervisor Placement Problem (k-HPP)

Table 4.5 specifies the input of the k-HPP. For a given physical SDN network $\mathcal{G}$ and set of vSDN requests $\mathcal{R}$, a prescribed number k of hypervisor locations need to be chosen among all potential hypervisor locations $\mathcal{H}$. The set $\mathcal{H}$ specifies the hypervisor locations on the network; hypervisors can only be placed at the locations $\mathcal{H}$. In real networks, those hypervisor locations could be data center locations, which are connected to the network topology at the given network locations $i \in \mathcal{H} \subseteq \mathcal{V}$. Optimizing the k-hypervisor placement provides many outcomes: the locations of the hypervisors, the assignment between switches and hypervisors, the routing of vSDN demands.

4.2.1.3 Multi-controller Switch Deployment Problem (McSDP)

We denote M for the number of multi-controller SDN network nodes. Solving our problem formulation determines which switches should support multiple controllers (hypervisors). An alternative input setting of our problem formulation could include a pre-determined set of switches supporting the multiple controllers feature. In case $M = 0$, no physical SDN switch supports the multiple controllers feature, i.e., no SDN switch can simultaneously connect to multiple hypervisor instances. For $0 < M < |\mathcal{V}|$, a subset of the physical SDN switches supports multiple controllers. In case $M = |\mathcal{V}|$, all physical SDN switches support multiple controllers.

Table 4.5: Problem input for k-HPP and McSDP.

Notation	Description
$\mathcal{G}$	Physical SDN network
$\mathcal{R}$	Set of virtual SDN network (vSDN) requests
k	Number of hypervisor nodes to be placed
M	Number of physical SDN switches (network nodes) supporting multiple controllers

4.2.2 Problem Setting for Dynamic Hypervisor Placement Problem (DHPP)

The DHPP aims at dynamic traffic scenarios: the number of vSDN requests changes over time. The hypervisor locations should always provide the best possible latency for the different amounts of vSDN requests over time: e.g., whereas hypervisors might be co-located with vSDN controllers for a small number of vSDN requests, the hypervisor locations might tend to more central locations for a larger amount of vSDN requests. Changing the hypervisor locations introduces reconfigurations (see Sec. 4.1.1.1).

4.2.2.1 Network Models for Dynamic Use

This section briefly reports on the hypervisor problem setting for dynamic use. Taking the static model setting as given, the section describes only the updated notation considering particular the time aspect. In case a detailed explanation is missing, all explanations from the static case hold also for the dynamic case.

Table 4.6: Notation of sets and constants for physical SDN network $\mathcal{G}$ for dynamic use.

Notation	Description
$\mathcal{T}$	Set of points in time over a finite time horizon $\Psi = [0, T]$
τ	A point in time with $\tau \in \mathcal{T}$
Φ	Set of hypervisor instances with $\lvert\Phi\rvert = k$

Physical SDN network specification for dynamic use. Table 4.6 provides the main difference between the dynamic and the static setup: the dynamic hypervisor placement is investigated for points of time $\tau \in \mathcal{T}$ where τ is the current point in time and $\tau - 1$ is the point in time before τ. It is assumed that changes do not happen between $\tau - 1$ and τ. In contrast to the static use case, the dynamic one additionally considers individual hypervisor instances, denoted by the set Φ. Modeling individual hypervisor instances makes it possible to account for reconfigurations of the adapted hypervisor instance: for example, the hypervisor instances might store different information about their connected switches such as the flow table settings of the switches. When reconfiguring these hypervisors, transmitting the different state information might lead to different reconfiguration times

Table 4.7: Notation of sets and constants for virtual SDN network (vSDN) requests $\mathcal{R}^\tau$ for dynamic use.

Notation	Description
$\mathcal{R}^\tau$	Set of virtual network requests at point in time $\tau \in \mathcal{T}$
r	Virtual network request $r \in \mathcal{R}^\tau$
$\bar{\mathcal{R}^\tau}$	$= \mathcal{R}^\tau \cap \mathcal{R}^{\tau-1}$ set of persistent virtual networks

due to varying sizes of state information. By differentiating the hypervisor instances, the model can analyze the cost of such instance reconfigurations. All other notations of sets and constants can be taken from the static use case as listed in Tab. 4.1; it is assumed that the substrate network settings do not change over time.

Virtual SDN network (vSDN) request for dynamic use. Table 4.7 summarizes the notations regarding the vSDN requests for the dynamic use case. The set $\mathcal{R}^\tau$ denotes the requests existing at time τ; in parallel, the set $\mathcal{R}^{\tau-1}$ stands for the requests that are present before, i.e., the requests at $\tau - 1$. As a special extension of the dynamic setup, the set $\bar{\mathcal{R}^\tau}$ represents the persistent requests (i.e., virtual networks) that are present at both points in time $\tau - 1$ and τ; reconfigurations can increase the latencies of the persistent requests $r \in \bar{\mathcal{R}^\tau}$. Table 4.8 outlines the additional helper functions for the dynamic setup: $\psi^\tau_{\text{loc}}(v^r)$ and $\psi^\tau_{\text{ins}}(v^r)$. The function $\psi^\tau_{\text{loc}}(v^r)$ reveals *the location of the responsible hypervisor instance* $\phi \in \Phi$ for a given $v^r \in \mathcal{V}^r$ at τ, whereas function $\psi^\tau_{\text{ins}}(v^r)$ reveals *the responsible instance.*

Table 4.8: Helper functions for virtual SDN network (vSDN) requests $\mathcal{R}^\tau$ for dynamic use.

Notation	Description
$\forall \tau \in \mathcal{T}, \forall r \in \mathcal{R}^\tau : \psi^\tau_{\text{loc}} : \mathcal{V}^r \to \mathcal{H}$	The function resolves for a virtual node v^r at point in time τ the hypervisor location $h \in \mathcal{H}$.
$\forall \tau \in \mathcal{T}, \forall r \in \mathcal{R}^\tau : \psi^\tau_{\text{ins}} : \mathcal{V}^r \to \Phi$	The function resolves for a virtual node v^r at point in time τ the hypervisor instance $\phi \in \Phi$.

4.2.2.2 Modeling vSDN Reconfigurations

Additional sets are introduced that make it possible to consider reconfigurations during optimization. The three sets $\mathcal{Z}^\tau$, $\mathcal{Z}^\tau_{\text{ins}}$ and $\mathcal{Z}^\tau_{\text{loc}}$ contain the virtual nodes of all $r \in \mathcal{R}^\tau$ that are reconnected to a new hypervisor instance or rerouted to a new hypervisor location due to a change; the sets capture the vSDN nodes, i.e., their VCPs that are affected at point in time τ. The sets are defined as follows:

$$\mathcal{Z}^\tau_{\text{loc}} = \bigcup_{r \in \bar{\mathcal{R}^\tau}} \{v^r | v^r \in \mathcal{V}^r : \psi^{\tau-1}_{\text{loc}}(v^r) \neq \psi^\tau_{\text{loc}}(v^r)\} \tag{4.1}$$

$$\mathcal{Z}^\tau_{\text{ins}} = \bigcup_{r \in \bar{\mathcal{R}^\tau}} \{v^r | v^r \in \mathcal{V}^r : \psi^{\tau-1}_{\text{ins}}(v^r) \neq \psi^\tau_{\text{ins}}(v^r)\} \tag{4.2}$$

$$\mathcal{Z}^\tau = \mathcal{Z}^\tau_{\text{ins}} \cup \mathcal{Z}^\tau_{\text{loc}} \tag{4.3}$$

Eq. 4.1 introduces the set $\mathcal{Z}^{\tau}_{\text{loc}}$ containing the virtual nodes $v^r \in \mathcal{V}^r$ of all requests $\mathcal{R}^{\tau}$ whose locations change at τ; this is independent of whether the instance changes or not. Eq. 4.2 gives the virtual nodes $v^r \in \mathcal{V}^r$ of all requests $\mathcal{R}^{\tau}$ whose hypervisor instances change; again, this is independent of whether the location changes or not. The set $\mathcal{Z}^{\tau}$ contains all virtual nodes that reconnect to new locations or different hypervisor instances: it is defined as the union of $\mathcal{Z}^{\tau}_{\text{loc}}$ and $\mathcal{Z}^{\tau}_{\text{ins}}$.

The sets $\mathcal{Z}^{\tau}_{\text{loc},\overline{\text{ins}}}$ and $\mathcal{Z}^{\tau}_{\overline{\text{loc}},\text{ins}}$ are subsets of $\mathcal{Z}^{\tau}_{\text{loc}}$ and $\mathcal{Z}^{\tau}_{\text{ins}}$; they additionally differentiate between instance and location changes. They are defined as:

$$\mathcal{Z}^{\tau}_{\text{loc},\overline{\text{ins}}} = \bigcup_{r\in\mathcal{R}^{\tau}} \{v^r | v^r \in \mathcal{V}^r : \psi^{\tau-1}_{\text{loc}}(v^r) \neq \psi^{\tau}_{\text{loc}}(v^r) \wedge \psi^{\tau}_{\text{ins}}(v^r) = \psi^{\tau-1}_{\text{ins}}(v^r)\} \tag{4.4}$$

$$\mathcal{Z}^{\tau}_{\overline{\text{loc}},\text{ins}} = \bigcup_{r\in\mathcal{R}^{\tau}} \{v^r | v^r \in \mathcal{V}^r : \psi^{\tau-1}_{\text{ins}}(v^r) \neq \psi^{\tau}_{\text{ins}}(v^r) \wedge \psi^{\tau}_{\text{loc}}(v^r) = \psi^{\tau-1}_{\text{loc}}(v^r)\} \tag{4.5}$$

Eq. 4.4 defines the set $\mathcal{Z}^{\tau}_{\text{loc},\overline{\text{ins}}}$ containing the virtual nodes whose location changes but not the hypervisor instance; the VCPs are migrated with the hypervisor instance to a new location. Similarly, the set $\mathcal{Z}^{\tau}_{\overline{\text{loc}},\text{ins}}$ as given by Eq. 4.5 provides the vSDN nodes where only the hypervisor instances change; the vSDN nodes are always connected through the same physical node location to their controllers.

4.2.2.3 Dynamic Hypervisor Placement Problem (DHPP): Multi-Objective Optimization

Similar to the k-HPP and McSDP, the DHPP requires various input parameters summarized in Table 4.9: a physical SDN network $\mathcal{G}$, a time horizon $\mathcal{T}$, and for each point in time a set of requests $\mathcal{R}^{\tau}$. For this input, a prescribed number k of hypervisor locations need to be chosen among all potential hypervisor locations $\mathcal{H}$. Again, M defines the number of multi-controller switches.

The DHPP introduces conflicting objectives: minimizing the control plane latency and minimizing the different types of reconfigurations. For designing network virtualization layers, gaining knowledge of the trade-offs between achievable control plane latencies and the amount of reconfigurations is important for efficient resource management. For instance, a small relaxation of the latency constraint might potentially decrease the amount of reconfigurations significantly, resulting in a more stable network operation while still guaranteeing an acceptable control plane latency.

Different methodologies exist to solve multi-objective optimizations: the weighting method, the constraint method, the non-inferior set estimation method, the multi-objective simplex method etc.. In this thesis, the ϵ-constraint method is applied [CH08; Coh13]; it can reuse the existing HPP model that optimizes latency only; it makes it possible to analyze the different objectives in an isolated manner by controlling relaxations through so called ϵ parameters. In contrast to the constraint method, the weighting method does not efficiently find the Pareto (non-inferior) points; a large number of weights might be needed to find valuable solution points [Coh13]. Consequently, the constraint method is chosen for optimizing the DHPP.

Figure 4.4 shows the multi-objective optimization steps. The first step minimizes one of the four latency objectives (see Section 4.3.1.2); the second step minimizes the number of hypervisor location changes (Eq. 4.27); the third step minimizes the number of hypervisor instance changes (Eq. 4.29). The output of one optimization step is used as input for the next optimization step. For analyzing

Table 4.9: Problem input for DHPP.

Notation	Description
$\mathcal{G}$	Physical SDN network
$\mathcal{T}$	Set of points in time over a finite time horizon $\Psi = [0, T]$
$\mathcal{R}^{\mathcal{T}}$	Set of virtual SDN network (vSDN) requests
$\bar{\mathcal{R}}^{\mathcal{T}}$	Set of consistent virtual SDN network (vSDN) requests
k	Number of hypervisor nodes to be placed
M	Number of physical SDN switches (network nodes) supporting multiple controllers
L'_{obj}	Optimal latency for the given input data; "obj" is a placeholder for avg, avg max, max, or max avg
$\epsilon_{\text{location}}$	Fraction of vSDN nodes that is affected by a reconfiguration from $\tau - 1$ to τ w.r.t. the location of the assigned hypervisor; $\epsilon_{\text{location}} \in [0, 1]$
$\epsilon_{\text{instance}}$	Fraction of vSDN nodes that is allowed to be reconfigured w.r.t. the hypervisor instances; $\epsilon_{\text{instance}} \in [0, 1]$
$\epsilon_{\text{latency}}$	Relative relaxation of target latency L'_{obj}; $\epsilon_{\text{latency}} \in \mathbb{R}^+$

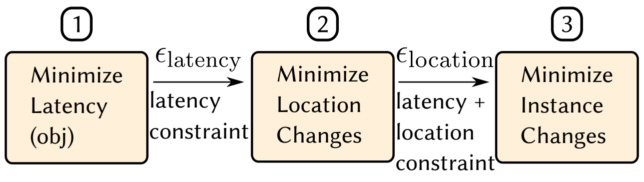

1
Minimize Latency (obj)
$\epsilon_{\text{latency}}$
latency constraint
2
Minimize Location Changes
$\epsilon_{\text{location}}$
latency + location constraint
3
Minimize Instance Changes

Figure 4.4: Optimization stages. $obj \in \{L_{\text{avg}}, L_{\text{avg max}}, L_{\text{max}}, L_{\text{max avg}}\}$, the result of the previous stages is added as a constraint to the current stage (Eq. 4.46 or/and Eq. 4.45).

the trade-offs, i.e., creating the Pareto frontier, the parameters $\epsilon_{\text{instance}}, \epsilon_{\text{latency}}$ and $\epsilon_{\text{location}}$ relax the respective input values of the optimization steps; the next step uses the achieved objective value of the previous step as (relaxed) input. L'_{obj} denotes the value of the latency objective of the first step.

Solving the DHPP provides the following outcomes for a point in time τ: the achievable hypervisor latency objective, the locations of the hypervisors at point in time τ, the assignments of switches to hypervisor locations and instances, and the amount of reconfigurations. The results of this thesis report on trade-offs between minimizing one of the latency objectives ($L_{\text{avg}}, L_{\text{avg max}}, L_{\text{max}}, L_{\text{max avg}}$) and the number of location changes.

4.3 Modeling Network Hypervisor and Multi-Controller Switch Placements

Section 4.3.1 first introduces the hypervisor model for a static use. Section 4.3.2 introduces the dynamic model, which extends the static model for dynamic use.

Table 4.10: Binary decision variables for k-HPP and McSDP.

Notation	Description
y_h	=1, if a hypervisor is placed at potential hypervisor node (location) $h \in \mathcal{H}$; 0, otherwise
x_{r,v^r,h,c^r}	=1, if vSDN node $v^r \in \mathcal{V}^r$ of request r is connected to its controller c^r via hypervisor node (location) $h \in \mathcal{H}$; 0, otherwise
$q_{i,h}$	=1, if physical SDN node $i \in \mathcal{V}$ is controlled by hypervisor node $h \in \mathcal{H}$; 0, otherwise
w_i	=1, if physical SDN node $i \in \mathcal{V}$ is controlled by multiple hypervisor instances, i.e., if multiple node to hypervisor (controller) connections exist; 0, otherwise

4.3.1 Modeling HPP and McSDP for Static Use

This section introduces the decision variables, the four objective functions, and the constraints guaranteeing a correct solution. The modeling does not consider capacity constraints.

4.3.1.1 Decision Variables

Table 4.10 specifies the binary decision variables of the MILP formulation of the k-HPP and McSDP. The variable y_h determines whether a hypervisor is located at the network node (location) $h \in \mathcal{H}$. For a request $r \in \mathcal{R}$, the variable x_{r,v^r,h,c^r} is set to one if the vSDN node $v^r \in \mathcal{V}^r$ is connected to the vSDN controller c^r via the hypervisor node (location) $h \in \mathcal{H}$. Note that if a path x_{r,v^r,h,c^r} is set to one, then a hypervisor needs to be placed at the potential hypervisor node (location) h. The variable $q_{i,h}$ indicates whether physical node $i \in \mathcal{V}$ is controlled by the hypervisor instance placed at location $h \in \mathcal{H}$. The variable w_i indicates whether the multiple controllers feature is deployed and used at physical node $i \in \mathcal{V}$. In case of a multi-controller SDN switch, i.e., where $w_i = 1$, the variable $q_{i,h}$ for a given node $i \in \mathcal{V}$ is possibly one for multiple hypervisor nodes (locations) $h \in \mathcal{H}$.

4.3.1.2 Objective Functions

We focus on objective functions that seek to minimize the control plane latency. In particular, we introduce four latency metrics: maximum latency $L_{\max}$, average latency L_{avg}, average maximum latency $L_{\text{avg}\max}$, and maximum average latency $L_{\max\text{avg}}$. Note that optimizing for $L_{\max}$, $L_{\text{avg}\max}$ and $L_{\max\text{avg}}$ requires additional variables and constraints. These variables and constraints are subsequently introduced when the metrics are presented. As these variables and constraints are specific to an objective, Section 4.3.1.3 does not describe them with the general constraints.

Maximum latency. The maximum latency for a considered hypervisor placement is the maximum latency of all utilized shortest paths from all requests $r \in \mathcal{R}$. Recall that the binary decision variable x_{r,v^r,h,c^r} indicates (i.e., is equal to one) when for a request r the path from v^r via h to c^r is used. Thus, the maximum latency of all paths that have been selected to fulfill the requests $r \in \mathcal{R}$ is given by

$$L_{\max} = \max_{r\in\mathcal{R},\ v^r\in\mathcal{V}^r,\ h\in\mathcal{H}} x_{r,v^r,h,c^r} \cdot d_h(\pi(v^r), h, \pi(c^r)). \tag{4.6}$$

Minimizing the latency metric $L_{\max}$ involves minimizing a maximum over sets, which is not directly amenable to some solvers. The maximum over sets can be readily expressed as an equivalent constrained minimization problem. Specifically, we can equivalently minimize $L_{\max}$defined through the constraints

$$L_{\max} \geq x_{r,v^r,h,c^r} \cdot d_h(\pi(v^r), h, \pi(c^r)),\ \ \forall r\in\mathcal{R},\ \forall v^r\in\mathcal{V}^r,\ \forall h\in\mathcal{H}. \tag{4.7}$$

The resulting objective function is

$$\min L_{\max}. \tag{4.8}$$

Average latency. The average latency is the average of all path latencies of all VCPs. For a vSDN request r, there are $|\mathcal{V}^r|$ vSDN nodes that need to be connected to the vSDN controller c^r. Therefore, for a set of requests $\mathcal{R}$, there are overall $\sum_{r\in\mathcal{R}} |\mathcal{V}^r|$ paths and the average latency is

$$L_{\text{avg}} = \frac{1}{\sum_{r\in\mathcal{R}} |\mathcal{V}^r|} \sum_{r\in\mathcal{R}} \sum_{v^r\in\mathcal{V}^r} \sum_{h\in\mathcal{H}} x_{r,v^r,h,c^r} \cdot d_h(\pi(v^r), h, \pi(c^r)). \tag{4.9}$$

Note that this metric does not differentiate between the vSDNs. Here, no additional variables or constraints are needed; thus the average latency objective function is

$$\min\ L_{\text{avg}}. \tag{4.10}$$

Average maximum latency. The average maximum latency for a given hypervisor placement is defined as the average of all the maximum latencies of the individual vSDN requests $r\in\mathcal{R}$. First, the maximum path latency for each vSDN request r is evaluated. Second, the average of all maximum path values is evaluated: the sum of the maximum path latencies is divided by the total number of vSDN requests $|\mathcal{R}|$.

$$L_{\text{avg max}} = \frac{1}{|\mathcal{R}|} \sum_{r\in\mathcal{R}} \max_{v^r\in\mathcal{V}^r,\ h\in\mathcal{H}} x_{r,v^r,h,c^r} \cdot d_h(\pi(v^r), h, \pi(c^r)). \tag{4.11}$$

In order to circumvent the maxima over sets, we define constraints for the maximum latency of each given vSDN request $r\in\mathcal{R}$:

$$L^r_{\max} \geq x_{r,v^r,h,c^r} \cdot d_h(\pi(v^r), h, \pi(c^r)),\ \forall r\in\mathcal{R},\ \forall v^r\in\mathcal{V}^r,\ \forall h\in\mathcal{H}. \tag{4.12}$$

The objective function then minimizes the average of the $L^r_{\max}$ over all requests $|\mathcal{R}|$:

$$\min \frac{1}{|\mathcal{R}|} \sum_{r\in\mathcal{R}} L^r_{\max}. \tag{4.13}$$

This objective function provides a relaxed average latency towards a better maximum latency per vSDN. Note that this objective function differentiates between vSDNs.

Maximum average latency. The maximum average latency is defined as the maximum of the average latencies for the individual vSDNs. First, the average latency of each requested vSDN request $r \in \mathcal{R}$ is determined. Second, the maximum of these averages is evaluated, i.e.,

$$L_{\max\text{avg}} = \max_{r \in \mathcal{R}} \frac{1}{|\mathcal{V}^r|} \sum_{v^r \in \mathcal{V}^r} \sum_{h \in \mathcal{H}} x_{r,v^r,h,c^r} \cdot d_h(\pi(v^r), h, \pi(c^r)). \tag{4.14}$$

This metric corresponds to the maximum of the vSDN average latencies, i.e., the maximum latencies are relaxed per vSDN towards a better overall maximum average latency. Minimizing the maximum over the set $\mathcal{R}$ is equivalent to minimizing $L_{\max\text{avg}}$ defined through the constraints

$$L_{\max\text{avg}} \geq \frac{1}{|\mathcal{V}^r|} \sum_{v^r \in \mathcal{V}^r} \sum_{h \in \mathcal{H}} x_{r,v^r,h,c^r} \cdot d_h(\pi(v^r), h, \pi(c^r)), \ \forall r \in \mathcal{R}. \tag{4.15}$$

The objective function then minimizes $L_{\max\text{avg}}$:

$$\min L_{\max\text{avg}}. \tag{4.16}$$

4.3.1.3 Constraints

This section introduces the constraints for the k-HPP and McSDP.

Hypervisor selection constraint. We ensure that the number of placed hypervisor instances (i.e., the number of selected hypervisor nodes (locations)) is equal to k:

$$\sum_{h \in \mathcal{H}} y_h = k. \tag{4.17}$$

Virtual node path selection constraint. Each virtual node $v^r \in \mathcal{V}^r$ of each vSDN request $r \in \mathcal{R}$ must be connected to its corresponding controller c^r via exactly one hypervisor node h. This means that per virtual node v^r per request r, exactly one path has to be used:

$$\sum_{h \in \mathcal{H}} x_{r,v^r,h,c^r} = 1, \quad \forall r \in \mathcal{R}, \forall v^r \in \mathcal{V}^r. \tag{4.18}$$

Hypervisor installation constraint. We place (install) a hypervisor instance at location h (i.e., set $y_h = 1$) if at least one virtual node v^r is connected to its controller c^r via the hypervisor location h (i.e., if $x_{r,v^r,h,c^r} = 1$). At the same time, at most $\sum_{r \in \mathcal{R}} |\mathcal{V}^r|$ virtual nodes can be connected via a given hypervisor location h to their respective controllers. Thus,

$$\sum_{r \in \mathcal{R}} \sum_{v^r \in \mathcal{V}^r} x_{r,v^r,h,c^r} \leq y_h \sum_{r \in \mathcal{R}} |\mathcal{V}^r|, \quad \forall h \in \mathcal{H}. \tag{4.19}$$

Physical node to hypervisor assignment constraint. We let a hypervisor node (location) h control a physical SDN switch (network node) v^r, if a path of any request $r \in \mathcal{R}$ is selected to connect a virtual node v^r of $\mathcal{V}^r$ to its controller c^r via h (i.e., if $x_{r,v^r,h,c^r} = 1$) and additionally, this virtual node is hosted on i, i.e., $\pi(v^r) = i$. Thus:

$$x_{r,v^r,h,c^r} \leq q_{i,h}, \ \forall r \in \mathcal{R}, \ \forall v^r \in \mathcal{V}^r, \ i = \pi(v^r), \ \forall h \in \mathcal{H}. \tag{4.20}$$

Multiple hypervisors constraint. We determine the physical SDN switches $i \in \mathcal{V}$ that can be controlled by multiple hypervisors, i.e., the switches i (with $w_i = 1$) that support multiple controllers. For a given physical multi-controller SDN switch $i \in \mathcal{V}$ (with $w_i = 1$), the number of controlling hypervisors must be less than or equal to the total number of hypervisor nodes k, if the switch hosts at least one virtual SDN switch (which needs to be connected to its controller). On the other hand, for a physical single-controller SDN switch $i \in \mathcal{V}$ (with $w_i = 0$), the number of controlling hypervisors must equal one, if the switch hosts at least one virtual SDN switch. Thus, for an arbitrary physical SDN switch (node) $i \in \mathcal{V}$ (irrespective of whether i is a single- or multi-controller SDN switch), the total number of controlling hypervisor instances (locations) must be less than or equal to $[1 - w_i] + k \cdot w_i$. Thus,

$$\sum_{h \in \mathcal{H}} q_{i,h} \leq [1 - w_i] + k \cdot w_i, \quad \forall i \in \mathcal{V}. \tag{4.21}$$

We note that some solvers may unnecessarily set some $q_{i,h}$ to one for a hypervisor node h, even though network node i does *not* host any virtual node v^r that is connected to its corresponding controller c^r via hypervisor node h. This is because the solver can find a valid minimal latency solution while setting some $q_{i,h}$ unnecessarily to one. We circumvent this issue by forcing $q_{i,h}$ to zero if no corresponding path for this hypervisor instance was selected:

$$q_{i,h} \leq \sum_{r \in \mathcal{R}} \sum_{\{v^r \in \mathcal{V}^r : i = \pi(v^r)\}} x_{r,v^r,h,c^r}, \quad \forall i \in \mathcal{V}, \forall h \in \mathcal{H}. \tag{4.22}$$

Multi-controller switches constraint. We limit the number of special multi-controller SDN switches that are physically deployed in the network:

$$\sum_{i \in \mathcal{V}} w_i \leq M. \tag{4.23}$$

Note that via this constraint the four different architectures, as introduced in Sec. 4.1.1, can be modeled, optimized, and analyzed. Setting $M = 0$ forces all w_i to zero. Accordingly, there are no physical multi-controller SDN switches in the network; a physical SDN switch node can only be controlled by one hypervisor node. Thus, shared control domains, i.e., one node being controlled by multiple hypervisor nodes, are *not* possible.

4.3.2 Modeling Network Hypervisor Placement for Dynamic Use

This section introduces the decision variables, objective functions, and the constraints needed to solve the DHPP. In contrast to the static HPP, decision variables, objective functions, and constraints of the DHPP need to additionally consider the hypervisor instances Φ and the time aspect $\mathcal{T}$. The static model can be used as a basis; as a consequence, this section only introduces the additional constraints in detail. Moreover, it only briefly outlines the equations that are updated with the hypervisor instances or the time, or, if the case permits, it only shows an exemplary updated version of an equation.

Table 4.11: Binary decision variables for the DHPP with hypervisor instance differentiation. All variables represent decisions at point in time τ.

Notation	Description
$y^{\tau}_{\phi,h}$	=1, if the hypervisor instance $\phi \in \Phi$ is placed at node $h \in \mathcal{H}$ at point in time τ; $=0$, otherwise
$x^{\tau}_{r,v^r,\phi,h,c^r}$	=1, if virtual SDN node $v^r \in \mathcal{V}^r$ is connected to controller c^r via hypervisor $\phi \in \Phi$ at node $h \in \mathcal{H}$; $=0$, otherwise
$q^{\tau}_{i,h}$	=1, if physical SDN node $i \in \mathcal{V}$ is controlled by at least one hypervisor at node $h \in \mathcal{H}$; $=0$, otherwise
w^{τ}_{i}	=1, if physical SDN node $i \in \mathcal{V}$ is controlled by multiple hypervisor instances that are located on different nodes; $=0$, otherwise.
z^{τ}_{r,v^r,h,j,c^r}	$=1$ if the control path of vSDN node $v^r \in \mathcal{V}^r$ is moved from a hypervisor instance at node $h \in \mathcal{H}$ to an instance at node $j \in \mathcal{H}$.
$o^{\tau}_{r,v^r,\phi,\gamma,c^r}$	$=1$ if the control path of vSDN node $v^r \in \mathcal{V}^r$ is moved from the instance $\phi \in \Phi$ to instance $\gamma \in \Phi$

4.3.2.1 Decision Variables

To implement the DHPP differentiating hypervisor instances, six groups of binary variables are required. Four of the variables represent the hypervisor placement similar to the static case: $x^{\tau}_{r,v^r,\phi,h,c^r}$, $y^{\tau}_{\phi,h}$, $q^{\tau}_{i,h}$ and w^{τ}_{i}. The variables z^{τ}_{r,v^r,h,j,c^r} and $o^{\tau}_{r,v^r,\phi,\gamma,c^r}$ indicate reconfigurations affecting vSDN nodes at point in time τ. Table 4.11 gives an overview and explains each variable type in detail.

4.3.2.2 Objective Functions

Latency objectives. All objective functions $L_{\max}, L_{\max\text{avg}}, L_{\text{avg}\max}, L_{\text{avg}}$ as introduced for the static placement are updated to consider the hypervisor instances Φ and the time $\tau \in \mathcal{T}$. For brevity, only the updated version of the $L^{\tau}_{\max}$ (indicated by the superscript τ) is shown:

$$L^{\tau}_{\max} = \max_{r\in\mathcal{R}^{\tau},\, v^r\in\mathcal{V}^r} \sum_{\phi\in\Phi}\sum_{h\in\mathcal{H}} x^{\tau}_{r,v^r,\phi,h,c^r} \cdot d_h(\pi(v^r), h, \pi(c^r)). \tag{4.24}$$

Additional constraint:

$$L^{\tau}_{\max} \geq \sum_{\phi\in\Phi}\sum_{h\in\mathcal{H}} x^{\tau}_{r,v^r,\phi,h,c^r} \cdot d_h(\pi(v^r), h, \pi(c^r)),\ \forall r \in \mathcal{R}^{\tau},\ \forall v^r \in \mathcal{V}^r. \tag{4.25}$$

Actual objective:

$$\min L^{\tau}_{\max}. \tag{4.26}$$

Reconfiguration objectives. The metrics R^{τ}_{loc} and R^{τ}_{ins} consider location or instance changes. The metric R^{τ}_{loc} simply counts the number of vSDN nodes that are rerouted to a new hypervisor location: $R^{\tau}_{\text{loc}} = |\mathcal{Z}^{\tau}_{\text{loc}}|$. For this, it counts all $z^{\tau}_{r,v^r,h,j,c^r} = 1$ among all virtual nodes $v^r \in \mathcal{V}^r$ and their controller c^r of all consistent requests $\bar{\mathcal{R}}^\tau$ at point in time τ:

$$R^\tau_{\text{loc}} = \sum_{r \in \bar{\mathcal{R}}^\tau} \sum_{v^r \in \mathcal{V}^r} \sum_{\{h,j \in \mathcal{H}, h \neq j\}} z^\tau_{r,v^r,h,j,c^r}. \tag{4.27}$$

Actual objective:

$$\min R^\tau_{\text{loc}}. \tag{4.28}$$

Similar, the metric R^τ_{ins} considers the instance changes among all requests $\mathcal{R}^\tau$; it counts the affected virtual nodes $v^r \in \mathcal{V}^r$ whose hypervisor instances change, i.e., $o^\tau_{r,v^r,\phi,\gamma,c^r} = 1$:

$$R^\tau_{\text{ins}} = \sum_{r \in \bar{\mathcal{R}}^\tau} \sum_{v^r \in \mathcal{V}^r} \sum_{\{\phi,\gamma \in \Phi, \phi \neq \gamma\}} o^\tau_{r,v^r,\phi,\gamma,c^r} \tag{4.29}$$

Actual objective:

$$\min R^\tau_{\text{ins}}. \tag{4.30}$$

4.3.2.3 Constraints

The constraints of the DHPP are classified into four categories: the general constraints of the static setup adapted to consider instances and time, the constraints counting the amount of reconfigurations, the constraints bounding the fraction of reconfigurations, and the constraints bounding the latency.

General constraints. All constraints as introduced for the static setup are adapted to the dynamic use case as listed in Eq. 4.31-Eq. 4.38. In a nutshell, this includes allocating all k hypervisor instances, assigning every vSDN node to one hypervisor instance, connect the physical nodes $i \in \mathcal{V}$ with the hypervisor instances and limit the number of used multi-controller switches.

$$\sum_{h \in \mathcal{H}} y^\tau_{\phi,h} = 1, \qquad , \forall \phi \in \Phi \tag{4.31}$$

$$\sum_{\phi \in \Phi} y^\tau_{\phi,h} \leq 1, \qquad \forall h \in \mathcal{H} \tag{4.32}$$

$$\sum_{\phi \in \Phi} \sum_{h \in \mathcal{H}} x^\tau_{r,v^r,\phi,h,c^r} = 1, \qquad \forall r \in \mathcal{R}^\tau,\ \forall v^r \in \mathcal{V}^r \tag{4.33}$$

$$\sum_{r \in \mathcal{R}^\tau} \sum_{v^r \in \mathcal{V}^r} x^\tau_{r,v^r,\phi,h,c^r} \leq y^\tau_{\phi,h} \sum_{r \in \mathcal{R}^\tau} |\mathcal{V}^r|, \qquad \forall \phi \in \Phi,\ \forall h \in \mathcal{H} \tag{4.34}$$

$$\sum_{\phi \in \Phi} x^\tau_{r,v^r,\phi,h,c^r} \leq q^\tau_{\pi(v^r),h}, \qquad \forall r \in \mathcal{R}^\tau,\ \forall v^r \in \mathcal{V}^r,\ \forall h \in \mathcal{H} \tag{4.35}$$

$$q^\tau_{i,h} \leq \sum_{r \in \mathcal{R}^\tau} \sum_{\{v^r \in \mathcal{V}^r : \pi(v^r) = i\}} \sum_{\phi \in \Phi} x^\tau_{r,v^r,\phi,h,c^r}, \qquad \forall h \in \mathcal{H},\ \forall i \in \mathcal{V} \tag{4.36}$$

$$\sum_{h \in \mathcal{H}} q^\tau_{i,h} \leq [1 - w^\tau_i] + k w^\tau_i, \qquad \forall i \in \mathcal{V} \tag{4.37}$$

$$\sum_{i \in \mathcal{V}} x_{\mathcal{M}}(i) \leq M. \tag{4.38}$$

Reconfiguration constraints. The Eqs. 4.39-4.44 introduce the constraints tracking the amount of reconfigurations. They ensure that the reconfiguration variables z^{τ}_{r,v^r,h,j,c^r} and $o^{\tau}_{r,v^r,\phi,\gamma,c^r}$ are set properly; the variables should be pushed to 0 if no reconfiguration happens and they should be pushed to 1 if a reconfiguration happens.

As an example, Eq. 4.39 sets $z^{\tau}_{r,v^r,h,j,c^r} = 1$ only if the sum of left hand side of the equation is 2. Meanwhile, Eqs. 4.40-4.41 push the variables z^{τ}_{r,v^r,h,j,c^r} to 0 if the hypervisor instances Φ on the hypervisor locations $\mathcal{H}$ are not used at any of the points in time τ and $\tau - 1$. Eqs. 4.42-4.44 work simultaneously for the hypervisor instance changes.

$$\sum_{\phi\in\Phi}\left(x^{\tau-1}_{r,v^r,\phi,h,c^r} + x^{\tau}_{r,v^r,\phi,j,c^r}\right) - 1 \le z^{\tau}_{r,v^r,h,j,c^r}, \qquad \forall r \in \bar{\mathcal{R}}^{\tau},\ \forall v^r \in \mathcal{V}^r,\ h,j \in \mathcal{H},\ h \neq j \tag{4.39}$$

$$z^{\tau}_{r,v^r,h,j,c^r} \le \sum_{\phi\in\Phi} x^{\tau-1}_{r,v^r,\phi,h,c^r}, \quad \forall r \in \bar{\mathcal{R}}^{\tau},\ \forall v^r \in \mathcal{V}^r,\ h,j \in \mathcal{H},\ h \neq j \tag{4.40}$$

$$z^{\tau}_{r,v^r,h,j,c^r} \le \sum_{\phi\in\Phi} x^{\tau}_{r,v^r,\phi,j,c^r}, \quad \forall r \in \bar{\mathcal{R}}^{\tau},\ v^r \in \mathcal{V}^r,\ h,j \in \mathcal{H},\ h \neq j \tag{4.41}$$

$$\sum_{h\in\mathcal{H}}\left(x^{\tau-1}_{r,v^r,\phi,h,c^r} + x^{\tau}_{r,v^r,\gamma,h,c^r}\right) - 1 \le o^{\tau}_{r,v^r,\phi,\gamma,c^r}, \qquad \forall r \in \bar{\mathcal{R}}^{\tau},\ \forall v^r \in \mathcal{V}^r,\ \phi,\gamma \in \Phi,\ \phi \neq \gamma \tag{4.42}$$

$$o^{\tau}_{r,v^r,\phi,\gamma,c^r} \le \sum_{h\in\mathcal{H}} x^{\tau-1}_{r,v^r,\phi,h,c^r}, \quad \forall r \in \bar{\mathcal{R}}^{\tau},\ v^r \in \mathcal{V}^r,\ \phi,\gamma \in \Phi,\ \phi \neq \gamma \tag{4.43}$$

$$o^{\tau}_{r,v^r,\phi,\gamma,c^r} \le \sum_{h\in\mathcal{H}} x^{\tau}_{r,v^r,\gamma,h,c^r}, \quad \forall r \in \bar{\mathcal{R}}^{\tau},\ v^r \in \mathcal{V}^r,\ \phi,\gamma \in \Phi,\ \phi \neq \gamma \tag{4.44}$$

Reconfiguration bounding constraints. Restricting the number of reconfigurations makes it possible to analyze the trade-off between control plane latency and reconfigurations; if less reconfigurations are allowed, the virtualization layer may not completely adapt towards a new placement that improves the latency. The introduced relaxation parameters $\epsilon_{\text{instance}}$ and $\epsilon_{\text{location}}$ bound the total amount of reconfigurations. The following Eq. 4.45 and Eq. 4.46 use the relaxation parameters; they bound the total amount of location or instance changes among all vSDN requests $\mathcal{R}^{\tau}$. The right hand sides of the equations provide the maximum possible number of changes; the maximum number is bounded by the total number of all vSDN nodes of persistent requests $\bar{\mathcal{R}}^{\tau}$. Multiplying this number with $\epsilon_{\text{instance}}$ or $\epsilon_{\text{location}}$ restricts the number of R^{τ}_{ins} or R^{τ}_{loc} to be between $[0, \sum_{r \in \bar{\mathcal{R}}^{\tau}} |\mathcal{V}^r|]$.

$$R^{\tau}_{\text{loc}} \leq \epsilon_{\text{location}} \cdot \left[\sum_{r \in \bar{\mathcal{R}}^{\tau}} |\mathcal{V}^r| \right] \tag{4.45}$$

$$R^{\tau}_{\text{ins}} \leq \epsilon_{\text{instance}} \cdot \left[\sum_{r \in \bar{\mathcal{R}}^{\tau}} |\mathcal{V}^r| \right] \tag{4.46}$$

Latency bounding constraints. Vice versa to bounding the reconfigurations, relaxing or restricting the latency via a constraint makes it possible to analyze the latency-to-reconfiguration trade-off. The following equation describes the constraint:

$$L_{obj} \leq (1 + \epsilon_{\text{latency}}) L'_{obj} \tag{4.47}$$

As previously described, the optimal latency L'_{obj} for a given objective and the relaxation parameter $\epsilon_{\text{latency}}$ are used as inputs; note that L'_{obj} needs to be pre-determined. Relaxing the latency should decrease the amount of needed reconfigurations; intuitively, with a higher acceptable latency compared to the optimal latency, the hypervisor instances remain at their old locations.

4.4 Simulative Analysis of Hypervisor Placement Problems

This section investigates the hypervisor placement for static and dynamic traffic scenarios. The number of vSDNs is constant in a static traffic scenario. Furthermore, the vSDNs themselves do not change; tenants do not add or delete vSDN nodes neither do they request any further adaptation of their vSDN requests like changing the positions of their controllers.

In the dynamic traffic scenario, the virtualization layer faces vSDN requests arriving and leaving the substrate network over time: the number of vSDN tenants changes. The analysis demonstrates that the virtualization layer needs to adapt in order to always provide the best possible latency guarantees.

4.4.1 Simulator for CPP, HPP, DHPP and VNE: *virtshouse*

The Python-based simulation framework *virtshouse* has been implemented for all subsequent studies [BBZ+15; BBZ+16; BKS+16; BKS+17] of this thesis: (virtual) controller placement, hypervisor placement, and virtual network embedding. The framework uses Gurobi [GO16] to solve MILP-based problem formulations and algorithms relying on relaxed linear program formulations. In this chapter, the evaluation focuses mainly on the latency and reconfiguration analysis, i.e., the hypervisor placement (HP) latency values (defined in 4.3.1.2), the latency values of the individual vSDN requests (see Sec. 4.4.2.2, Eq. 4.48 and Eq. 4.49), and the different types of reconfigurations (introduced in Sec. 4.2.2.2). Following [HSM12], the hypervisor architectures and objectives are analyzed when deployed on the real SDN-based OS3E network topology [Os3], topologies taken from the Topology Zoo (TZ) data set [KNF+11], or topologies from the SNDLib [OWP+10].

Table 4.12: Evaluation settings for $k = 1$ hypervisor.

Parameter	**Values**
Number of vSDNs $\|\mathcal{R}\|$	$1, 10, 15, 20, 25, 40, 60, 80$
Number of virtual nodes per vSDN $\|\mathcal{V}^r\|$	Uniformly distributed 2...10
Controller location for each vSDN c^r	Uniformly selected from set $\mathcal{V}$
Virtual node locations $\pi(v)$	Uniformly selected from $\mathcal{V}$ without replacement
Runs per model	≥ 30
HPP objectives	$L_{\max}$, L_{avg}, $L_{\text{avg max}}$, $L_{\text{max avg}}$

4.4.2 Placement Results for Static Use and the Cost of Virtualization

The first part (Sec. 4.4.2.1) of this section analyzes the deployment of one ($k = 1$) hypervisor on a real network topology. Analyzing one hypervisor provides basic insights for the multiple hypervisor scenario; it illustrates the impact of the hypervisor latency objective, the number of vSDNs, and the vSDN objective. The second part (Sec. 4.4.2.2) then extends the analysis to multiple hypervisors. It focuses on the potential benefit of multiple hypervisors and it questions whether the observations from one topology can be generalized for other topologies. All latency results will be given in milliseconds [ms].

4.4.2.1 Analyzing the Effect of one ($k = 1$) Hypervisor

The parameter settings are given in Table 4.12. The network topology is the ATT North America topology (ATTMpls) [KNF+11], which consists of 25 nodes and 36 edges. We evaluate the HPP for different numbers $|\mathcal{R}|$ of vSDNs; a physical node thus hosts a variable number of vSDN switches, possibly also 0. We call this the *vSDN network density* of the virtualization layer. For example, for $|\mathcal{R}| = 1$ only one virtual SDN network is considered during the hypervisor placement while $|\mathcal{R}| = 80$ means that 80 vSDNs are considered. For all vSDN requests $r \in \mathcal{R}$, we assume that the virtual nodes and their controllers are given as input. The number of virtual nodes $|\mathcal{V}^r|$ per request r is uniformly distributed between 2 and 10. The vSDN node locations are uniformly selected from all available physical network locations $\mathcal{V}$. For each request, the controller location c^r is also uniformly selected from all physical network locations $\mathcal{V}$.

Is there a dominant location for the network hypervisor under different objectives? We start by comparing the different objectives with respect to the geographical locations of the hypervisor nodes. Figure 4.5 shows the resolved hypervisor locations for each objective metric in case $|\mathcal{R}| = 80$. The size of a circle of a node represents the frequency of how often the node is selected as the optimal hypervisor location over 30 runs. A physical node location that is colored white with the smallest possible size indicates that the node is never selected as hypervisor location.

Figure 4.5a depicts the results for minimizing $L_{\max}$: the hypervisor location converges to the network node 15 lying in the sparse part of the network. The longer links connecting between the east

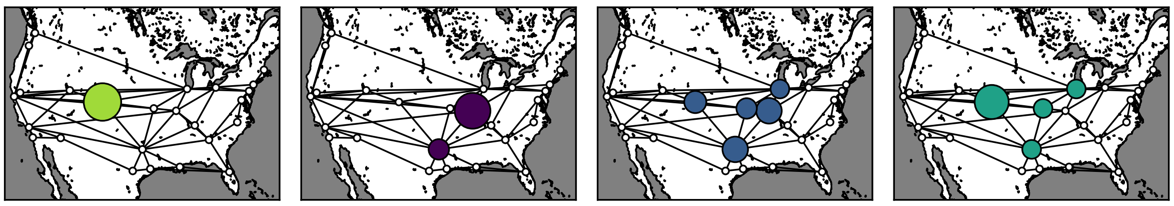

(a) min $L_{\max}$. (b) min L_{avg}. (c) min $L_{\text{avg max}}$. (d) min $L_{\text{max avg}}$.

Figure 4.5: Hypervisor placement distribution over ATTMpls network topology. A node size indicates how often this node is considered as a hypervisor location among all simulation runs. Simulation settings are $k = 1, |\mathcal{R}| = 80$. One figure per hypervisor latency objective. For the given settings, hypervisors are located only on a subset of nodes. The new virtual objectives show with 4-5 nodes a higher distribution among potential substrate locations.

and west parts of the network determine the hypervisor locations. The maximum latency of the longer links are mostly impacting the objective function.

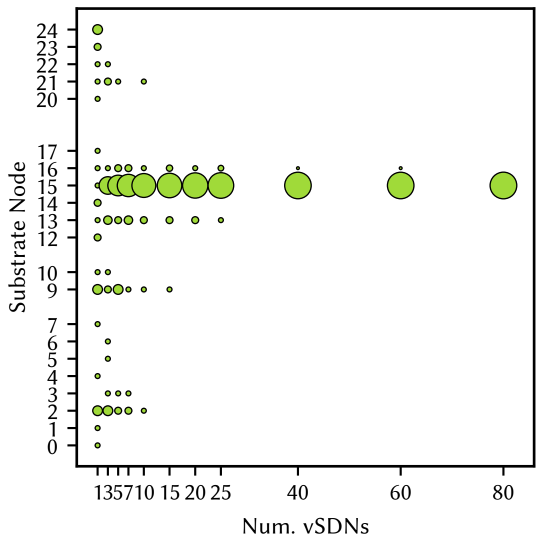
Figure 4.5b shows two dominant hypervisor locations towards the center of the network (node 9 and 13). In contrast to solving the model for the maximum latency, minimizing the average latency considers all controller to virtual node connections of all vSDNs. As the vSDNs are randomly distributed, the hypervisor locations converge to the more populated part of the network, i.e., the east of the network.

Figure 4.5c illustrates the hypervisor locations with a minimum $L_{\text{avg max}}$. As this model considers the average of the maximum latency of all vSDNs, more physical locations (node 9, 12, 13, 15, 16) for the hypervisor are optimal among the 30 runs. It can be observed that the hypervisor locations are overlapping with the locations as shown in Figure 4.5b.

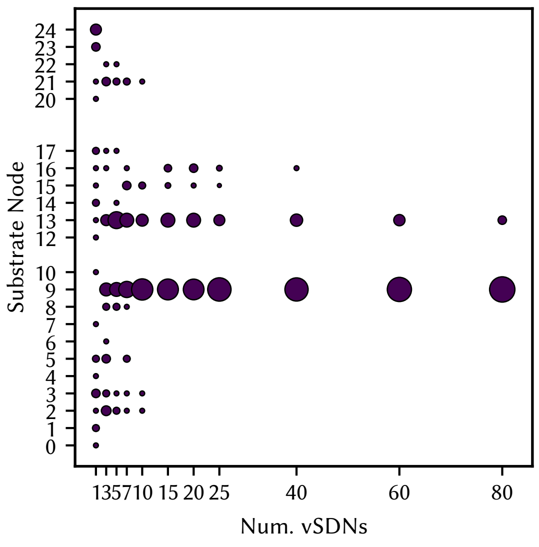
Finally, minimizing $L_{\text{max avg}}$ as shown in Figure 4.5d considers the maximum of the average latency within each vSDN. This model results again in more candidate locations for the hypervisor (node 2, 9, 11, 12, 13, 15, 16). Due to again minimizing the maximum among all average vSDN latencies, the dominant hypervisor locations overlap with the locations shown in Figure 4.5a; vSDNs with longer paths mainly determine the latency $L_{\text{max avg}}$.

On the basis of the points mentioned above, it is apparent that only a subset of nodes are resolved as candidate hypervisor locations under the given input parameters: the potential hypervisor locations concentrate on a small subset of physical locations for each objective. Such knowledge can be used when targeting search space reduction mechanisms as analyzed in Chapter 5. Due to efficient search space reduction mechanisms, an optimization needs to consider only a subset of nodes when determining a new placement. However, $L_{\text{avg max}}$ or $L_{\text{max avg}}$ demand a higher optimization effort as more nodes are providing an optimal solution for the hypervisor location.

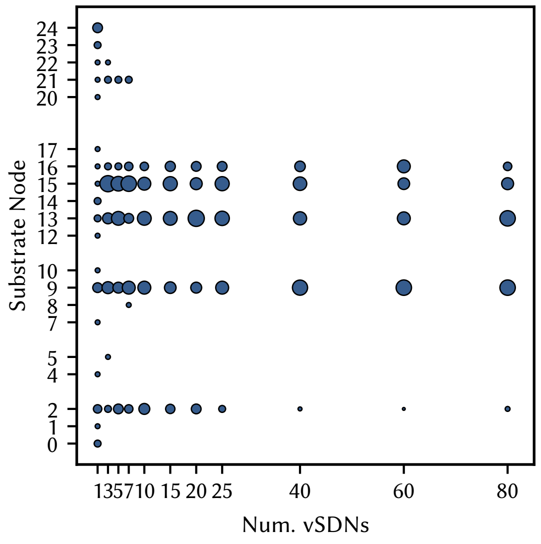
How does the vSDN network density influence the hypervisor placement? Figure 4.6 shows the impact of the vSDN density, i.e., number $|\mathcal{R}|$ of vSDNs. While the x-axis shows the vSDN density, the y-axis shows the numbers of the network nodes. Again, the size of the circles indicate how often a location is chosen among 30 or more runs.



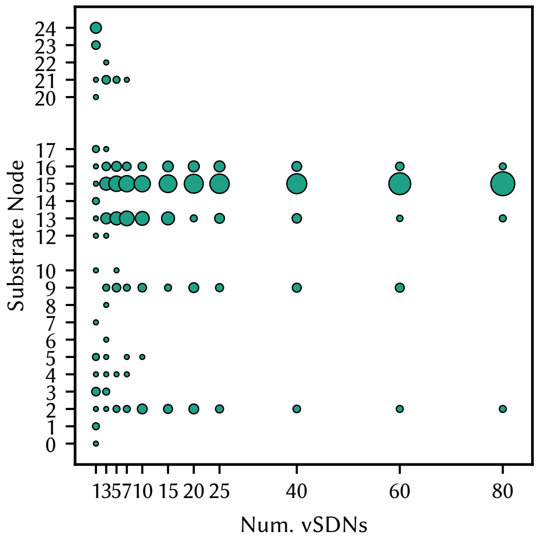
(a) min $L_{\max}$.



(b) min L_{avg}.



(c) min $L_{\text{avg max}}$.



(d) min $L_{\text{max avg}}$.

Figure 4.6: Importance of substrate nodes as hypervisor locations over number of vSDNs. One figure per hypervisor objective. Figures indicate how often a node location was chosen relatively for all simulation runs of one setting. The size of a circle indicates the frequency how often a location was chosen. A setting is defined by the number of vSDNs. The x-axis gives the number of vSDNs and the y-axis shows the substrate node ids. More vSDNs force the hypervisor to dominant locations: e.g., substrate node 15 is dominating for $L_{\max}$ when more than 25 vSDNs are hosted on the substrate network.

In case of a single vSDN network ($|\mathcal{R}| = 1$), most physical nodes are selected at least one time as a hypervisor location within the 30 runs. This can be explained by the fact that just one vSDN network with 7 nodes on average is placed randomly among the network. This single vSDN determines the hypervisor location for all metrics; actually, as the hypervisor serves only one vSDN, it could be co-located with the tenant controller, which achieves the lowest latency. Accordingly, more potential hypervisor locations are optimal for each individual run. With an increasing vSDN density, the solutions converge to deterministic hypervisor location(s): more vSDNs are generated with

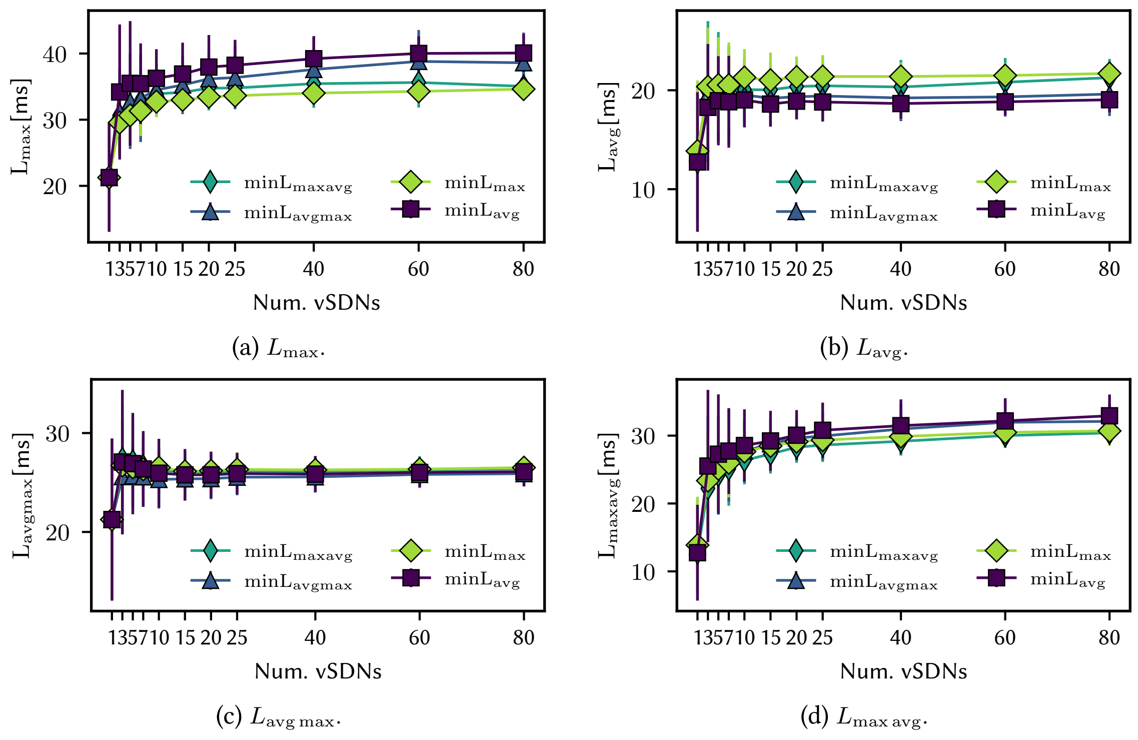


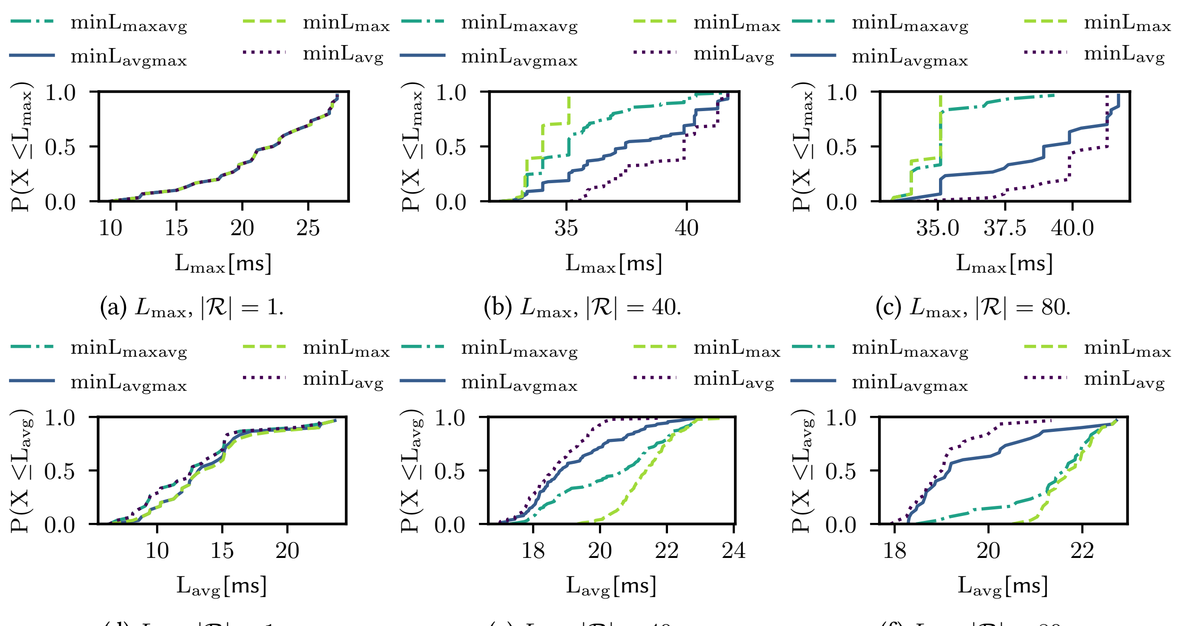
Figure 4.7: Hypervisor latency metric over number of vSDNs for all objectives; one figure per hypervisor latency metric (not objective). Each objective achieves the optimal solution with respect to its targeted metric. An increasing number of vSDNs increases the achieved latencies. Metrics converge between 25 and 40 vSDNs; adding vSDNs does not increase latencies further.

a possibly large geographical distribution. Since these vSDNs are spread with a higher probability among the physical network, the topology of the physical network determines an upper bound for the latencies. For instance, the long links from east to west determine the hypervisor locations when minimizing $L_{\max}$.

To conclude, the vSDN network density has to be considered for the hypervisor placement: a high density actually leads to a small subset of valuable nodes. This observation can be used to predetermine a set of potential hypervisor locations. For instance, in case the number of vSDNs is varying around a high density value, the hypervisor might potentially move only among the subset of nodes. As a result, the execution cost of online adaptation algorithms might be decreased.

What are the trade-offs between the objective metrics? In order to show the trade-offs between the four objective metrics in more detail, we calculate the values of the other three metrics for each optimization of an objective accordingly. Figure 4.7 shows the average of all metrics with 95 % confidence intervals. In general, optimizing the model for a particular metric also leads to the best solution with respect to the metric. For instance, optimizing $L_{\max}$ achieves the lowest L_{max} value, whereas minimizing L_{avg} achieves the best L_{avg} solution; a numerical indicator for the correctness of the proposed model.

In Figure 4.6 we have observed that the sets of selected hypervisor locations for different objectives, e.g., L_{avg} and $L_{\text{avg}\max}$, involve the same network node 9: the solutions of both objectives



(a) $L_{\max}$, $|\mathcal{R}| = 1$. (b) $L_{\max}$, $|\mathcal{R}| = 40$. (c) $L_{\max}$, $|\mathcal{R}| = 80$.

(d) L_{avg}, $|\mathcal{R}| = 1$. (e) L_{avg}, $|\mathcal{R}| = 40$. (f) L_{avg}, $|\mathcal{R}| = 80$.

Figure 4.8: Cumulative distribution functions of individual latency values for all hypervisor objectives. Each subfigure shows one vSDN setting (1,40,80). The latency objectives $L_{\max\text{avg}}$ and $L_{\text{avg}\max}$ trade-off the maximum or average latency. For instance, $L_{\max\text{avg}}$ worsens the maximum latency towards an improved average latency.

overlap. As a result of the geographically overlapping locations, Figure 4.7 shows the same behavior with respect to all metric values. As an example, Fig. 4.7a shows that $\min L_{\max\text{avg}}$ achieves the second best performance for $L_{\max}$. At the same time, it also improves L_{avg} compared to $\min L_{\max}$ (Fig. 4.7b) - minimizing $L_{\max\text{avg}}$ worsens $L_{\max}$ in order to improve L_{avg}.

To investigate this behavior in more detail, Figures 4.8a to 4.8f show the cumulative distribution function for L_{max} and L_{avg} for all objectives. We compare the behavior for $|\mathcal{R}| = 1$, $|\mathcal{R}| = 40$, and $|\mathcal{R}| = 80$ between each objective.

For $|\mathcal{R}| = 1$, all models have the same latency results for the maximum latency ($L_{\max}$) as shown in Figure 4.8a. This means that for all models the hypervisor location is placed on the path having the maximum latency for the particular vSDN network. For the average latency, as illustrated in Fig. 4.8d, optimizing for average or maximum average latency leads to better results. Note in particular the behavior of $L_{\max\text{avg}}$: since only one virtual network exists, the average latency of this vSDN determines the maximum latency of $L_{\max\text{avg}}$. Both L_{avg} and $L_{\max\text{avg}}$ optimize for all paths and do not stop when the minimal maximum latency is reached.

Fig. 4.8b and Fig. 4.8e already show a trade-off between the models for $|\mathcal{R}| = 40$. In particular for the objectives maximum latency and average latency, a clear gap exists. In contrast to $|\mathcal{R}| = 1$, optimizing for multiple vSDNs leads to different effects. In detail, optimizing for maximum and maximum average as well as optimizing for average and average maximum show results that are close together with respect to $L_{\max}$ and L_{avg}.

For $|\mathcal{R}| = 80$, as shown in Figures 4.8c and 4.8f, a clear trade-off between optimizing for maximum

Table 4.13: Evaluation settings of static analysis.

Parameter	**Values**
Network topology	Abilene ($\lvert\mathcal{V}\rvert = 11, \lvert\mathcal{E}\rvert = 14$), Quest ($\lvert\mathcal{V}\rvert = 20, \lvert\mathcal{E}\rvert = 31$), OS3E ($\lvert\mathcal{V}\rvert = 34, \lvert\mathcal{E}\rvert = 42$)), Bellcanada ($\lvert\mathcal{V}\rvert = 48, \lvert\mathcal{E}\rvert = 64$), Dfn ($\lvert\mathcal{V}\rvert = 51, \lvert\mathcal{E}\rvert = 80$)
Ratio of multi-controller switches M_r	0, 0.25, 0.5, 0.75, 1
Number of $\lvert\mathcal{R}\rvert$	1, 3, 5, 7, 10, 15, 20, 40, 80
Number of $\lvert\mathcal{V}^r\rvert$ per vSDN	Uniformly distributed $2, \ldots, 10$
Number of c^r per request r	1
Virtual SDN controller placements (CP)	Random (rnd), min average latency (avg), min maximum latency (max); for avg and max controller placement is solved optimally a priori
Virtual node locations $\pi(v^r)$	Uniformly selected from set $\mathcal{V}$
Hypervisor placement objectives	$L_{\max}, L_{\max\text{avg}}, L_{\text{avg}\max}, L_{\text{avg}}$
Runs per setup	≥ 30

or maximum average and average or average maximum can be observed. Furthermore, the resulting latencies are varying less for $\lvert\mathcal{R}\rvert = 80$ than for $\lvert\mathcal{R}\rvert = 40$. In particular for $L_{\max}$, the optimization leads to a small number of latency values, as indicated by the vertical process of the CDF. We conclude that the trade-offs between the optimizations of the different metrics depend on the density of the vSDNs: their number and their locations are the dominant impact factor. Furthermore, with increasing density, the values of the metrics are varying less among all objective optimizations; the hypervisor control plane latency varies less when more virtual networks are hosted on the infrastructure.

4.4.2.2 Analyzing the Effect of Multiple ($k \geq 1$) Hypervisors

Many factors have already been identified that impact the performance of a virtualization layer consisting of a single hypervisor instance: mainly the number of vSDNs and the objective. In this section, the initial analysis is extended towards multiple hypervisor instances. Since multiple hypervisor instances open further options, e.g., the use of multi-controller switches, more aspects need to be analyzed. Operators require such detailed analysis in order to always choose the best option according to the requirements of their customers. The evaluation settings are summarized in Table 4.13. We analyze the k-HPP for the OS3E topology [Os3]: a topology for networking science having deployed SDN located in North America. Hence, the results with respect to latency can be seen comparable to the ones of the previous study on the ATTMpls topology.

For all architectures, we assume that all network nodes can host a hypervisor node, i.e., $\mathcal{H} = \mathcal{V}$. The number of hypervisor nodes k and the number of multi-controller switches M determine the type of hypervisor architecture. The centralized architecture, see Section 4.1.1 and Fig. 4.1a, is characterized by $k = 1$ and $M = 0$, i.e., each switch has only one controller (hypervisor) connection. Note also that $k > 1$ and $M = 0$ corresponds to the distributed architecture operating on single-controller switches (see Fig. 4.1b), whereas $1 < M < |\mathcal{V}|$ corresponds to the hybrid architecture (Fig. 4.1d) and $M = |\mathcal{V}|$ represents the distributed architecture where only multi-controller switches are deployed (cf. Fig. 4.1c).

The ratio $M_r = M/|\mathcal{V}| = 0, 0.25, 0.5, 0.75, 1$ defines M; the ratio specifies the maximum number of network nodes supporting the multi-controller feature. For instance, $M_r = 0.5$ corresponds to $M = 17$ multi-controller switches that can be placed inside the OS3E network. We initially compare all four SDN network hypervisor architectures in terms of the hypervisor latency metrics defined in Section 4.3.1.2. Subsequently, we analyze the latency values of the vSDN requests in order to evaluate the impact of virtualization. We then show through a generalized topology-aware analysis how the architectures behave for different network topologies.

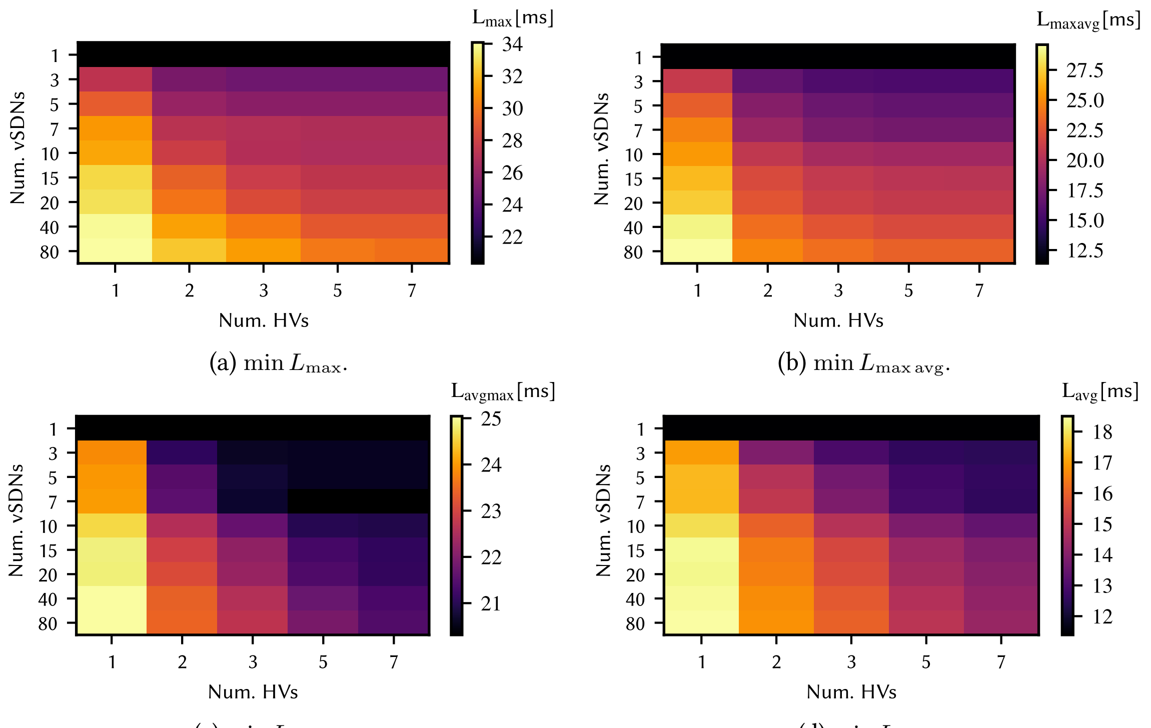
Impact of Hypervisor Placement (HP) on latency metrics. We first present and discuss a compact representation of the results for varying number of vSDN requests $|\mathcal{R}|$ and increasing number of hypervisor instances k in Fig. 4.9. Based on our observations we then conduct a more detailed evaluation of selected setups in Fig. 4.10 to clearly illustrate the effects of different architecture attributes, namely multi-controller switches, number of hypervisor instances k, and controller placements (CPs). In order to evaluate the virtualization overhead, i.e., the cost of virtualization, in terms of additional control plane latency, we conclude the OS3E evaluation by investigating the individual request latencies of the vSDN requests in Figs. 4.12–4.14. Finally, we provide an analysis of five different substrates in Figs. 4.15–4.16 to assess how our observations may be generalized.

Severe impact of number of vSDN requests and hypervisor instances on HP latency metrics. Figures 4.9a–d provide a compact representation of the HP latency metrics for every combination of number of hypervisors k and number of vSDN requests $|\mathcal{R}|$. We consider the random CP strategy in order to focus on the impact of the parameters k and $|\mathcal{R}|$. The figures show heatmaps of the latency values averaged over at least 30 independent runs. The lowest latency value is represented in black color and the highest latency value in bright yellow color. Red represents intermediate latency values.

When only a single vSDN is considered ($|\mathcal{R}| = 1$), increasing the number of hypervisor instances k does not reduce any of the resulting latency metrics. When only a single hypervisor instance is considered ($k = 1$), the latencies are significantly increasing with an increasing number of vSDNs $|\mathcal{R}|$. On the other hand, for multiple requested vSDNs ($|\mathcal{R}| > 1$), we observe from Fig. 4.9 that increasing the number of hypervisor instances k generally reduces the latencies.

The number of requested vSDNs $|\mathcal{R}|$ plays an important role when optimizing the HP. For small $|\mathcal{R}|$, a small number of hypervisor instances k suffices to achieve optimal placements. In order to investigate the impact of k, M (M_r), and the CP in more detail, we set $|\mathcal{R}| = 40$ for the subsequent evaluations as this setting has shown a clear effect of increasing k on the HP latencies.

Increasing the number of hypervisor instances k minimizes latency metrics differently. Figures 4.10a-4.10d show the impact of the number of hypervisors k, the number of multi-controller switches M, and the CPs on the achieved latencies. Each figure shows the result of one HP objective. Further-



(a) min $L_{\max}$.

(b) min $L_{\max\,\mathrm{avg}}$.

(c) min $L_{\mathrm{avg}\,\max}$.

(d) min L_{avg}.

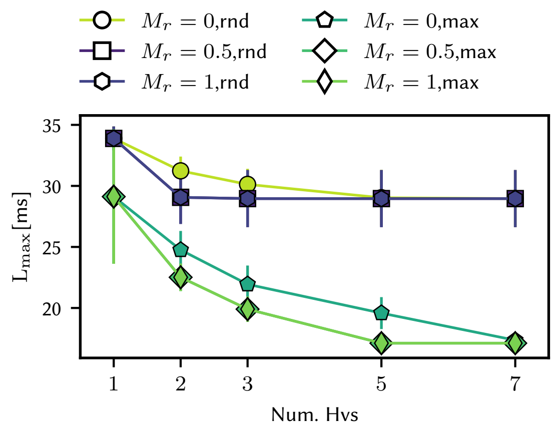
Figure 4.9: Heatmaps show the latency values (in milliseconds [ms]) averaged over at least 30 independent runs. Light yellow represents high latency values, while black represents low latency values. For each subfigure, the numbers of vSDN requests $|\mathcal{R}|$ are indicated on the left, the numbers of hypervisor instances k (Num. HVs) on the bottom, and the heatmap scale for the latencies on the right. Fixed param.: no multi-controller switches $M_r = 0$, random controller placement (CP).

more, the random CP is compared to the best CP, i.e., either average or maximum CP, which achieved the best results in the conducted simulations.

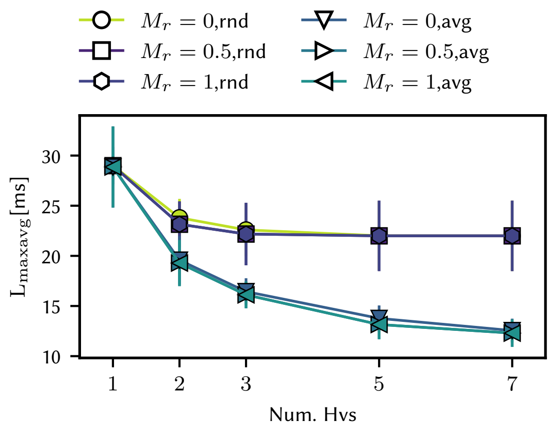
We observe from Figs. 4.10a-4.10d that additional hypervisor instances generally reduce the latency objectives for all setups. This decrease of latencies with increasing k is consistent with the observations from Figs. 4.9, which considered increasing k for a range of numbers of vSDN requests $|\mathcal{R}|$ (and $M_r = 0$). Notice in particular the continuous drop of L_{avg} in Fig. 4.9d.

However, we also observe from Figs. 4.10a-4.10d that for increasing k there is typically a point of diminishing returns, where adding hypervisor instances does not further reduce the latency. This point of diminishing returns varies according to latency objective and CP. For instance, the point of diminishing returns ranges from $k = 2$ for random CP with the $L_{\max}$ objective and $M_r = 1$ (Fig. 4.10a), to $k = 7$ for L_{avg} (Fig. 4.10d). That is, the convergence point differs strongly among the setups. Thus, in case of changing the operation goal of a hypervisor deployment, e.g., for $M_r = 0$ from $L_{\max\,\mathrm{avg}}$ to $L_{\mathrm{avg}\,\max}$, a re-optimization of the HP may be necessary as a different number k of hypervisors may be needed for achieving an optimal latency value (e.g., from $k = 5$ for $L_{\max\,\mathrm{avg}}$ to $k = 7$ for $L_{\mathrm{avg}\,\max}$ with random CP).

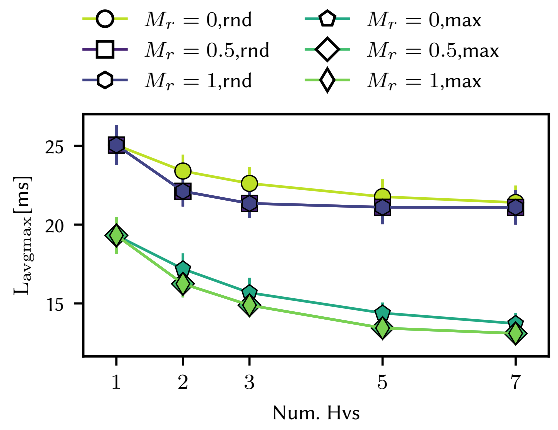
More multi-controller switches demand less hypervisor instances for an optimal solution. Fig. 4.10 also shows that the objectives benefit from multi-controller switches. This means that increasing the number of multi-controller switches M (M_r) decreases the number of hypervisor instances k re-



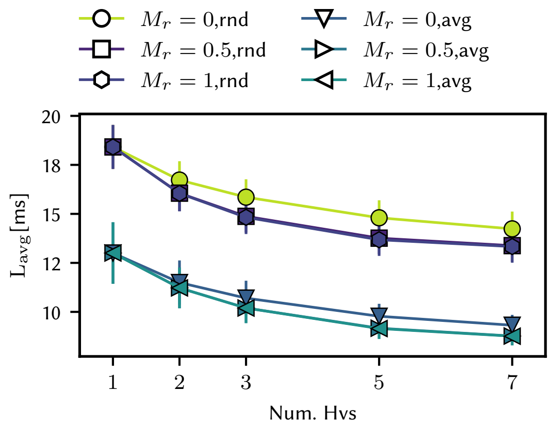
(a) min $L_{\max}$.



(b) min $L_{\max\text{avg}}$.



(c) min $L_{\text{avg}\max}$.



(d) min L_{avg}.

Figure 4.10: Latency values (95 % confidence intervals over 30 runs, in milliseconds [ms]) obtained with the different latency minimization objectives $L_{\max}$, $L_{\max\text{avg}}$, $L_{\text{avg}\max}$, and L_{avg} as a function of number of hypervisor instances k. The number of multi-controller switches is $M = 0$, $M = 17$, and $M = 34$. The controller placement (CP) strategies are random and maximum for $L_{\max}$ and $L_{\max\text{avg}}$ and random and average for L_{avg} and $L_{\text{avg}\max}$. Different markers indicate the different settings for all objectives. Deploying multi-controller switches and optimizing CP a priori improves achieved average latencies for all objectives. Latency values converge for 5 hypervisor instances.

quired for an optimal solution - the point of diminishing returns is reached earlier. For instance, $L_{\max}$ (Fig. 4.10a) achieves for $k = 2$ hypervisor instances in combination with $M_r = 0.5$ or 1.0 the same latency as with $k = 5$ hypervisor instances without any multi-controller switches $M_r = 0$: $L_{\max}$ can save 3 hypervisor instances due to the deployment of multi-controller switches. L_{avg} shows a more significant benefit of multi-controller switches over all k (Fig. 4.10d): for both CP strategies, there is always a gap between $M_r = 0$ and $M_r = 0.5$ or 1. Using multi-controller switches can always reduce the hypervisor control plane latency for L_{avg}. To conclude, with respect to all objectives, only 50 % of switches need to support the multi-controller feature in order to achieve an optimal HP, as it is shown by the overlapping lines of $M_r = 0.5$ and $M_r = 1$.

The best controller placement strategy depends on the hypervisor latency objective. Fig. 4.10 indicates that an optimized controller placement significantly decreases the values of all latency metrics, in some cases by more than 50 %. For instance, for the objective $L_{\max}$, the latency is reduced by nearly

42 % from an average value of 30 ms to 18 ms (Fig. 4.10a). The optimized CP also improves the centralized architecture ($k = 1$) for the $L_{\max}$, L_{avg}, and $L_{\text{avg}\max}$ objectives. For $L_{\max\text{avg}}$, however, an optimized CP does not significantly reduce the latency of the centralized architecture ($k = 1$). Furthermore, the best CP strategy depends on the HP objective. The maximum CP achieves the most pronounced latency reduction for the $L_{\max}$ and $L_{\text{avg}\max}$ latency objectives. For L_{avg} and $L_{\max\text{avg}}$, the average CP shows the best performance improvement.

The average/maximum controller placements demand more hypervisors for an optimal solution. In addition to reducing the latency values in general, the maximum and average controller placements demand less hypervisor instances k (Fig. 4.10) to achieve optimal placements. Also, the number of multi-controller switches M impacts the convergence point per HP objective. For the $L_{\max\text{avg}}$, $L_{\text{avg}\max}$, and L_{avg} objectives (Fig. 4.10b,Fig. 4.10c, and Fig. 4.10d), there is a small gap between $M_r = 0$ and $M_r = 1$. However, for $L_{\max}$ (Fig. 4.10a), there is a pronounced gap between $M_r = 0$ and $M_r = 1$; and only for $k = 7$ hypervisor instances do the $M_r = 0$ and $M_r = 1$ curves converge. For the $L_{\max\text{avg}}$ objective, the convergence point is also only reached for $k = 7$ hypervisor instances. When comparing all latency values for $k = 1$, only $L_{\max\text{avg}}$ benefits neither from an optimized controller placement nor from multi-controller switches. This effect can be explained by the examination of the individual latencies of the vSDN requests, as investigated next.

Analysis of the vSDN requests' control plane latencies—The Cost of Virtualization. Before analyzing the impact of the HP on the individual vSDN requests, we first examine the impact of the CP on the individual requests without virtualization. This means that we calculate for each request the best possible latency values, which are determined by the CP. Without virtualization, the connections between the requested switches and controllers do not have to pass through any hypervisor instance. We define the maximum request latency

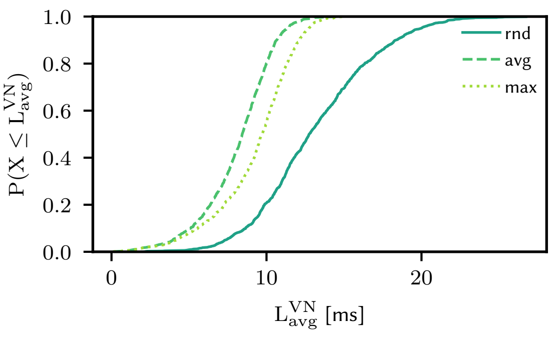
$$L_{\max}^{VN,CP}(r) = \max_{v^r \in \mathcal{V}^r} d(\pi(v^r), \pi(c^r)),\ \forall r \in \mathcal{R} \tag{4.48}$$

and the average request latency

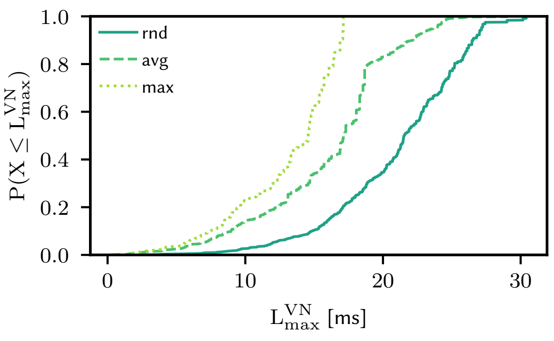
$$L_{\text{avg}}^{VN,CP}(r) = \frac{1}{|\mathcal{V}^r|} \sum_{v^r \in \mathcal{V}^r} d(\pi(v^r), \pi(c^r)),\ \forall r \in \mathcal{R}. \tag{4.49}$$

Note that these are the definitions of the request latencies without any virtualization. For calculating the latencies with virtualization $L_{\text{avg}}^{VN,HP}(r)$ and $L_{\max}^{VN,HP}(r)$, the function $d(\pi(v^r), \pi(c^r))$ denoting the distance without intermediate node (hypervisor) needs to be replaced by the function $d_h(\pi(v^r), h, \pi(c^r))$, which denotes the distances of the paths via the used hypervisor instances. We omit the request specification '(r)' in the following to avoid notational clutter.

Fig. 4.11 shows the $L_{\text{avg}}^{VN,CP}$ and $L_{\max}^{VN,CP}$ CDFs for the random, average, and maximum CPs without virtualization (i.e., no HP). In general, they show the best possible request latencies that can be achieved for each request. Virtualization, i.e., hypervisor placement, will achieve in the best case the latency values as shown by the figures. The average CP achieves the lowest latency values for $L_{\text{avg}}^{VN,CP}$ (Fig. 4.11a), while the maximum CP achieves the lowest latencies for $L_{\max}^{VN,CP}$ (Fig. 4.11b). Interestingly, the results of the maximum CP are close to the average CP for $L_{\text{avg}}^{VN,CP}$. The reason is that the maximum CP places the controller in the middle of the longest path between two virtual

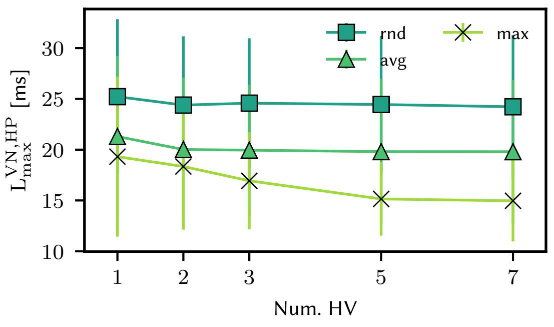


(a) Average request latency.

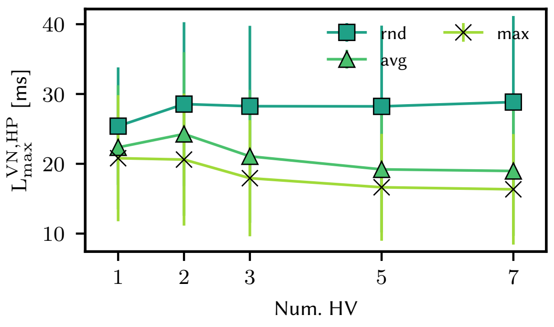


(b) Maximum request latency.

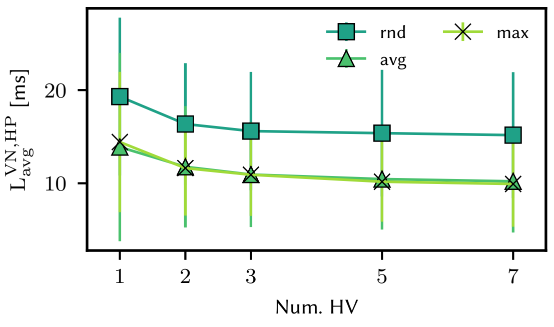
Figure 4.11: Cumulative distribution functions of average ($P(X \leq L_{\text{avg}}^{VN,CP})$) and maximum ($P(X \leq L_{\text{max}}^{VN,CP})$) latencies for direct virtual switch to controller connections of individual requested vSDNs $r \in \mathcal{R}$, without traversing hypervisors. The controller placement (CP) strategies are: random (solid line), average (dashed line), and maximum (dotted line).



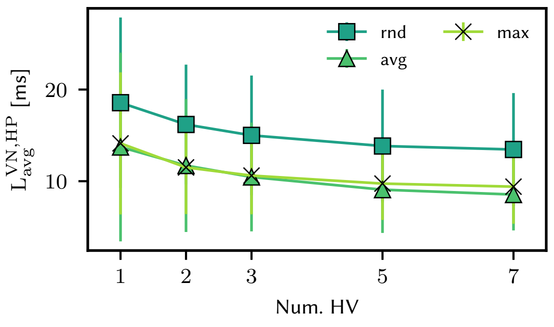
(a) min L_{max}, req. lat. $L_{\text{max}}^{VN,HP}$.



(b) min $L_{\text{max avg}}$, req. lat. $L_{\text{max}}^{VN,HP}$.



(c) min $L_{\text{avg max}}$, req. lat. $L_{\text{avg}}^{VN,HP}$.



(d) min L_{avg}, req. lat. $L_{\text{avg}}^{VN,HP}$.

Figure 4.12: Mean values with 95 % confidence intervals of average ($L_{\text{max}}^{VN,HP}$) and maximum ($L_{\text{avg}}^{VN,HP}$) latencies for VCP connections of individual vSDNs $r \in \mathcal{R}$. For each HP latency minimization objective, the impact of k hypervisor instances and the controller placement (CP) are depicted: random CP (boxes), average CP (triangles up), and maximum CP (crosses). Fixed parameters: $M_r = 0.5$ multi-controller switches, 40 vSDNs.

SDN switches to reduce $L_{\text{max}}^{VN,CP}$. In most case, this is a central position of the vSDN, which leads also to low $L_{\text{avg}}^{VN,CP}$ values.

Figure 4.12 shows the impact of CPs and the number of hypervisor instances k on the request latencies $L_{\text{max}}^{VN,HP}$ and $L_{\text{avg}}^{VN,HP}$. Each figure shows the behavior for a given HP objective. For distributed architectures ($k > 1$), we set the number of multi-controller switches to $M = 17$ ($M_r = 0.5$) as the hybrid architecture has already optimal HP latency values. To begin with, we observe from Fig. 4.12a that the maximum CP achieves the lowest maximum request latencies $L_{\max}^{VN,HP}$ while the average CP achieves the lowest average request latencies $L_{\text{avg}}^{VN,HP}$ (Fig. 4.12d). Another simulation-based proof of correctness of the proposed models.

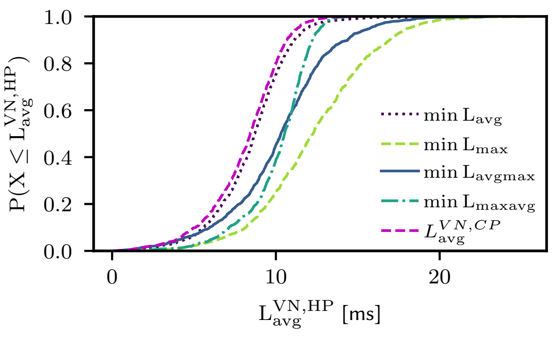
Adding hypervisor instances may increase the virtual request latency with maximum-based objectives ($L_{\max}$ and $L_{\max\text{avg}}$). For the maximum-based latency objectives $L_{\max}$ and $L_{\max\text{avg}}$, which consider the maximum or the maximum of the average vSDN (request) latencies (see Eqn. (4.14)), Fig. 4.12a and Fig. 4.12b show interesting working behaviors. Whereas the maximum CP achieves generally the lowest individual maximum request latencies $L_{\max}^{VN,HP}$, additional hypervisor instances may increase the request latencies in case of an average CP. This is because both maximum-based latency objectives strive to minimize the maximum path latency or the maximum average latency over all requested vSDNs (see Eqn. (4.6)). For this, they relax the maximum request latency $L_{\max}^{VN,HP}$(r) and average request latency $L_{\text{avg}}^{VN,HP}$(r) for some vSDN requests in order to improve the maximum latency over all requests. Thus, a single vSDN request, e.g., the vSDN with the longest VCP for $L_{\max}$ or the highest average request latency for $L_{\max\text{avg}}$, governs the optimal latency objective value. For all other vSDNs not increasing the objective, the responsible hypervisors may not be placed optimally with respect to $L_{\max}^{VN,HP}$ and $L_{\text{avg}}^{VN,HP}$.

In summary, while adding hypervisors can improve the overall objective, it may worsen individual latency objectives. Similarly, additional hypervisors increase the request latencies for several other combinations of CPs and request latency metrics, which are not shown for brevity.

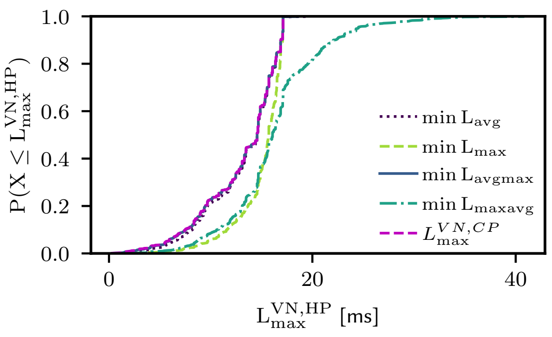
Average-based latency objectives always benefit from additional hypervisor instances. As an example for the average objectives L_{avg} and $L_{\text{avg}\max}$, we observe from Figure 4.12c that for the average-based latency objective $L_{\text{avg}\max}$ the individual requests always benefit from additional hypervisor instances, i.e., from increasing k. By averaging through the maximum path lengths of all vSDN requests ($L_{\text{avg}\max}$), the average-based latency metrics consider all vSDN requests and exploit additional hypervisor instances to achieve lower latency objectives and lower individual vSDN request latencies.

Significant request latency trade-offs among all objectives can be observed. In order to achieve their optimization goal, the objectives lead to trade-offs among the request latencies $L_{\max}^{VN,HP}$ and $L_{\text{avg}}^{VN,HP}$. We illustrate these trade-offs for the hybrid architecture ($M = 17, M_r = 0.5$) with $k = 7$ hypervisor instances. The following observations hold in general also for other setups. As depicted in Fig. 4.13a, the L_{avg} objective achieves the lowest request latencies. We observe a clear trade-off between the $L_{\text{avg}\max}$ and $L_{\max\text{avg}}$ objectives with respect to $L_{\text{avg}}^{VN,HP}$. As expected, $L_{\max\text{avg}}$ pushes down the maximum average latency among all requests, thus, achieving lower latencies for the upper 20 % of the requests. By pushing down the individual maximum path latencies over all requests, $L_{\text{avg}\max}$ pays more attention to the individual paths, i.e., controller to switch connections, of the requests. Consequently, $L_{\text{avg}\max}$ accepts larger values for 20 % of the requests in order to improve the latency of the 80 % remaining requests.

Fig. 4.13b shows again important trade-offs among all objectives. Although $L_{\max}$ minimizes the maximum request latency, it accepts overall worse request latencies than L_{avg} and $L_{\text{avg}\max}$. Further,



(a) Average request latency (HP).



(b) Maximum request latency (HP).

Figure 4.13: Cumulative distribution functions of average ($P(X < L_{\text{avg}}^{VN,HP})$) and maximum ($P(X < L_{\text{max}}^{VN,HP})$) individual vSDN request latencies with virtualization; $L_{\text{avg}}^{VN,CP}$ and $L_{\text{max}}^{VN,CP}$ show the request latencies without virtualization (cf. Fig. 4.11). Fixed param.: $k = 9$ hypervisors, $M_r = 0.5$ multi-contr. switches.

the curve of $\min L_{\max \text{avg}}$ illustrates the working behavior of minimizing $L_{\max \text{avg}}$. While minimizing $L_{\max \text{avg}}$ pushes the maximum average latencies of all requests down (Fig. 4.13a), it relaxes the request latencies $L_{\max}^{VN,HP}$ towards higher values (Fig. 4.13b).

Controller placement strategy and additional hypervisor instances can significantly reduce virtualization overhead. Having observed that the different latency objectives show trade-offs among individual request latencies, we now analyze the virtualization overhead per vSDN request in detail. We introduce metrics that reflect the virtualization overhead ratio, i.e., *the cost of virtualization.* We define the maximum latency overhead ratio of a request $r \in \mathcal{R}$

$$R_{\max}^{VN}(r) = \frac{L_{\max}^{VN,HP}(r)}{L_{\max}^{VN,CP}(r)} \tag{4.50}$$

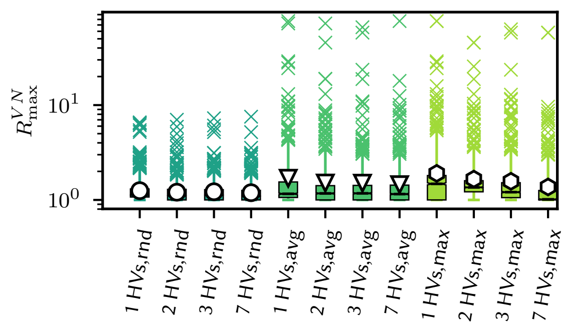
and the average latency overhead ratio

$$R_{\text{avg}}^{VN}(r) = \frac{L_{\text{avg}}^{VN,HP}(r)}{L_{\text{avg}}^{VN,CP}(r)}. \tag{4.51}$$

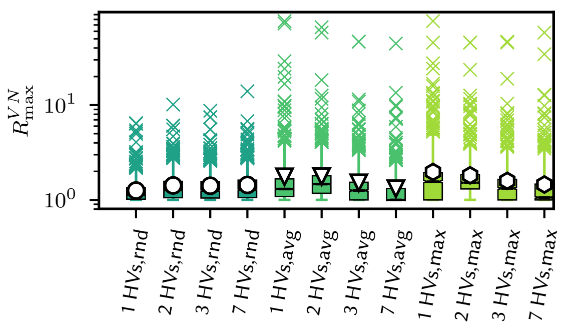
The control plane latency of a request is increased due to virtualization if an overhead ratio is larger than one. An overhead ratio of one means that the request latency is not increased by virtualization.

For analysis, the distributed hybrid architecture ($k > 1$, $M_r = 0.5$) is chosen as it has shown an optimal performance for the HP latency objectives. We selected $k = 1, 2, 3, 7$ to provide a representative set to illustrate the impact of using additional hypervisor instances. Figures 4.14a-4.14d represent the latency overhead ratios of all latency objectives. Boxplots depict how additional hypervisor instances and the CP impact the overhead ratios. As shown by Fig. 4.14, for some vSDN requests the controller latency is up to 100 times higher: small vSDN networks at the border of the network are connected to hypervisor instances on the opposite side of the network. The random CP has the lowest virtualization overhead since it has already a relative high $L_{\text{avg}}^{VN,CP}$ and $L_{\max}^{VN,CP}$, see Fig. 4.11.

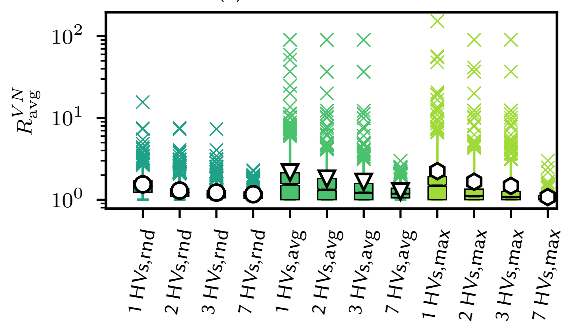
Generally, we observe from Fig. 4.14 that the objectives $L_{\text{avg}\max}$ and L_{avg} achieve the lowest overheads (see $k = 7$). Specifically for R_{avg}^{VN}, the objectives $L_{\text{avg}\max}$ and L_{avg} achieve decreasing latency



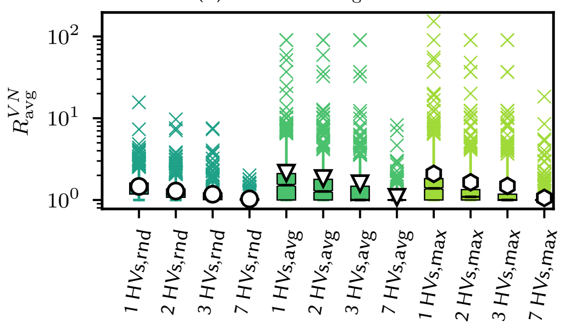
(a) $\min L_{\max}$.



(b) $\min L_{\max\,\text{avg}}$.



(c) $\min L_{\text{avg}\,\max}$.



(d) $\min L_{\text{avg}}$.

Figure 4.14: Boxplots for the maximum and average latency overhead ratios $R^{VN}_{\max}$ and R^{VN}_{avg} (Eqs. (4.50) and (4.51)). An overhead ratio of one corresponds to *no* overhead, i.e., a *zero cost of virtualization.* The green boxes show the upper 75 % quartile and the lower 25 % quartile. The white marker shows the mean and the black line the median. In case the upper and the lower quartile are equal, the whiskers reach the maximum outlier value, shown via dashed lines. The crosses indicate the outliers that do not fall into the 1.5 times interquartile range of the whiskers. For each figure, $k = 1, 2, 3, 7$ hypervisor instances (HVs) are compared for the three controller placement (CP) strategies (rnd, max, avg). **Y-axes are scaled logarithmically.**

overheads as more hypervisor instances are deployed, i.e., k is increased. More than 75 % of the requests (Fig. 4.14c and Fig. 4.14d) achieve an overhead ratio $R^{VN}_{\max} = 1$, i.e., their maximum latencies are not increased at all by virtualization, when $k = 5$ or 7. In contrast, $L_{\max\,\text{avg}}$ exhibits again the mixed behavior for increasing k as observed in Sec. 4.4.2.2.

To conclude, with a moderately high number of hypervisor instances ($k = 5$), the average-based latency objectives $L_{\text{avg}\,\max}$ and L_{avg} have demonstrated the lowest overhead ratios, irrespective of the CP strategy. Thus, when individual request latencies need to be optimized, the objectives L_{avg} and $L_{\text{avg}\,\max}$ should be chosen over $L_{\max}$ or $L_{\max\,\text{avg}}$. Besides, since latencies of only a minor number of vSDNs are significantly increased, a virtualization layer should prepare for those outliers: e.g., additional small-scaled hypervisor instances that only serve small vSDN requests.

Analysis of different substrate network topologies. We now examine the impact of different network topologies. The examination should determine which observations from the OS3E network can be generalized to other topologies. We focus on minimizing L_{avg} as it has generally achieved low latency values so far, including the individual request latencies $L^{VN,HP}_{\max}$ and $L^{VN,HP}_{\text{avg}}$. The substrate topologies have varying numbers of network nodes and links. We set the number of requested vSDNs to $|\mathcal{R}| = 40$ for a close comparison to the preceding analysis. Throughout, we present the

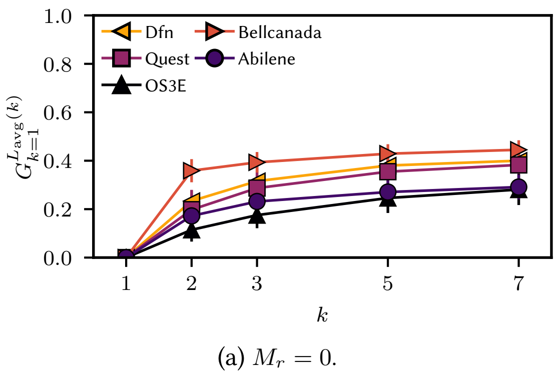
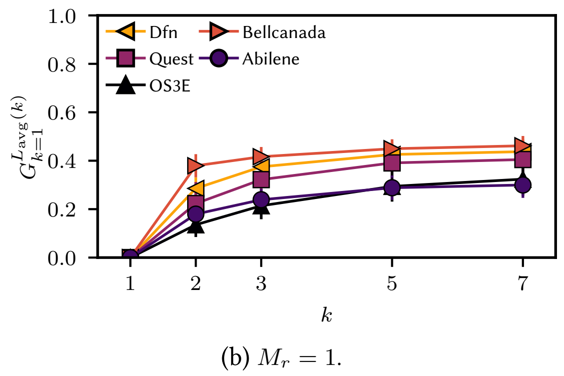


Figure 4.15: Latency reduction due to adding hypervisor instances for five substrate topologies (indicated by marker styles and colors). Multi-controller ratios $M_r = 0$ and $M_r = 1$ are compared for average CP.



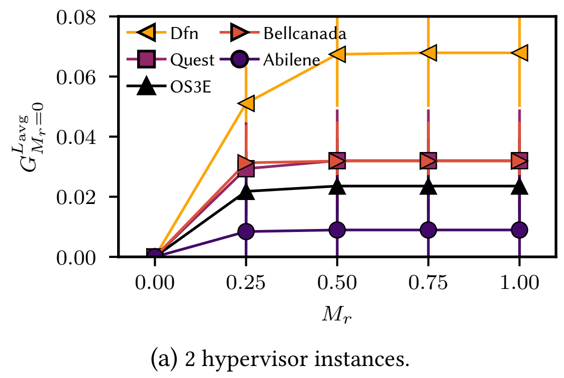
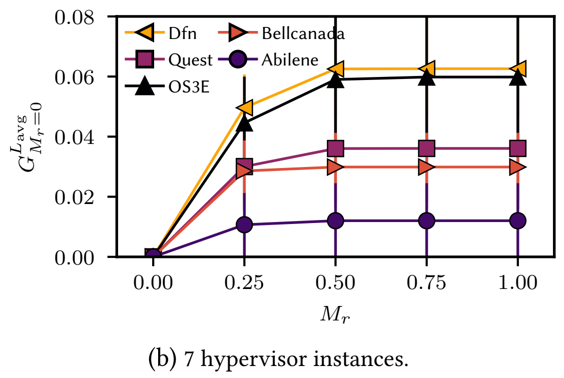


Figure 4.16: Relative latency reduction due to increasing ratio M_r of multi-controller switches in 0.25 steps for different topologies (indicated by marker styles and colors). Distributed architectures are compared for $k = 2$ and 7 hypervisor instances for average CP.



results as relative values: the performance gain of a specific feature is compared to a baseline setup in order to facilitate comparisons across different topologies.

Impact of adding hypervisor instances. We start to examine the impact of adding hypervisor instances, i.e., we evaluate the latency reduction (performance gain)

$$G_{k=1}^{L_{avg}(k)} = 1 - \frac{L_{avg}(k)}{L_{avg}(k=1)}. \tag{4.52}$$

$L_{avg}(k)$ denotes the latency for k hypervisor instances and $L_{avg}(k = 1)$ is the latency of the centralized architecture. A higher gain when increasing k indicates that adding hypervisors reduces the latency. Figs. 4.15a-4.15b show the gains when using the average CP for up to $k = 7$ hypervisor instances. The latency reduction can reach 40 %, even without ($M_r = 0$) multi-controller switches (Fig. 4.15a). As already seen for the OS3E topology, the improvement slowly converges from $k = 5$ onward. This also holds for the distributed architectures, where all switches ($M_r = 1$) can operate in multi-controller mode (Fig. 4.15b).

Impact of adding multi-controller switches. We proceed to examine the performance gain from adding

Table 4.14: Evaluation settings of dynamic analysis.

Parameter	**Values**
Network topology	Cost266 ($\mathcal{V} = 37, \mathcal{E} = 57$)
Ratio of multi-controller switches M_r	0
Number of $\mathcal{R}$	10, 15, 20, 25
Number of $\lvert\mathcal{V}^r\rvert$ per vSDN	Uniformly distributed $2, \ldots, 10$
Number of c^r per request r	1
k	2,3,5,7
Virtual SDN controller placements (CP)	Random (rnd)
Virtual node locations $\pi(v^r)$	Uniformly selected from set $\mathcal{V}$
Hypervisor placement objectives	$L_{\text{max}}, L_{\text{max avg}}, L_{\text{avg max}}, L_{\text{avg}}$
Runs per setup	≥ 30
$\epsilon_{\text{latency}}$	0, 0.01, 0.02, 0.05, 0.1, 0.2

multi-controller switches. Figs.4.16a-4.16b depict the relative latency reduction

$$G_{M_r=0}^{L_{\text{avg}}(M_r)} = 1 - \frac{L_{\text{avg}}(M_r)}{L_{\text{avg}}(M_r = 0)}, \tag{4.53}$$

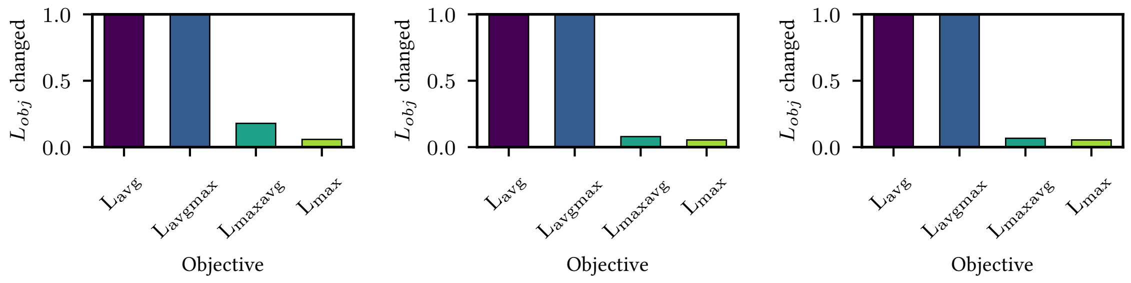
when increasing the ratio (proportion) of multi-controller switches from $M_r = 0$ to $M_r = x = 0.25, 0.5, 0.75, 1$. The figures compare $k = 2$ with $k = 7$ hypervisor instances. When $M_r = 0.5$ multi-controller switches are deployed, an architecture with $k = 2$ hypervisor instances can achieve up to 8 % performance gain (Fig. 4.16a). The larger *Dfn* topology benefits more from the multiple controllers feature than smaller topologies like *Abilene*. The point of diminishing returns of the considered topologies ranges from $M_r = 0.25$ to 0.5. For $k = 7$ hypervisor instances, the absolute performance gain is slightly lower than for $k = 2$ instances. Again, *Dfn* benefits more from the deployment of multi-controller switches than smaller topologies. Note however that some topologies (*OS3E*, *Quest*) still benefit from adding hypervisor instances.

In summary, whereas the topologies show different numbers in terms of absolute values, they all show similar trends. $M_r = 0.5$ multi-controller switches are enough to achieve optimal performance gains.

4.4.3 Placement Results for Dynamic Use and the Cost of Adaptation

Whereas the static use case does not consider any dynamics, e.g., changing vSDN requests or failures, the dynamic use case faces those dynamics. Hence, this analysis considers the case where the number of total vSDN requests changes over time. It focuses on a setup where for a given initial hypervisor placement and a given number of vSDN requests, a new vSDN request arrives. A virtualization layer would have to correctly connect the new vSDN demand under a given latency objective.

The static placement analysis actually revealed that for a varying number of vSDN requests, network operators might need to deploy the hypervisor instances at different network locations. Ac-



(a) $k = 3$. (b) $k = 5$. (c) $k = 7$.

Figure 4.17: Frequency of objective changes over four latency objectives; one figure per $k = 3, 5$ and 7. The y-axes are normalized by the total amount of scenarios for each objective function. Whereas latency always changes in case of average objectives, latency value changes in less than 0.2 of all scenarios for maximum objectives.

cordingly, adding a new vSDN request might demand the hypervisor instances to change their locations in order to still serve the vSDNs optimally. As changing the locations of the hypervisor instances might lead to reconfigurations, it is the aim of this section to answer the following questions:

- What is the impact of the number of hypervisors?
- What is the impact of the objective function?
- What is the impact of the number of vSDN requests?
- What is the cost of reconfiguration (number of vSDN requests whose connections are potentially interrupted, adapted hypervisor instances, severity of hypervisor adaptation)?

The first three questions have also been addressed in the static analysis; they showed the highest impact when analyzing the static setups. The last question particularly targets at *the cost of adaptation* - a new aspect when looking at dynamic scenarios. Table 4.14 summarizes all settings of the following analysis. We take the *Cost266* topology [OWP+10] for our investigation. It is similar in size to OS3E and ATTMpls. Furthermore, the results are comparable to those of ATTMpls.

Effect of number of hypervisor instances k and number of vSDNs. The first Figure 4.17 depicts how often the latency objective changed among all simulations; the figures plot the frequency of changes of L_{obj} of all scenarios for a given objective (obj). Independent of the number of hypervisor instances, the objective values always changed for L_{avg} and $L_{\text{avg max}}$, whereas they changed rarely in case of $L_{\text{max avg}}$ and L_{max}. The observation might lead to a false hypothesis: when adding a new vSDN for the first two objectives, the placement needs to be checked and possibly updated; in contrast, the maximum objectives might not demand an adaptation at all.

Fig. 4.18, however, illustrates that for any objective independent of k and $|\mathcal{R}^\tau|$, the virtualization layer needs to reconfigure VCPs to guarantee an optimal placement independent of the objective. The Figures 4.18a-4.18c indicate the probability $P(R^\tau_{\text{loc}} > 0)$ of a hypervisor location change for any virtual node over the number of all initial vSDNs: the probability that at least one VCP is rerouted to

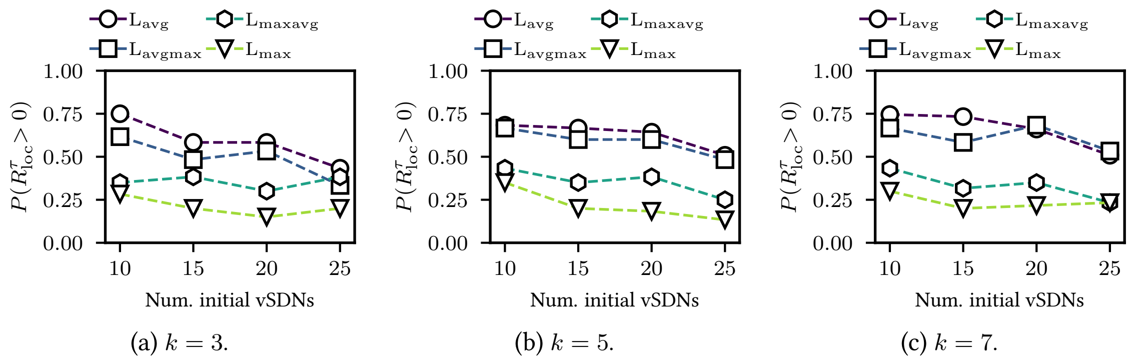


Figure 4.18: Probability that $R^\tau_{\text{loc}} > 0$ over the number of initial vSDN requests for all objectives and $k = 3, 5$ and 7. For each scenario, if there is at least one location change R^τ_{loc}, it will be counted to approximate the probability $P(R^\tau_{\text{loc}} > 0)$. The different marker styles indicate the different objectives: L_{avg} (circle), $L_{\text{avg max}}$ (box), $L_{\text{max avg}}$ (hexagon), L_{max} (triangle down). L_{max} has the lowest probability $P(R^\tau_{\text{loc}} > 0)$ with 0.25, whereas the average objectives show the highest probability with 0.75. The probability of at least one change is slightly decreasing with more vSDNs.

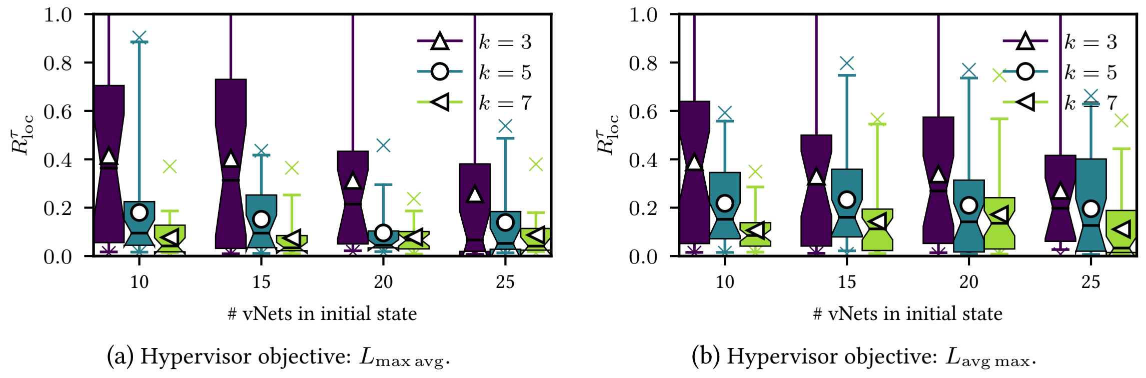
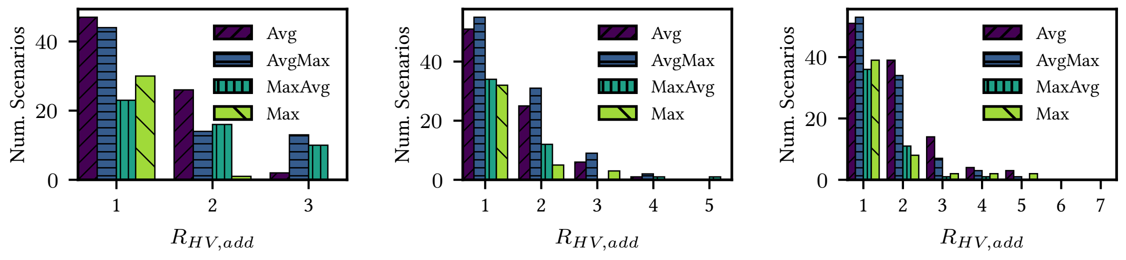


Figure 4.19: Boxplots showing the ratio of vSDN networks facing hypervisor location changes for $k = 3$ (triangle up), $k = 5$ (circle) and $k = 7$ (triangle left) hypervisor instances over the total number of vSDN networks (10, 15, 20, 25) of the initial state at point in time $\tau - 1$. More hypervisor instances decrease the amount of R^τ_{loc}. The numbers of vSDNs do not significantly increase or decrease the ratios of R^τ_{loc}. **Note that R^τ_{loc} is normalized by the total number of vSDN requests for every setup.**

a new hypervisor location. The average objectives (L_{avg} and $L_{\text{avg max}}$) show the highest probability for a location change; this correlates with the first observation that the hypervisor objective changes when a new vSDN arrives. Vice versa, the maximum objectives (L_{max} and $L_{\text{max avg}}$) show a lower probability; however, the probabilities still vary between 0.24 and 0.49. Beside, whereas the probability slightly decreases for more vSDNs in case of the average objectives, the maximum objectives show a more constant behavior; every vSDN contributes to the average objectives whereas only a subset of dominating vSDNs determine the maximum objectives.

More hypervisors need to reconfigure less vSDNs. Fig. 4.19 illustrates how many vSDNs are assigned to new hypervisor locations; the Figures 4.19a-4.19b show boxplots for $L_{\text{max avg}}$ and $L_{\text{avg max}}$ (the results for L_{max} and L_{avg} are similar). The boxplots illustrate the frequency of location changes per simulation run for $k = 3, 5$ and 7. More hypervisor instances reconfigure less VCPs. This result is independent of the objective and the number of vSDNs. Deploying more hypervisors



(a) $k = 3$. (b) $k = 5$. (c) $k = 7$.

Figure 4.20: Empirical distributions of migrated hypervisor instances (i.e., location changes) among simulation runs for twelve scenarios. $R_{HV,add}$ indicates the number of hypervisors that were added (migrated) to a new location. One figure shows the distribution among all objective functions for one k ($k = 3, 5, 7$). For all k, most scenarios have one hypervisor migration independent of the objectives. Note that for $k = 7$, some scenarios have even five hypervisor migrations. **Moreover, note that the number of scenarios is not normalized by the total number of location changes.**

makes the adaptation of the virtualization more flexible; only a small subset of hypervisors need to change their locations. Overall, $k = 7$ reconfigures only up to $20\,\%$ of vSDNs in contrast to nearly $60\,\%$ with $k = 3$. Consequently, operators should use between $k = 5 - 7$ hypervisors for potentially interrupting the least amount of VCPs.

How many hypervisors need to be flexible/dynamic? Fig. 4.20 confirms the previous observation: the virtualization layer relocates only up to three hypervisor instances to achieve a latency-optimal placement. The Figures. 4.20a-4.20c show barplots of the empirical frequency of how many hypervisors have been moved among all simulation runs for each objective; the subfigures represent the distribution of moved hypervisors for $k = 3, 5$ and 7. As it can be observed in all figures, one hypervisor instance is relocated in most cases. The number of relocated hypervisors then decreases until only a few scenarios demand to relocate 4 or more hypervisors for $k = 5, 7$. This also explains the previously observed effect: a larger k affects less vSDNs. With more instances, the vSDNs are more distributed among all instances, which causes less hypervisors to be relocated. When looking at the individual objectives, the maximum objectives migrate less hypervisor instances in total than the average objectives; hence, less vSDNs need to be rerouted to a new location.

Hypervisors might move through the whole network. Figure 4.21 shows the empirical distributions of hops that hypervisors need to move among all scenario settings with 10-25 vSDNs and 5 hypervisor instances. As indicated in Fig. 4.21a, L_{avg} moves a hypervisor only by one or two hops. In contrast, Fig. 4.21b shows that the chance of a movement is uniformly distributed between 1 and 6 hops for $L_{\max}$; the virtualization layer moves an instance up to 6 hops to achieve a new optimal placement. Although $L_{\max}$ might rarely adapt locations, there is high chance that the migration distance is significant. In order to avoid such long migration distances, network operators should particular pay attention when minimizing the maximum objectives.

Trade-off between latency and reconfiguration: How much do we need to relax latency to avoid reconfigurations? The final investigation looks at the trade-off between reconfigurations and latencies. It targets the following question: how much additional latency has to be accepted to

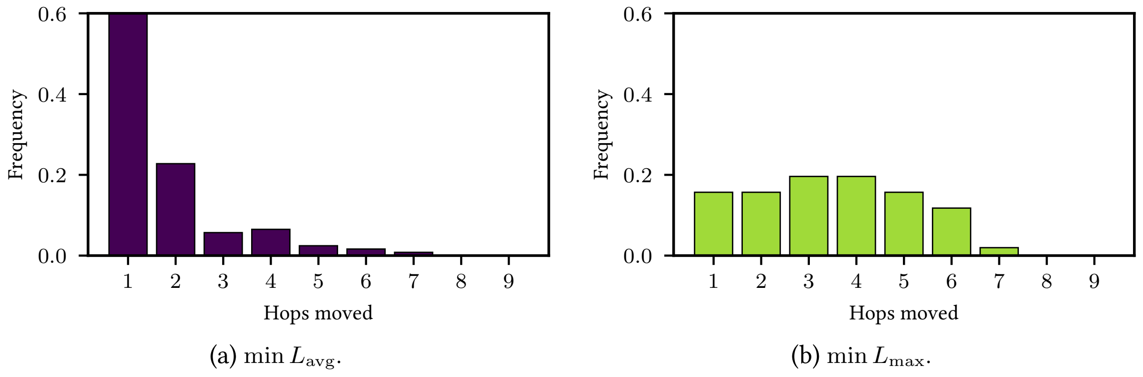


Figure 4.21: Empirical distributions of hops that a hypervisor needs to migrate among all simulation runs of two scenarios. Here, the data contains all vSDN settings (10, 15, 20, 25) for $k = 5$ hypervisor instances. One figure shows the result for one objective ($L_{\text{avg}}, L_{\text{max}}$). Whereas L_{avg} migrates hypervisors at most 1 or 2 hops, hypervisors migrate up to 7 hops for L_{max} with most hops between 4 and 5.

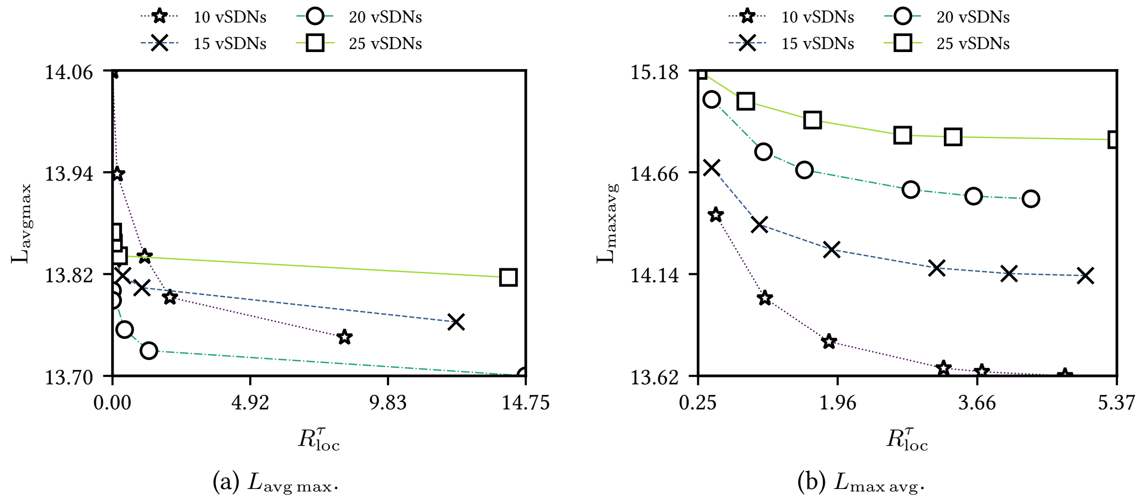


Figure 4.22: Line plots indicating the trade-off between average number of location changes and latency value of the respective objective function ($L_{\text{max avg}}$ (left) and $L_{\text{avg max}}$ (right)) for $k = 5$ hypervisor instances. The latency relaxation factors $\epsilon_{\text{latency}}$are: 0, 0.01, 0.02, 0.05, 0.1, 0.2.

completely avoid reconfigurations? The intuition is that when a new demand arrives, a network operator accepts a larger hypervisor latency in order to not reconfigure the virtualization layer; the old hypervisor placement achieves the relaxed optimal latency. Simulations with an increasing $\epsilon_{\text{latency}}$ are conducted in order to analyze this trade-off.

Fig. 4.22 shows lines that approximate the Pareto optimal solutions of this multi-objective optimization problem; the markers denote the average values of the respective objective solutions. For both latencies $L_{\text{avg max}}$ (Fig. 4.22a) and $L_{\text{max avg}}$ (Fig. 4.22b), relaxing the latency reduces the amount of reconfigurations. When more than 15 vSDNs share a network, $\epsilon_{\text{latency}} = 0.01$ can reduce the average number of reconfigurations to less than 2.5. The reason is that a higher number of vSDNs forces the hypervisors to dominant locations (see again Fig. 4.5). When adapting the virtualization layer, the hypervisors are only moved among these locations (see Sec. 4.4.2.1).

$L_{\text{avg max}}$ benefits more from relaxing the latency than $L_{\text{max avg}}$; whereas relaxing $L_{\text{avg max}}$ by

$\epsilon_{\text{latency}} = 0.01$ reduces reconfigurations by seven times, only $\epsilon_{\text{latency}} = 0.2$ brings R^{τ}_{loc} close to 0 for $L_{\max \text{avg}}$. Note, however, that the amount of reconfigurations is already quite low for $L_{\max \text{avg}}$. Since the average objectives ($L_{\text{avg}}, L_{\text{avg} \max}$) consider all VCPs of all requests $\mathcal{R}^{\tau}$, a new vSDN changes the hypervisor objective less than in case of the maximum objectives ($L_{\max}, L_{\max \text{avg}}$).

In summary, average objectives lead to more reconfigurations than maximum objectives. However, maximum objectives need a higher relaxation to decrease the amount of reconfigurations. It is important to take such observations into account when planning and operating a dynamic virtualization layer in order to not inflate significant latency overhead or to introduce unneeded reconfigurations.

4.5 Summary

In this chapter, we investigate the placement of the hypervisor instances for static and dynamic use; hypervisor instances are the critical components when virtualizing SDN networks. We define MILP models for a centralized and three distributed SDN network virtualization hypervisor architectures. Furthermore, we investigate the impact of multi-controller switches that can simultaneously connect to multiple hypervisor instances. For evaluation of the four modeled architectures, we investigate the impact of the hypervisor placement on the control plane latencies of the entire network as well as individual vSDN. We identify the control plane latency overhead due to the requirement for the SDN Virtual Control Path (VCP) connections to traverse a hypervisor instance for virtualization. This latency overhead represents *the cost of virtualization*.

We observe that virtualization can add significant control latency overhead for individual vSDNs. However, we also show that adding hypervisor instances and using multi-controller switches can reduce the hypervisor latencies for a range of substrate network topologies. Overall, the introduced optimization models let network operators rigorously examine the trade-offs between using SDN hypervisor instances and multi-controller SDN switches.

We extend the study of the hypervisor placement for a static use case to a dynamic traffic scenario: a virtual network request arrives over time and needs to be served by the virtualization layer. Such kind of scenarios are particularly important due to the emerging trend of network softwarization, where virtual networks can be provisioned at runtime. Dynamic scenarios force the virtualization layer to adapt. Adapting the virtualization introduces reconfigurations that should be considered; neglecting reconfigurations can lead to network instabilities resulting in service interruptions or network outages [PZH+11; ICM+02; GJN11; GMK+16]. We extend the MILP model for static use to count for reconfigurations; as minimizing reconfigurations and minimizing latency are conflicting objectives, a multi-objective optimization procedure based on the epsilon-constraint method is proposed. Using this model, we analyze the trade-off between reconfigurations and latency. Relaxing the latency represents *the cost of adaptation*; a new angle when looking at dynamic hypervisor deployments. We notice that the targeted objective functions significantly differ in the amount of reconfigurations when adapting the virtualization layer towards a new optimal placement. We also show that relaxing the latency can help to reduce the amount of hypervisor location changes; however, for some objectives, it is harder to reduce the amount of reconfigurations.

Chapter 5

Machine Learning-based Algorithm Preprocessing for Virtual Network Provisioning

Combining NV and SDN requires a strict performance isolation and resource reservation on both control and data plane. Only by tackling the resource management in both control and data plane, operators can provide guaranteed and predictable performance for virtual networks. Generally, the virtual resources available in virtualized environments, like cloud resources, should be instantiated and scaled quickly to fully exploit the advantages of shared environments [AFG+10; AFG+09; JS15]. With virtual networks affecting provisioned cloud resources [BVS+17], fast and efficient network provisioning algorithms are becoming as important as virtual machine provisioning algorithms [AFG+09; LC16; GHM+08]: e.g., in order to reach to changing networking conditions due to failures or unforeseen behaviors of the provisioned virtual resources. Moreover, operators can cherry-pick requests in a competitive business environment with faster provisioning algorithms [CSB10]. Overall, operators benefit from mechanisms that quickly and efficiently provision virtual network resources [AIS+14; AFG+10].

Resource provisioning in a shared networking environment is also known as the VNE problem; however, this problem is NP-hard [ACK+16; RS18]. The problem has been studied intensively over the last years [FBB+13]. While various optimal and heuristic solutions to the VNE problem have been proposed, current systems are lacking procedures that improve algorithm runtime and consequently algorithm efficiencies.

Beside the VNE problem, many hard algorithmic problems lie at the heart of resource management tasks in virtual SDN environments: network planning for virtualization, deployment of virtual SDN controllers, and operational tasks such as control path reconfigurations etc. Accordingly, over the last decades, we have witnessed a continuous pursuit for ever more accurate and faster algorithms solving each of those single tasks. However, literature so far has overlooked a simple but yet powerful optimization opportunity. Existing algorithms solve frequently a given hard computer networking problem but they do not use their produced data - the problem and solution data. As an example, VNE algorithms are executed repeatedly on potentially similar problem instances. This chapter makes two contributions to address the existing challenges and shortcomings.

First, this chapter presents *NeuroViNE*, a novel approach to speed up and improve a wide range of existing VNE algorithms. *NeuroViNE* relies on a Hopfield network, a special kind of Artificial Neural Network (ANN). *NeuroViNE* preprocesses a problem instance (a substrate state and a virtual network request) and reduces the search space for this instance by extracting a good combination of substrate nodes and links. The extracted subgraph can then be handed to existing VNE algorithms for faster and more resource-efficient virtual network embeddings, i.e., resource reservations.

Second, this chapter offers *o'zapft is*: an ML approach to network algorithm optimization and design in the spirit of data-driven networking. We investigate the feasibility of learning from previous solutions to similar problem instances, the so called problem/solution-pairs, and thereby speed up the solution process in the future; *o'zapft is* predicts upper and lower bounds on objective values, e.g., operator costs or revenues; it effectively prunes the search space, e.g., for MILPs solving the virtual controller placement; it predicts the feasibility of problems instances. Consequently, *o'zapft is* improves algorithm efficiency.

Content and outline of this chapter. Background and related work of this chapter are partly based on content from [BKS+16; BKS+17; BKJ+18]. Sec. 5.2 introduces *NeuroViNE*, which is taken in parts from [BKJ+18]. Sec. 5.3 presents *o'zapft is*; its content is taken in parts from [BKJ+18; BKS+17; BKS+16; KZM+17]. Finally, Sec. 5.4 summarizes this chapter.

5.1 Background

The first Sec. 5.1.1 introduces graph models and network topology sources used for evaluating the systems introduced in this chapter. Sec. 5.1.2 reports on ML in general, and on classifiers, regressors, and measures in particular. In Sec. 5.1.3, Hopfield networks are introduced; they are one representative realizing the concept of neural computation. The evaluations of the proposed systems mainly target the VNE problem, which is introduced in Sec. 5.1.4.

5.1.1 Graph Theory

This section briefly describes the following basics of graph theory: graph types, graph measures, and graph and network models. They are necessary to understand the system designs and the results presented in this chapter.

5.1.1.1 Graph Types

A graph can be simply seen as a set of points (nodes, vertices) interconnected by a set of lines (links, edges). Graphs are used to represent many "objects" in networking science, with the Internet as the most prominent example. Besides, many optimization problems rely on the presentation of problem instances as graphs: the Traveling Salesman Problem, the Graph Coloring Problem, etc.

Due to the huge variety of specifications of graphs, one way to differentiate graph types is to look more closely at the edges of a graph; nodes can either be connected by a single undirected edge, a directed edge, or even multiple edges. Moreover, nodes can connect to themselves by self-loops. Whereas undirected edges describe situations where a bidirectional connection between two vertices exists, directed edges are used in situations where only direct connections exist. For instance, bidirectional communication in computer networks can be represented via undirected edges; directed edges are used, e.g., to describe hyperlinks of a graph representing a network of webpages. All problems investigated in this chapter rely on the representation of graphs with undirected edges without self-loops or multi-edges between nodes.

In this chapter, a network or graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a tuple consisting of the set of nodes $\mathcal{V}$ and the set of links $\mathcal{E}$. The interconnections of an *undirected* graph are captured by its adjacency matrix denoted as $\mathbf{A} \in \{0, 1\}^{N \times N}$ where $N = |\mathcal{V}|$. Many problems and network representations only rely on the information whether a connection between two vertices exists or not: edges show a simple on/off connection. However, situations exist where further information is needed: for instance, the transmission of data from i to j induces a real value. Accordingly, in its weighted variant, the entries of the adjacency matrix carry further information, such as the cost of routing. For undirected networks without edge (cost) weights, the adjacency matrix is represented by a symmetric, binary matrix.

5.1.1.2 Graph Classification & Measures

Classifying graphs is generally hard. The reason is the underlying graph isomorphism problem: are two finite graphs isomorphic? The question is, whether for two graphs $\mathcal{G}^1 = (\mathcal{V}^1, \mathcal{E}^1)$ and $\mathcal{G}^2 = (\mathcal{V}^2, \mathcal{E}^2)$, is there a function $f : \mathcal{V}^1 \rightarrow \mathcal{V}^2$ such that two edges in $i, j \in \mathcal{E}^1$ are only adjacent if and only if $f(i), f(j)$ are adjacent in $\mathcal{E}^2$ [Gra]. As it is not known whether this problem can be solved in polynomial time, different heuristic ways to compare graph instances have been proposed in literature. A common method to classify graphs resp. networks is to employ *graph kernels*. However, kernels are expensive to compute [GFW03].

Another way to describe, and also to classify graphs, is given by graph and node measures, e.g., the degree of a node or the average node degree of a graph. Graph measures were originally used to quantitatively argue about graphs. Node measures can be used to make a point about nodes' importance when arguing about specific use cases, e.g., routing. Table 5.1 provides an overview about the most well-known graph and node measures and their computational complexities [LSY+12]. Each measure captures one or more interesting features of a graph or a node respectively. For instance, the betweenness centrality of a node can indicate important nodes in computer networks: many shortest paths are using a node with a high betweenness centrality. With respect to graphs, an average node degree can be used to quantify the general connectivity inside a graph: a high average node connectivity might yield a graph with many alternative routes. Later, this chapter will report on the node and graph features used for representation and learning (see Sec. 5.3.2).

Beside their ability to help quantifying graphs, using measures allows the representation of a graph or nodes as a fixed-length real-valued vector. For instance, graphs of similar sizes can initially and quite easily be compared by looking at their number of nodes or edges. It has been shown that they can also be used by ML algorithms to classify graphs [LSY+12]. In this chapter, both node and graph measures are used for representing graph data for learning.

Table 5.1: Node and graph features used for problem representation and learning. Note that node features can also be used as comprehensive graph features: for instance, taking the average of all nodes' degrees of a graph provides the average node degree of a graph. Node and graph representations as introduced in Sec. 5.3.2.2 make use of different forms of node-based graph features, namely also the minimum, maximum, and standard deviation value of a feature of all nodes.

Node/Graph Feature	Computational Complexity
Node Degree	$O(n+m)$
Neighbor Degree	$O(n+m)$
Closeness Centrality	$O(n^3)$
Betweenness Centrality	$O(n^3)$
Eigenvector Centrality	$O(n^3)$
Clustering Coefficient	$O(\frac{m^2}{n})$
Effective Eccentricity	$O(2n^2+nm)$
Effective Eccentricity	$O(2n^2+nm)$
Path Length	$O(2n^2+nm)$
Neighbor Degree	$O(2n^2+nm)$
Percentage of Central Points	$O(2n^2+nm)$
Percentage of Endpoints	$O(n+m)$
Number of Nodes	$O(n+m)$
Number of Edges	$O(n+m)$
Spectral Radius	$O(n^3)$
Second Largest Eigenvalue	$O(n^3)$
Energy	$O(n^3)$
Number of Eigenvalues	$O(n^3)$
Label Entropy	$O(n)$
Neighborhood Impurity	$O(nd_{max})$
Link Impurity	$O(n+m)$

5.1.1.3 Graph Models and Network Topologies

In order to provide first general insights into how well the proposed systems of this chapter perform on various network topologies, five substrate graph types are used for evaluation: the three random graph models Erdős-Rényi (ER) [ER59], Barabási-Albert (BA) [BR99], and Waxman (WAX) [Wax88], real substrate topologies from the Topology Zoo (TZ) [KNF+11], and the two data center topologies FatTree (FT) [AFL08] and BCube (BC) [GLL+09].

Whereas the latter three types of networks are representing classes of real networks, random graphs are network models where some specific attributes of the network are pre-determined while others are random. For instance, a simple model is the one where the number of vertices n and the number of edges m is given, and the n vertices are interconnected with m edges randomly.

Erdős-Rényi (ER). The ER graph model $\mathcal{G}(n,p)$, also called random graph, ”Poisson random graph” (refers to its degree distribution), or ”Bernoulli random graph” (refers to its edge distribution) has the number of vertices n which are interconnected with a probability p. The probability for one instance of graph G following this model is given by:

$$P(G) = p^{|\mathcal{E}|}(1-p)^{\binom{|\mathcal{V}|}{2}-|\mathcal{E}|}, \tag{5.1}$$

where each graph $G = (\mathcal{V}, \mathcal{E})$ exists with probability $P(G)$ [New10]. Note that for a given n and p, the number of edges is not fixed due to the generation process: for a given number of vertices n, each edge between a pair of nodes exists with probability $\mathrm{Ber}(p)$ (hence the name ”Bernoulli random graph”). As a result, the total number of edges is binomially distributed with mean $n(n-1)/2 \cdot p$.

The ER model generates graph instances which have shortcomings when compared to real network topologies: e.g, they do not show any clustering; there is no correlation between the degrees of their adjacent vertices; the shape of the degree distributions of the nodes follow a Poisson distribution, which is completely different from the power law distributions of node degrees of prominent graphs like the Internet [New10].

Barabási-Albert (BA). Generative network models try to overcome the mentioned issues of random network graphs. Generative network models create graphs given a generative process. One of the best known model is the ”preferential attachment model”, which is successfully accommodated by the BA network model. Generative network models like BA conduct a simple process to create a network: they add vertices one by one and connect them following a given rule of attachment. The number of new edges per node, e.g., c, is pre-determined. In case of BA, edges are added proportionally to the degree of existing vertices (preferential attachment).

Given this generation process, it has been shown that the BA model generates networks with nodes having a power law degree distribution (with exponent exactly $= 3$). The generated networks have some important attributes to be noted: e.g., (central) vertices that are generated first generally have high betweenness and closeness centrality values - attributes which can be observed in real network topologies [KNF+11].

This, in turn, leads to some criticisms: e.g., the linearity of the generation process; the changing importance of a vertex is neglected. However, as most existing research analyzes their algorithms for the most general models, we also use them for evaluation. A more detailed study of different generative models is an important aspect for future work.

Waxman (WAX). This is a random graph model. In contrast to ER and BA, the WAX model bases the existence of edges on the spatial location of incidents. The basic WAX model places vertices in a unit square, i.e., it generates both x and y coordinates of nodes according to a given distribution: e.g., uniform distribution or normal distribution. Then, the nodes are interconnected with probability [Wax88]:

$$p((v_i, v_j)) := \beta \exp(-\frac{d(v_i, v_j)}{\alpha d_{max}}), \tag{5.2}$$

where $d : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is the euclidean distance between two vertices i and j and d_{max} is a predetermined maximum distance between any two vertices. The parameters $\alpha, \beta \in [0, 1]$ additionally control the generation process; β affects the density, i.e., a larger value generates a more dense network; α controls the ratio of short to long edges, e.g., smaller α values increase the number of shorter edges.

The WAX graph model has disadvantages too: its generation process makes it difficult to provide any analytical expressions describing some features [RTP15]; the degree distributions of nodes of generated graphs do not follow a power law degree distribution [MMB00].

Topology Zoo (TZ). Simple and parameterized graph models are mostly criticized due to one main reason: their inability to capture attributes of realistic network topologies; hence, they do not generate network instances that are close to realistic network topologies. The reason is that realistic network topologies exhibit certain properties which cannot be modeled easily [KBK+17]. Accordingly, Knight et al. [KNF+11] proposed the Topology Zoo (TZ): a set of real network topologies. The set contains topologies with up to 709 nodes. The topologies belong to different types: access, backbone, customer, testbed, transit and Internet exchange points. Moreover, the footprints of the topologies range from metro areas to global networks covering countries among different continents.

Data Centers (DCs). The FatTree (FT) [AFL08] is one of the most common data center topologies. A k-ary FT topology consists of k pods of servers; one pod is connected via two layers of switches where each layer consists of $\frac{k}{2}$-k-port switches. $(\frac{k}{2})^2$-k-port core switches interconnect the pods; each core switch connects to one switch per pod. A k-ary FT interconnects $\frac{k^3}{4}$ hosts. All shortest paths between any hosts have a length of 6 hops.

The BCube (BC) topology is another DC topology [GLL+09]. The parameters are k and n: k is the number of BCubes and n the number of hosts per switch. It is a recursively defined topology: a BCube$_k$ is constructed from n BCube$_{k-1}$ and n^k n-port switches. Contrary to the FT topology, switches in the BC topology are never directly connected to other switches; one switch connects only to hosts; one host connects to multiple switches. A BCube$_0$ with $k = 0$ consist of n hosts that are all connected to a single switch. A BCube$_{k,n}$ topology has a total of n^{k+1} hosts. The longest shortest path in the network has a length of $2(k + 1)$ hops.

5.1.2 Machine Learning

Machine Learning (ML) is a subfield of Artificial Intelligence (AI) that aims at developing self-learning algorithms that can make intelligent decisions. Whereas humans tried to find rules for reasoning about complex systems, ML relies on efficient mechanisms to recognize patterns in data; hence, algorithms can make data-driven decisions based on patterns, which gradually improves, e.g., the performance of optimization processes which face similar problem instances over runtime. Designing a system for network optimization that uses ML, i.e., designing an ML application best suited for a particular networking use case, does not only involve the original problem modeling and solving, but also needs solving the problems of designing an efficient ML system.

5.1.2.1 Supervised Learning

The ML application tries to find a mapping between the input training data $\mathcal{X}$ and its target values $\mathcal{Y}$ by approximating a function $f : \mathcal{X} \to \mathcal{Y}$; it then uses the inferred function to predict the targets for unseen data [Bis06]. In more detail, supervised ML applications use a data set of n `iid` input vectors $\mathbf{x_1}, ..., \mathbf{x_n} \in \mathcal{X}$ along with their output (target) values $y_1, ..., y_n \in \mathcal{Y}$ for training to produce an inferred mapping function. The mapping, i.e., the relationship between input and output data, is captured by f. Hence, for instance, $f(\mathbf{x_1})$ provides the predicted value for $\mathbf{x_1}$.

The successful design of an ML application for supervised learning encompasses many challenges [Tri17]:

- Defining the supervised learning problem
- Finding the ”best” representation of the data
- Collecting the training data
- Finding the ”best” structure for the mapping function
- Choosing the best learning algorithm

Classification. The ML application learns a classification if the output values are chosen from a finite number of discrete categories; it classifies data into given categories [Bis06]. *Binary classification* considers output values with two categories (0/1) [Bis06]. The overall outcome of classifiers can be differentiated into True Positives (TPs), False Positives (FPs), True Negatives (TNs) and False Negatives (TNs).

- True Positive (TP): Counts the samples that were classified correctly as the correct class (1).
- False Positive (FP): Counts the samples that were classified incorrectly as the correct class (1).
- True Negative (TN): Counts the samples that were classified correctly as the wrong class (0).
- False Negative (FN): Counts the samples that were classified incorrectly as the wrong class (0).

Regression. If the output set contains values or vectors of continuous variables, then the ML application does a regression task [Bis06]. An example is *linear regression*: the model describes a linear relationship between input variables and the continuous response variable (output).

5.1.2.2 Machine Learning Algorithms

Different ML models and algorithms with varying complexities exist that can be trained for either classification or regression tasks. The following list outlines the algorithms used in the evaluation part of this chapter.

- Linear Regression & Linear Regression Classifier: both ML models rely on a linear model of the mapping function [Bis06].
- Bayesian Ridge Regressor: Similar like linear regression, a Bayesian Ridge regressor makes a parameter estimation of a linear model by applying Bayesian inference. The Bayesian approach additionally models uncertainty, which can also be used for analysis.
- Decision Tree Classifier/Regressor: A Decision Tree Classifier/Regressor [Qui86] consists of decision nodes and leafs. Decisions are made on nodes based on features. Leafs represent the actual classification or regression result. For a given input vector, multiple decisions are made upon its features until a leaf is reached.
- Random Forest Regressor: The Random Forest Regressor [Bre01] is an ensemble learning method: it uses the average result of a number of decision tree classifiers. Thereby, it tries to avoid the general problem of overfitting of decision tree classifiers.
- Extra Tree Classifier: Whereas training the tree of the Random Forest Regressor [GEW06] works like for a decision tree when taking decisions, the Extra Tree Classifier always chooses the decision parameter at random.
- AdaBoost Classifier: This classifier enhances the learning of a Decision Tree by retraining an instance with higher weights on misclassified training examples [FS97].
- Support Vector Regressor/Classifier: The basic idea behind support vector machines is to determine hyperplanes between the samples of the different classes that achieve a maximal margin - the distance between the hyperplane and the different classes of the training samples that are closest to the hyperplane [SC08].
- (Recurrent) Neural Network: A Neural Network is built by interconnecting an input vector with a layer of artificial neurons to an output layer. Training such single-layer neural network means to adapt the weights connecting different layers (input layer, a neuron layer, output layer). In particular the ”backpropagation” algorithm enabled the breakthrough of neural networks in machine learning [RHW86].

5.1.2.3 Machine Learning Measures/Metrics

The following metrics are used to quantify the performance of regressors:

- Coefficient of determination (R^2): The R^2 is defined as

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - f(\mathbf{x_i}))^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}, \tag{5.3}$$

where $\bar{y}$ is the mean of a set of target values. The nominator of the fraction gives the sum of squares of residuals and the denominator the total sum of squares. For regression, $R^2 = 1$ indicates a precise regression approximation and negative values (up to $-\infty$) indicate a bad performance; 0 means that a model is as good as predicting the average value of a set of samples.

- Root Mean Squared Error (RMSE): Another metric used to indicate the general error of a model is $RMSE$ given as

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(f(\mathbf{x_i}) - y_i)^2}. \tag{5.4}$$

  A value of 0 means that there is no deviation between the predicted value and the true value. The larger the value, the worse the prediction. This value has always to be put into relation to the target to obtain meaningful interpretations.

The following metrics provide information on the performance of classifiers:

- Accuracy: Gives general information on how many samples were classified correctly:

$$\mathrm{ACC} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{FP} + \mathrm{TN} + \mathrm{FN}} \tag{5.5}$$

  Using Accuracy as a measure in highly skewed data sets is problematic. In such case, a classifier can achieve a high accuracy by classifying all samples as the majority class (majority vote).

- Precision (or Positive Predictive Value (PPV)): This measure reveals the amount of samples of the positive class divided by all samples that were classified as positive (TP and FP):

$$\mathrm{PRE} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}} \tag{5.6}$$

  A high precision means that the majority of samples that were classified as positive are belonging to the true positive class (How many selected items are relevant?).

- Recall (or True Positive Rate (TPR)):

$$\mathrm{REC} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} \tag{5.7}$$

  A high recall means that among all samples originally belonging to the positive class most were correctly classified (How many of the relevant items are selected?).

- Specificity (or True Negative Rate (TNR)):

$$\mathrm{SPE} = \frac{\mathrm{TN}}{\mathrm{TN} + \mathrm{FP}} \tag{5.8}$$

  A high specificity means that among all samples originally belonging to the 0 class, most were correctly retrieved.

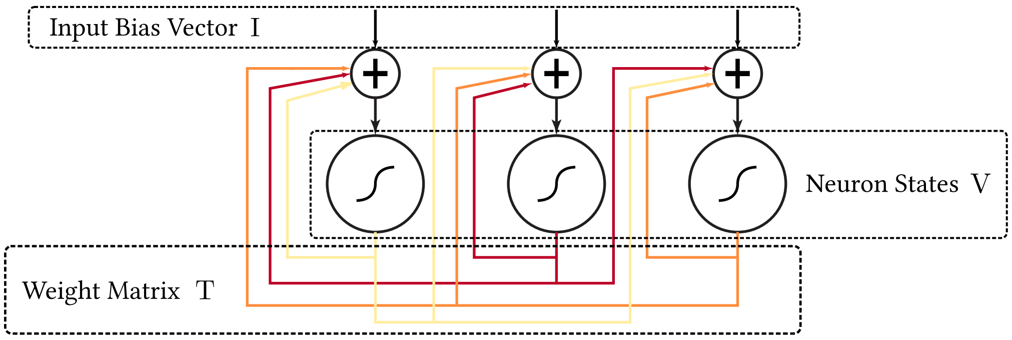


Figure 5.1: A Hopfield Network with 3 neurons. The output of one neuron is fed back as input to the other neurons. Input and output have the same size. The final output is given by the values of the neurons.

- F1 score [Sør48]: This metric is mainly used to provide a more comprehensive performance presentation of a classifier. It combines both precision and recall:

$$\mathrm{F1} = 2 \cdot \frac{\mathrm{PRE} \cdot \mathrm{REC}}{\mathrm{PRE} + \mathrm{REC}} \tag{5.9}$$

  This measure is particularly useful when working with skewed data sets; a poor performance for the underrepresented positive class equally affects the F1 score as the overrepresented negative class. Hence, the F1 score is robust against skewed data sets.

5.1.3 Neural Computation with Hopfield Networks

Neural computation covers research fields where a network of neurons is used to process information: e.g., neural networks are used for prediction or to solve optimization problems. Generally, neural computation is based on the idea to create a structure similar to the brain, which can be used to process information. For this, information is passed through a network of neurons, like in ANNs. Generally, the study of ANNs has been inspired by studies of biological neural networks. Moreover, ML techniques such as deep learning rely on the use of deep neural networks.

Hopfield networks. Hopfield networks are a form of recurrent ANNs [Hop84]. A Hopfield network consists of one layer of neurons that are all interconnected. Fig. 5.1 shows a Hopfield network consisting of three neurons. The input $\mathbf{U}_i$ of each neuron i is the sum of the output of all other neurons and the bias value $\mathbf{I}_i$. A Hopfield network has the following parameters:

- Number of neurons m.
- Vector $\mathbf{U} \in [0, 1]^m$ representing each neuron's internal state.
- Vector $\mathbf{V} \in [0, 1]^m$ representing each neuron's external state.
- Bias vector $\mathbf{I} \in \mathbb{R}^m$ serving as independent input to each neuron.
- Symmetric weight matrix $\mathbf{T} \in \mathbb{R}^{m \times m}$, with $\mathbf{T}_{ij} = \mathbf{T}_{ji}$ being the weight of the connection between neuron i and j and $\mathbf{T}_{ii} = 0$.

- Activation function $h : \mathbb{R}^m \rightarrow [0,1]^m$ calculating the *activation*, i.e., the external state of each neuron from its internal state.

We use a smooth approximation to the step function as activation function h, which was proposed by Hopfield and Tank [HT85] for optimization problems:

$$\mathbf{V} = h(\mathbf{U}) = \frac{1}{2} \cdot \left(1 + \tanh\left(\frac{\mathbf{U}}{u_0}\right)\right). \tag{5.10}$$

The free parameter u_0 controls the steepness of the curve. We fix u_0 to 1 in our experiments since its variation showed no significant impact on our results. With the above parameters, a scalar value can be calculated; the *energy* E of the Hopfield network:

$$E = -\frac{1}{2}\mathbf{V}^T\mathbf{T}\mathbf{V} - \mathbf{V}^T\mathbf{I}. \tag{5.11}$$

Originally, Hopfield networks were considered in the context of content addressable memory [Hop84], due to their ability to reconstruct noisy or partially available memory patterns. This is achieved by associating the pattern to be remembered, i.e., states of the neurons, with low values of the energy function by adapting the weights $\mathbf{T}$. Given a noisy or corrupted sample, the original pattern is recovered by recursively computing

$$\mathbf{V}(t+1) = h\left(\mathbf{T}\mathbf{V}(t) + \mathbf{I}\right) \tag{5.12}$$

until the system converges to a stable state. Note here the use of time t, which simply describes the state of the network at a given time t. Hence, $t+1$ is an indicator for the next time step. When used for optimization of a constrained optimization problem, parameters $\mathbf{T}$ and $\mathbf{I}$ have to be chosen in such a way that they represent the optimization problem to be minimized (or maximized). A state $\mathbf{V}$ for which E attains a minimum then corresponds to a solution of the optimization problem [Hop84], i.e., a state where the objective should be at a (local or global) minimum while all problem constraints are satisfied.

When executing Hopfield networks, their evolution can also be expressed by the following differential equation, as demonstrated by [Hop84]:

$$\frac{d\mathbf{U}}{dt} = -\frac{\mathbf{U}}{\tau_{HF}} + \mathbf{T}\frac{1}{2}\left(1 + \tanh\left(\frac{\mathbf{U}}{u_0}\right)\right) + \mathbf{I}. \tag{5.13}$$

Since Eq. 5.13 captures the Hopfield network's temporal behavior, solving Eq. 5.13 gives a stable state, and thus a solution to the optimization problem [TCP91]. Parameter τ_{HF} causes the neuron input $\mathbf{U}$ to decay towards zero in the absence of any input. Results presented in Section 5.2.3 are obtained with τ_{HF} fixed to 1 as also originally proposed [HT85].

5.1.4 Virtual Network Embedding (VNE)

The Virtual Network Embedding (VNE) problem targets the problem of embedding Virtual Network Requests (VNRs) to a substrate network. The VNE problem is NP-hard in general [ACK+16]. Two variants of the VNE problem are studied in literature: an online variant and an offline variant. In the offline variant, one or more virtual networks are given; the task is either to embed as many VNRs as possible with the highest possible revenue or to embed all VNRs with a minimal resource footprint. The online variant of the VNE problem targets a dynamic use-case; a substrate network operator needs to embed VNRs arriving over time. In this thesis, we focus on the online VNE problem.

5.1.4.1 Virtual Network Embedding Formulation

The VNE formulation is an adapted version of the formulation introduced in [MSK+13].

Substrate network. We consider an undirected graph $\mathcal{G}^s := (\mathcal{N}^s, \mathcal{L}^s, \mathcal{C}^s, \mathcal{B}^s)$ to describe a substrate network. $\mathcal{N}^s$ is the set of all physical nodes $\mathcal{N}^s := \{N_i^s\}_{i=1}^{\iota}$ where ι is the number of nodes of the substrate. $\mathcal{L}^s$ is the set of physical edges with $\mathcal{L}^s \subseteq \mathcal{N}^s \times \mathcal{N}^s$ and $L_{ij}^s = (N_i^s, N_j^s)$. Each node $N_i^s \in \mathcal{N}^s$ has a capacity C_i^s (e.g., CPU) and a residual capacity $C_i^s(t)$ at time t. Every link $L_{ij}^s \in \mathcal{L}^s$ has bandwidth B_{ij}^s and a residual bandwidth $B_{ij}^s(t)$ at time t.

Virtual network requests (VNRs). A Virtual Network Request (VNR) is an undirected graph $G^v := (\mathcal{N}^v, \mathcal{L}^v, \mathcal{C}^v, \mathcal{B}^v)$. $\mathcal{N}^v$ is the set of all virtual nodes $\mathcal{N}^v := \{N_m^v\}_{m=1}^{r}$ of a VNR, where r is the number of virtual nodes. $\mathcal{L}^v$ contains all virtual links with $\mathcal{L}^v \subseteq \mathcal{N}^v \times \mathcal{N}^v$ and $L_{mn}^v = (N_m^v, N_n^v)$. Vice versa, every virtual node $N_m^v \in \mathcal{N}^v$ has a resource requirement C_m^v and every virtual link $L_{mn}^v \in \mathcal{L}^v$ has a bandwidth requirement B_{mn}^v.

The Virtual Network Embedding (VNE) problem. VNE algorithms map arriving VNRs $G^v \in \mathcal{G}^v$ to a substrate network $G^s \in \mathcal{G}^s$, defining a node mapping f_N and a link mapping f_L:

$$f_N : \mathcal{N}^v \rightarrow \mathcal{N}^s, \tag{5.14}$$

$$f_L : \mathcal{L}^v \rightarrow 2^{\mathcal{L}^s} \setminus \emptyset, \tag{5.15}$$

such that

$$\forall N_m^v \in \mathcal{N}^v : C_m^v \leq C_{f_N(N_m^v)}^s(t), \tag{5.16}$$

$$\forall L_{mn}^v \in \mathcal{L}^v : \forall L_{ij}^s \in f_L(L_{mn}^v) : L_{mn}^v \leq L_{ij}^s(t). \tag{5.17}$$

For a valid mapping of G^v to G^s, two steps are important. First, all virtual nodes $\mathcal{N}^v$ need to be mapped to different substrate nodes $\mathcal{N}^s$ (Eq. 5.14). Second, all virtual links $\mathcal{L}^v$ need to be assigned to at least one path in the substrate network (Eq. 5.15), i.e., a subset of links $\mathcal{L}^{p\prime} \in 2^{\mathcal{L}^s} \setminus \emptyset$. Note that we assume unsplittable flows in this work. For a VNR to be embedded, the resource requirements of its virtual nodes (Eq. 5.16) and the bandwidth requirements of its virtual edges must be fulfilled (Eq. 5.17). The right hand sides of Eq. 5.16 and Eq. 5.17 give the residual capacity of physical nodes and links respectively. Moreover, two additional functions are defined to indicate the mapped virtual nodes/edges on a substrate node/edge: f_N^{-1} and f_L^{-1}:

$$f_N^{-1} : \mathcal{N}^s \rightarrow \mathcal{N}^v, \tag{5.18}$$

$$f_L^{-1} : \mathcal{L}^s \rightarrow \mathcal{L}^v. \tag{5.19}$$

5.1.4.2 Virtual Network Embedding Performance Metrics

The following metrics are used to quantify the quality of the solution for the online VNE problem. All metrics are standard in literature to quantify VNE algorithms. The result sections discuss the various metrics and trade-offs among them in more detail.

Acceptance Ratio (AR) is the ratio of accepted VNRs among all received VNRs. Formally, the AR for a given time interval $\mathcal{T} := [t^{\text{start}}, t^{\text{end}}]$ is defined as

$$\text{AR}(\mathcal{T}) := \frac{|\,\mathcal{R}^{\text{acc}}(\mathcal{T})\,|}{|\,\mathcal{R}^{\text{rej}}(\mathcal{T}) \cup \mathcal{R}^{\text{acc}}(\mathcal{T})\,|}, \tag{5.20}$$

where $\mathcal{R}^{\text{acc}}(\mathcal{T})$ is the set of accepted VNRs and $\mathcal{R}^{\text{rej}}(\mathcal{T})$ is the set of rejected VNRs during $\mathcal{T}$.

Virtual Network Embedding Cost (EC) comes from the invested node and link resources to realize a mapping of a VNR. While there is a 1-to-1 mapping between requested and assigned node resources, the costs for realizing a virtual path depend on the physical path length. The embedding cost $\text{EC}(G^v)$ of a VNR $G^v \in \mathcal{G}^v$ is the sum of all substrate resources that have been used to embed the nodes and links for the VNR. It is defined as:

$$\text{EC}(G^v) := \sum_{N_m^v \in \mathcal{N}^v} C_m^v + \sum_{L_{mn}^v \in \mathcal{L}^v} |\,f_L(L_{mn}^v)\,| \cdot B_{mn}^v, \tag{5.21}$$

where $|\,f_L(L_{mn}^v)\,|$ provides the length of the physical path on which the virtual edge L_{mn}^v is mapped.

Revenue (REV) of a VNR is simply determined by the requested virtual resources. The revenue $\text{REV}(\mathcal{G}^v)$ of a VNR is the sum of all its requested virtual link and node resources. It is given as:

$$\text{REV}(\mathcal{G}^v) := \sum_{N_m^v \in \mathcal{N}^v} C_m^v + \sum_{L_{mn}^v \in \mathcal{L}^v} B_{mn}^v. \tag{5.22}$$

The more resources a virtual network requests, the higher the revenue.

Total Revenue (TR) in a time interval $\mathcal{T}$ is the sum over all accepted VNRs $G^v \in \mathcal{R}^{\text{acc}}(\mathcal{T})$:

$$\text{TR}(\mathcal{T}) := \sum_{G^v \in \mathcal{R}^{\text{acc}}(\mathcal{T})} \text{REV}(G^v). \tag{5.23}$$

Revenue-Cost-Ratio (RCR) indicates the relation between revenue and embedding cost. The best possible RCR can be achieved by a 1-to-1 mapping between requested and allocated demands, i.e., all virtual nodes and links are implemented on one physical node and link respectively. The $\text{RCR}(G^v)$ of a VNR is defined as the fraction of its revenue $\text{REV}(G^v)$ divided by its embedding cost $\text{EC}(G^v)$

$$\text{RCR}(G^v) := \frac{\text{REV}(G^v)}{\text{EC}(G^v)}. \tag{5.24}$$

A high Revenue-Cost-Ratio (RCR) indicates a low cost of an embedding: an inevitable target for network operators.

5.1.4.3 Virtual Network Embedding Algorithms

A general survey of existing VNE algorithms is given in [FBB+13]. Algorithms for solving the VNE problem broadly fall in two categories: those which solve the VNE problem in one step, during which both nodes and links are mapped, and those which divide the problem into separate node and link mapping steps. An example for the former are exact algorithms based on mathematical programming [MSK+13]. An example for a non-optimal one step algorithm is [CRB12].

Most heuristic algorithms are 2-step algorithms. In the first step, every virtual node is assigned to the best substrate node that fulfills the capacity constraints. In order to determine the best substrate node for every virtual node, the substrate nodes are usually ranked according to substrate node attributes, e.g., based on their remaining CPU [YYR+08] or graph measure [ZA06; CSZ+11]. To integrate the physical connections between nodes, i.e., physical links and paths, into the ranking, algorithms apply a global ranking procedure. For instance, the MCRank [ZQW+12] or the GRC rating [GWZ+14] make a global rating of the nodes. For each node, their global ranking metric integrates also the distance to all other nodes, e.g., by using random walks through the substrate network. In the second step, after all nodes are first rated and successfully embedded, shortest path or multi commodity flow approaches are used to interconnect the nodes. For evaluations in this chapter, the following optimal, rounding-based, as well as heuristic VNE algorithms are investigated.

- *Optimal algorithms.* The Load Balancing Algorithm (LB) tries to find a mapping solution that first targets a minimum maximum node utilization. In contrast to LB, the Shortest Distance Path Algorithm (SDP) tries to find a cost-optimal embedding with respect to the used links: this is beneficial if link resources are scarce in the substrate network [MSK+13]. Whereas LB works ”well” in situations where bandwidth resources are of no concern, SDP particular targets situations where bandwidth is rarely available.

- *Rounding-based algorithms.* The Deterministic Virtual Network Embedding Algorithm (DViNE) and the Randomized Virtual Network Embedding Algorithm (RViNE) use a relaxed integer programming formulation of the VNE problem [CRB09; CRB12]. Either deterministic (DViNE) or randomized (RViNE) rounding is applied to the relaxed solutions of the VNE problem to find a feasible solution. For the evaluation in this thesis, the shortest path versions of DViNE and RViNE are used.

- *Heuristic algorithms.* The Greedy Algorithm (GDY) is an algorithm with low complexity originally introduced by [YYR+08]. The algorithm rates the nodes based on their residual CPU capacities. In the edge embedding stage, GDY embeds all virtual edges one after the other depending on their ranks. The edges are mapped using the shortest path in terms of number of hops. Gong et al. [GWZ+14] introduced the Global Resource Capacity Algorithm (GRC). The GRC calculates the global resource capacity metric for every node of the substrate network. It then embeds the virtual nodes to the substrate nodes with the highest metric values. The edge mapping stage is the same as one used by GDY.

5.2 Algorithm Preprocessing System Using Neural Computation: *NeuroViNE*

Operators and tenants clearly profit from a fast and efficient embedding of virtual networks: saving computational cost, faster provisioning of virtual networks, smaller waiting times of subsequent virtual network reconfigurations etc. Many exact and heuristic algorithms exist [BK13; FLW+14; ZQW+12; CRB09; CRB12; YYR+08; CSZ+11; FBB+13] for solving the VNE problem. While exact solutions are attractive for their resource efficiency (in terms of resource footprints of virtual networks), they are expensive to compute [ACK+16]. In contrast, heuristic algorithms solve the VNE problem in acceptable time, but, their embedding footprints can be far from optimal. Especially problematic are heuristic algorithms that split the embedding problem into a node and a link mapping step [GWZ+14]: substrate nodes that may be ranked highly in the node embedding step (e.g., due to their available node resources) might be located far away from each other.

This chapter offers *NeuroViNE*, which is motivated by the observation that efficient solutions to the VNE problem place frequently communicating nodes close to each other. Moreover, we observe that many existing time-intensive VNE algorithms may benefit if they can be executed on subgraphs selected intelligently from the substrate network. Accordingly, *NeuroViNE* extracts subgraphs that provide (1) a high probability for being able to accommodate a virtual network and (2) ensure a low-cost embedding. As a consequence, *NeuroViNE* can potentially shorten network algorithm runtimes while preserving a high embedding quality.

5.2.1 Related Work

ANNs have already been successfully applied on a variety of combinatorial optimization problems, including the Traveling Salesman Problem [HT85; Hop84] (which introduced the Hopfield network), the shortest path problem [RW88], the hub-placement problem [SKP96], or the Knapsack problem [AKH92; HK92; OPS93]. In the context of communication networks, recent work used Hopfield networks, e.g., for channel selection in radio networks [AEE15]. Besides, recent work has successfully demonstrated the application of deep neural networks, i.e., deep reinforcement learning [MAM+16] in the context of cloud resource management.

5.2.2 Neural Preprocessor for Virtual Network Embedding

Figure 5.2 shows an overview of *NeuroViNE*. In a nutshell, *NeuroViNE* is a preprocessor which extracts subgraphs to improve the solution quality of both rigorous (exact) or heuristic embedding algorithms which are executed subsequently on the subgraphs.

5.2.2.1 System Overview

NeuroViNE is based on a Hopfield network. The main components of *NeuroViNE* are the ratings for nodes and links of the substrate, the filtering functions ζ determining the number of physical nodes in the subgraph, and the parameters of the Hopfield network, i.e., the weight matrix $\mathbf{T}$, the bias vector $\mathbf{I}$ and the energy function E. Our goal is to accept as many virtual networks as possible while preserving a high Revenue-Cost-Ratio (RCR), which also means to save cost per accepted virtual

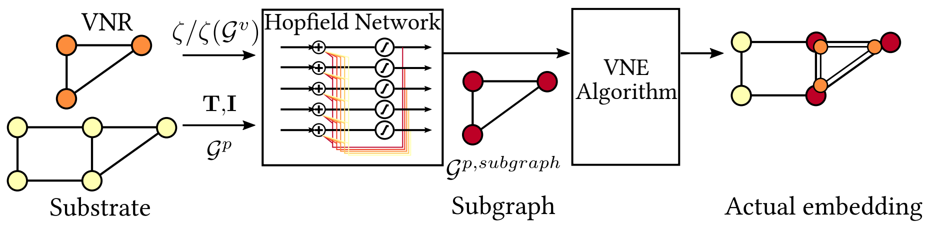


Figure 5.2: *NeuroViNE* first uses a Hopfield network to create a subgraph of the substrate and then applies a VNE algorithm. The VNR size determines the number of preselected nodes. The substrate size determines the number of neurons of the Hopfield network, here five. The substrate provides the input for the weight matrix **T** and the bias vector **I**. Subsequently, the VNE algorithm uses the subgraph to determine the embedding.

Algorithm 1: Pre-selection and Virtual Network Embedding.

```
Input: $\mathcal{G}^s, \mathcal{G}^v$
Output: NodeMapping $f_N$ and LinkMapping $f_L$
1  $\Xi(t) \leftarrow \beta\cdot$ calculateNoderanks($\mathcal{N}^s$)
2  $\Psi(t) \leftarrow \gamma$ calculateEdgeranks($\mathcal{L}^s$)
3  $\zeta \leftarrow$ setNumberOfPreselectedNodes($|\mathcal{N}^v|$)
4  $(\mathbf{T}, \mathbf{I}) \leftarrow$ createHopfieldNetwork($\Xi$, $\Psi$, $\zeta$)
5  $\mathbf{V} \leftarrow$ executeHopfieldNetwork($\mathbf{T}$, $\mathbf{I}$)
6  $G^{s,\text{ subgraph}} \leftarrow \mathcal{G}^s$
7  for $V_i \in \mathbf{V}$ do
8      if $V_i < 0.5$ then
9          removeFilteredNode($G^{s,\text{ subgraph}}$, $N_i^{s,\text{ subgraph}}$)
10 $f_N \leftarrow$ mapNodes($\mathcal{G}^v$, $G^{s,\text{ subgraph}}$)
11 $f_L \leftarrow$ mapEdges($\mathcal{G}^v$, $\mathcal{G}^s$, $f_N$)
12 return $f_N$, $f_L$
```

network. The process is shown in-depth in Algorithm 1 and all steps are explained in the following subsections.

5.2.2.2 Substrate Node and Edge Ranking

In *NeuroViNE*, neurons in the Hopfield network represent substrate nodes. In order to select the potentially best nodes and edges from the substrate, the neurons representing those nodes must be associated with low values of Eq. 5.13. We introduce a node and edge (node connectivity) rating for this purpose, which will be used for the initialization of parameters **T** and **I** in Section 5.2.2.4. Line 1 and line 2 of Algorithm 1 calculate ratings for all nodes and edges of the substrate network. As a good solution for an optimization problem is related with a low energy value of the Hopfield network, node and edge ratings have to be designed accordingly. More in detail, this means that a rating score needs to be low for potentially good substrate nodes; low rating values for both nodes and edges reveal good solutions. Note further that all rankings, matrices etc. are recalculated whenever a new VNR needs to be embedded.

The node calculation step (line 1) can consider any node attribute, e.g., memory or CPU, or any edge attribute, e.g., latency or data rate. In this work, the node ranking vector $\Xi(t) \in \mathbb{R}^{|\mathcal{N}^s|}$ considers the residual CPU capacities at time t as follows:

$$\Xi_i(t) = \beta \cdot \frac{\max_{N_j^s \in \mathcal{N}^s} C_j^s(t) - C_i^s(t)}{\max_{N_j^s \in \mathcal{N}^s} C_j^s(t)} \qquad \forall N_i^s \in \mathcal{N}^s, \tag{5.25}$$

where β is a parameter that weights the importance of the node ranking. By taking the residual CPU, the Hopfield network tries to identify subgraphs with a high remaining CPU capacity, which provides a high chance for acceptance. Dividing by the highest available CPU capacity normalizes the ranking in the interval between 0 and 1. Setting $\beta = 7$ showed the best performance in the conducted simulations.

The edge ratings (line 2) are represented in the matrix $\mathbf{\Psi}(t) \in [0,1]^{|\mathcal{N}^s| \times |\mathcal{N}^s|}$. Determining the values of $\mathbf{\Psi}(t)$ involves two steps; (1) setting the weights of all links and (2) calculating the shortest paths between all nodes based on these weights. The weights of all links are set as

$$w_{\mathrm{HF}}(ij) = B_{max}^s(t) - \frac{B_{ij}^s(t)}{B_{max}^s(t)}, \tag{5.26}$$

where $B_{max}^s(t) := \max_{L_{ij}^s \in \mathcal{L}^s} B_{ij}^s(t)$ is the maximum residual bandwidth at time t among all links. The idea behind the weight setting is to integrate the distance between nodes while simultaneously considering the remaining capacities of links. The link weights w_{HF} are then used to calculate the distance matrix $\mathbf{D}(t)$ containing the costs of shortest paths between all nodes at time t. Finally, the values of the matrix $\mathbf{\Psi}$ are

$$\mathbf{\Psi}_{ij}(t) = \gamma \cdot \frac{\mathbf{D}_{ij}(t)}{\max(\mathbf{D}(t))}, \tag{5.27}$$

where every value is normalized by the maximum value of the matrix $\mathbf{D}(t)$, and the parameter γ weights the edge ratings. Setting $\gamma = 3$ provided the best results in the simulations. Note that we assume that at least one path with remaining capacity exists, otherwise the algorithm would not be executed at all. The obtained ranking results in the selection of a subset of substrate nodes with small distances in terms of hop count to each other, and high residual capacities on connecting links/paths. Combining distance and residual capacity into the path calculation should ensure that the virtual nodes can be connected successfully with low cost.

This ranking procedure makes it possible that the Hopfield network selects a subset of nodes that have few hops between each other, which should result in low path cost solutions. Selecting nodes with high residual capacities ensures that the selected substrate nodes can host the requested virtual nodes. Similarly, integrating the residual capacity into the link weighting should ensure that the virtual nodes can be connected successfully.

5.2.2.3 Node Number Selection Functions

The node number selection functions determine the amount of nodes that should be preselected. The preselected nodes determine the subgraph. As this function is interchangeable, it can be adapted to the embeddings of specific goals. For instance, the selection function can weight smaller networks higher than larger networks. In order to realize such strategy, an operator would simply have to select more nodes for smaller networks than larger networks. Thereby, the acceptance ratio for smaller networks might be larger than for larger networks. In this work, three selection functions are proposed: $\zeta_{\text{const}}(\mathcal{G}^v)$, $\zeta_{\text{factor}}(\mathcal{G}^v)$, and $\zeta_{\text{inverse}}(\mathcal{G}^v)$.

Line 3 calculates the number of nodes based on one specific function. Function $\zeta_{\text{const}}(\mathcal{G}^v)$ simply sets ζ to any pre-defined constant value κ for all VNRs; it is independent of the requested graph $\mathcal{G}^v$:

$$\zeta = \zeta_{\text{const}}(\mathcal{G}^v) = \kappa \tag{5.28}$$

Function $\zeta_{\text{factor}}(\mathcal{G}^v)$ uses the size of a VNR $\mathcal{G}^v$, i.e, $\mid \mathcal{N}^v \mid$:

$$\zeta = \zeta_{\text{factor}}(\mathcal{G}^v) = \kappa \cdot \mid \mathcal{N}^v \mid . \tag{5.29}$$

κ allows to linearly scale the number of selected nodes: e.g., $\kappa = 1$ forces the Hopfield network to select exactly the number of requested VNR nodes. Larger κ should increase the probability of accepting a VNR. Function ζ_{inverse} selects ζ inversely proportional to the number of requested virtual nodes:

$$\zeta = \zeta_{\text{inverse}}(\mathcal{G}^v) = \frac{\kappa}{\mid \mathcal{N}^v \mid} \tag{5.30}$$

ζ_{inverse} is tailored towards the demands of optimal algorithms. When embedding small networks, optimal algorithms can use the entire network, while for VNRs with many virtual nodes, the substrate search space should be reduced for runtime reasons.

For *NeuroViNE* to function correctly, it is important that its Hopfield network preselects a number of nodes that is equal or greater than the number of requested nodes of a VNR (i.e. $\zeta \geq \mid \mathcal{N}^v \mid$ or $\kappa \geq 1$). Note that the Hopfield network may not always select exactly the requested number of nodes. Instead the number of selected nodes might slightly vary around ζ resp. $\zeta(\mathcal{G}^v)$ (Note that we omit to write $\zeta(\mathcal{G}^v)$ later on for brevity). Increasing the amount of preselected nodes increases the likelihood that a VNR can be accepted but also decreases the efficiency of *NeuroViNE*. Accordingly, Sec. 5.2.3.5 analyzes the effect of the choice and the parameter settings of the selection functions.

5.2.2.4 Hopfield Network Creation

Having determined the rankings and the number of nodes that have to be selected, the Hopfield network can be created. The Hopfield network has $\mid \mathcal{N}^s \mid$ neurons. Its creation is done in line 4 of Algorithm 1. This step involves the calculation of the neuron weight matrix $\mathbf{T}$ and the bias vector $\mathbf{I}$. First, we calculate the parts of $\mathbf{T}$ and $\mathbf{I}$ that are caused by the constraints $\mathbf{T}^{\text{constraint}}$ and $\mathbf{I}^{\text{constraint}}$. This is done for all elements of the weight matrix and bias vector using the k-out-of-n rule proposed by [TCP91] as follows

$$\begin{aligned} \mathbf{T}^{\text{constraint}}_{ij} &= \begin{cases} 1 & \text{if } i \neq j, \\ 0 & \text{if } i = j, \end{cases} \\ \mathbf{I}^{\text{constraint}}_{k} &= -(2 \cdot \zeta - 1), \end{aligned} \tag{5.31}$$

where i and j are the indexes of the weight matrix $\mathbf{T}^{\text{constraint}}$ and k is the index of the bias vector $\mathbf{I}^{\text{constraint}}$. The k-out-of-n rule ensures that for a decision problem with n, e.g., Integer variables, k variables will be chosen, i.e., be set to 1. Accordingly, the Hopfield network chooses ζ substrate nodes out of all $\mathcal{N}^s$. Finally, the actual weight matrix $\mathbf{T}$ and the actual bias vector $\mathbf{I}$ can be calculated:

Algorithm 2: Execution of the Hopfield Network.

```
   Input: $\mathbf{I}, \mathbf{T}, \Delta, i_{\max}, dc\_flag, \mathcal{C}_0$
   Output:
1  $\mathbf{U} \leftarrow \mathcal{U}([0,1])^{|\mathbf{U}|}$
2  $i \leftarrow 0$
3  $\Delta \leftarrow \infty$
4  while $(\Delta > \infty) \wedge (i < i_{max})$ do
5      $i \leftarrow i + 1$
6      $\mathbf{k_1} \leftarrow \mathbf{T} \cdot \mathbf{V} + \mathbf{I} - \mathbf{U}$
7      $\mathbf{k_2} \leftarrow \mathbf{T}(\frac{1}{2}(1 + \tanh(\frac{\mathbf{U} + \frac{1}{2}\tau\mathbf{k_1}}{u_0}))) + \mathbf{I} - (\mathbf{U} + \frac{1}{2}\tau\mathbf{k_1})$
8      $\mathbf{k_3} \leftarrow \mathbf{T}(\frac{1}{2}(1 + \tanh(\frac{\mathbf{U} - \tau k_1 + 2\tau k_2}{u_0}))) + \mathbf{I} - (\mathbf{U} - \tau\mathbf{k_1} + 2\tau\mathbf{k_2})$
9      $\mathbf{dU} \leftarrow \frac{\mathbf{k_1} + 4\mathbf{k_2} + \mathbf{k_3}}{6}$
10     if $dc\_flag == True$ then
11         for $i \in \mathcal{C}_0$ do
12             $U_i \leftarrow -\infty$
13             $\mathbf{dU}i \leftarrow 0$
14     $\mathbf{U} \leftarrow \mathbf{U} + \tau \cdot \mathbf{dU}$
15     $\Delta \leftarrow |\mathbf{dU}|$
16     $\mathbf{V} \leftarrow \frac{1}{2}(1 + \tanh(\frac{\mathbf{U}}{u_0}))$
17 return $\mathbf{V}$
```

$$\mathbf{T} = -2(\mathbf{\Psi}(t) + \alpha \cdot \mathbf{T}^{\text{constraint}}), \tag{5.32}$$

$$\mathbf{I} = -(\mathbf{\Xi}(t) + \alpha \cdot \mathbf{I}^{\text{constraint}}). \tag{5.33}$$

The parameter α weighs the constraint terms against the optimization terms. The energy function of the network is given as:

$$E = \mathbf{V}^T(\mathbf{\Psi}(t) + \alpha \cdot \mathbf{T}^{\text{constraint}})\mathbf{V} + \mathbf{V}^T(\mathbf{\Xi}(t) + \alpha \cdot \mathbf{I}^{\text{constraint}}), \tag{5.34}$$

where $\mathbf{V}$ is the vector of the neuron states. The inclusion of the terms $\mathbf{T}^{\text{constraint}}$ and $\mathbf{I}^{\text{constraint}}$ ensures that the correct number of nodes is selected. When Eq. 5.34 is at its minimum, it implies the best possible combination of nodes according to the edge and node ranking. In particular, the sum of the edge rating of the paths between the selected nodes and the node rankings of the selected nodes is at the smallest possible value, i.e., a locally optimal best combination of substrate nodes has been found. The inclusion of the terms $\mathbf{T}^{\text{constraint}}$ and $\mathbf{I}^{\text{constraint}}$ should satisfy the constraint to select a number of substrate nodes as determined by the node number selection function.

5.2.2.5 Hopfield Network Execution

We use the Runge-Kutta-Method [Run95] to solve differential Equation 5.13, which is described in Algorithm 2. First, $\mathbf{U}$, the iteration count i and the state change variable Δ are initialized (line 1 to line 3). The while loop (line 4 to line 16) repeats until either the change to the neuron state vector Δ is smaller than a threshold δ or the maximum number of iterations $i_{\max}$ is reached. Inside the while loop, the next iteration of $\mathbf{U}$ and the difference to the last iteration are calculated (line 6 to line 11). Finally, the activation function calculates the output of the neurons (line 11).

5.2.2.6 Modifications for Data Center Use Case

So far, we made the usual assumption that virtual nodes can be embedded on any substrate node. In order to apply our Hopfield network-based algorithm in data centers, we introduce additional concepts; in more detail, the Hopfield algorithm is adapted to count for the fact that only servers can host virtual machines.

To model data center topologies, we leverage that Hopfield networks can fix some variables before executing them. This method is called *clamping* [GS89]: certain neurons are fixed to certain values. Clamping can be useful when variables should be excluded from the search space or if some variable solutions are already given.

The algorithm now requires additional information about whether the substrate is a data center and also the set of switches $\mathcal{C}_0$. The set $\mathcal{C}_0$ contains all switches, which are used by the Hopfield algorithm according to Constraint 5.35.

$$V_i = 0 \qquad \forall N_i^s \in \mathcal{C}_0. \tag{5.35}$$

For each switch $N_i^s \in \mathcal{C}_0$, Constraint 5.35 forces the associated neuron to 0; switches are excluded from the subgraph.

The parts relevant for data centers are marked in blue in Algorithm 2. If the topology is a data center, the elements of the internal state vector $\mathbf{U}$ representing switches ($\mathcal{C}_0$) are set to large negative values while the deviations $\mathbf{dU}$ are set to 0 (11-13). When the Hopfield network execution converges, the solution contains nodes $\mathcal{C}_0$ which are all set to 0, which excludes switches from the subgraph.

To be a compatible alternative, the data center algorithm uses a fallback algorithm (any VNE algorithm) in case the preprocessor-based subgraph leads to an early rejection. This might increase the computational resources and runtime; nevertheless, as the results will demonstrate, the benefits in terms of cost savings and revenues favor this design. Furthermore, using a heuristic algorithm as alternative still guarantees compatible algorithm runtime.

5.2.2.7 Embedding of the Links and Nodes

After the pre-selection is complete, the subgraph $G^{s,\,\text{subgraph}}$ is created (line 6 to line 9 of Algorithm 1.). $G^{s,\,\text{subgraph}}$ contains only the substrate nodes $N_i^{s,\,\text{subgraph}} \in \mathcal{N}^{s,\,\text{subgraph}}$ for which $V_i \geq 0.5$. In line 10 of Algorithm 1, we call the node mapping function with the subgraph $G^{s,\,\text{subgraph}}$. After it has mapped all virtual nodes, the edges are mapped in line 11. The edge mapping is done with the complete substrate network G^s. If node and link mappings are successful, the network is embedded to the substrate; otherwise, it is rejected.

5.2.3 Evaluation

NeuroViNE can be employed together with many existing algorithms. The results show how *NeuroViNE* works on random network graphs and real topologies in combination with different VNE algorithms.

Table 5.2: Simulation parameters for the study of *NeuroViNE*. Note the different arrival rates and lifetimes between real/random topologies and data center topologies.

Parameter	**Values**
Simulation Process on Random and Real Topologies	
Arrival rate λ	Exponential distribution with $\lambda = \frac{5}{100}$
Lifetime	Exponentially distributed with mean 100
Number of VNRs	2 500
Simulation Process on Data Center Topologies	
Arrival rate λ	Exponentially distributed with $\lambda = \frac{5}{100}$ (FT), $\lambda = \frac{10}{100}$ (BC)
Lifetime	Exponentially distributed with mean 1 000
Number of VNRs	2 500
VNE Algorithms and Hopfield Variants	
VNE algorithms	GRC, SDP, GDY, DViNE, RViNE
Hopfield-VNE algorithms	Hopfield-enhanced Global Resource Capacity Algorithm (HF-GRC), Hopfield-enhanced Shortest Distance Path Algorithm (HF-SDP), Hopfield-enhanced Deterministic Virtual Network Embedding Algorithm (HF-DViNE), Hopfield-enhanced Randomized Virtual Network Embedding Algorithm (HF-RViNE), Hopfield-enhanced Global Resource Capacity Data Center Algorithm (HF-GRC-DC-FB)

5.2.3.1 Methodology

The following sections summarize the simulation setups involving VNE algorithms, substrate network graphs and virtual network requests. For all settings, 10 runs are performed: e.g., 10 substrate graphs are generated for an ER topology with connection probability 0.11. For every setup, a simulation lasts until 2 500 VNRs are processed. All simulation parameters are summarized in Table 5.2.

Virtual network embedding algorithms. In our evaluation, we compare five embedding algorithms with and without *NeuroViNE*: the Global Resource Capacity Algorithm (GRC) [GWZ+14], the Greedy Algorithm (GDY) [YYR+08], the optimal Shortest Distance Path Algorithm (SDP) [MSK+13], and the two ViNEYard [CRB09] algorithms Deterministic Virtual Network Embedding Algorithm (DViNE) and Randomized Virtual Network Embedding Algorithm (RViNE). We use *NeuroViNE* in combination with GRC, SDP, DViNE and RViNE as VNE algorithms; we refer to the algorithm variants with Hopfield preprocessing as HF-GRC, HF-SDP, HF-DViNE and HF-RViNE. The Hopfield data center variant using GRC as fallback is called HF-GRC-DC-FB.

Substrate network graphs. We compare the performance of the algorithms for five substrate graph types: the two random network graph models ER [ER59] and BA [BR99], real substrate topolo-

Table 5.3: Evaluation parameters for substrate network settings.

Parameter	**Values**				
	Substrate				
Graph type	ER	BA	TZ	BC	FT
Number of nodes $\mid \mathcal{N}^s \mid$	100	100	≥ 50	64	54
Model Parameter	-	-	-	k=2,n=4	k=6
Connection Probability	0.11	-	-	-	-
m_0	-	20	-	-	
m	-	1-10	-	-	
Node capacity $\mathcal{C}^s$	$\sim U(50, 100)$		$\sim U(50, 100)$	100	
Edge Capacity $\mathcal{B}^s$	$\sim U(50, 100)$		$\sim U(250, 500)$	100	

gies from the TZ [KNF+11], and the two data center topologies FT and BC. The configurations of the various topologies are summarized in Table 5.3

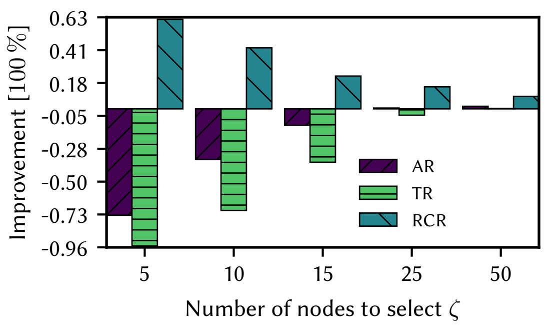
Virtual network requests. For ER, BA and TZ substrates, the VNRs are created using the ER method [ER59]. The number of virtual nodes is equally distributed in the range between 2 and 20. The connection probability is set to 0.5. The required CPU capacities and bandwidths of virtual nodes and links are equally distributed in the range from 0 to 50. The arrival rate λ of VNRs is set to 5 arrivals per 100 time units. Every VNR has a negative exponentially distributed lifetime with an average of 500 time units. This VNR generation is identical to the one used by [GWZ+14].

For data center topologies, the WAX graph model is used. The model parameters are set as follows: $\alpha_{WAX} = 0.2$ and $\beta_{WAX} = 0.4$. The number of nodes is distributed between 3 and 10; the CPU capacities of the nodes between 2 and 20; the virtual link demands between 1 and 10. This is in compliance with a recently published study of VNE in data center networks [HT16].

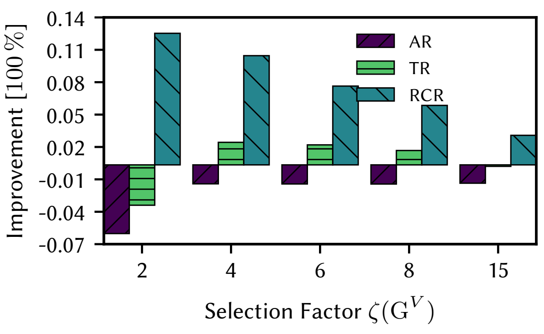
5.2.3.2 Optimization Opportunities

We first analyze the impact of the node selection function on the three key performance metrics: Acceptance Ratio (AR), Total Revenue (TR) (where the time interval lasts until all 2 500 requests are processed), and Revenue-Cost-Ratio (RCR) for the linear and the constant selection function. Figure 5.3a shows the performance improvement for the constant function ζ_{const} and Figure 5.3b illustrates the improvement for the linear function ζ_{factor}. For each selection size, the figures show the bar plots of the improvement defined as *Improvement* $= (\text{HF-GRC} - \text{GRC})/\text{GRC} \cdot 100\,\%$; e.g., a positive value of the AR indicates that HF-GRC accepts more virtual networks.

Constant selection size. Figure 5.3a illustrates that HF-GRC blocks many virtual networks (73 %) when ζ ($= \zeta_{\text{const}}$) is set to a small value like 5 nodes. Consequently, the TR is decreased as less virtual networks are accepted. The RCR, however, is higher: the nodes of the small virtual



(a) $\zeta = \zeta_{\text{const}}$.



(b) $\zeta(\mathcal{G}^v)$.

Figure 5.3: Impact of the selected number of nodes: the left figure shows a constant selection size $\zeta = \zeta_{\text{const}}$; the right figure shows the impact of the selection factor $\zeta(\mathcal{G}^v)$. We show the improvement in percent for each measure: Acceptance Ratio (AR), Revenue (REV), and Revenue-Cost-Ratio (RCR). Positive values indicate a performance improvement. Embedding algorithm: GRC.

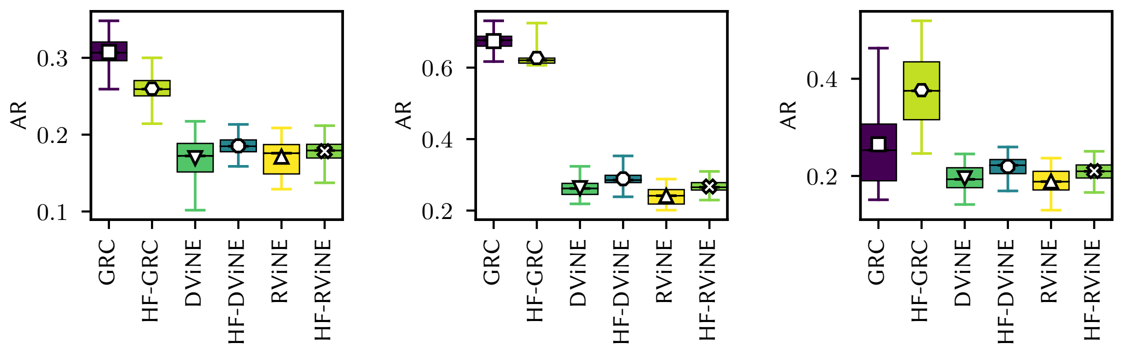
networks are always embedded very close to each other. Increasing the number of nodes to 25 increases the AR to a level that is minimally higher than the one of GRC; whereas TR is still lower. Yet, *NeuroViNE* achieves an up to 18 % higher RCR. When *NeuroViNE* allows only to select subgraphs with $\zeta = 50$ nodes, it shows slightly higher AR and TR, while it still provides a 9 % higher RCR: with *NeuroViNE*, all virtual networks with a size up to 20 nodes can be embedded much more efficiently. However, the problem of the constant function is that it is sensitive to the substrate network size and the VNR request size in terms of nodes. It needs to be pre-determined based on the VNR request size to function well.

Selection factor - dependent on the number of virtual nodes. Accordingly, Fig. 5.3b shows that the factor-based function $\zeta(\mathcal{G}^v)$ has almost the same AR as GRC, except for $\kappa = 2$. Also the TR is insignificantly different from the performance of GRC, again besides for $\kappa = 2$ where it is slightly lower. For $\kappa = 2$, the Hopfield network does not always select enough nodes for bigger networks. Thus, bigger networks are rejected even before the final embedding stage, i.e., before the VNE algorithm, can be run. Fig. 5.3b indicates that HF-GRC improves the RCR ratio up to 8 % for factors ranging from 2 to 8. We believe that trading off the increased RCR for the slightly worse AR and TR is justifiable: decreasing the cost by 10 % for 1 % lower TR. As we aim at a significantly higher RCR for real topologies in the following studies, we use $\zeta(\mathcal{G}^v)$ with $\kappa = 2.5$ showing a still acceptable trade-off between RCR and AR or TR.

5.2.3.3 How does it perform on different topology types (random graphs vs. realistic)?

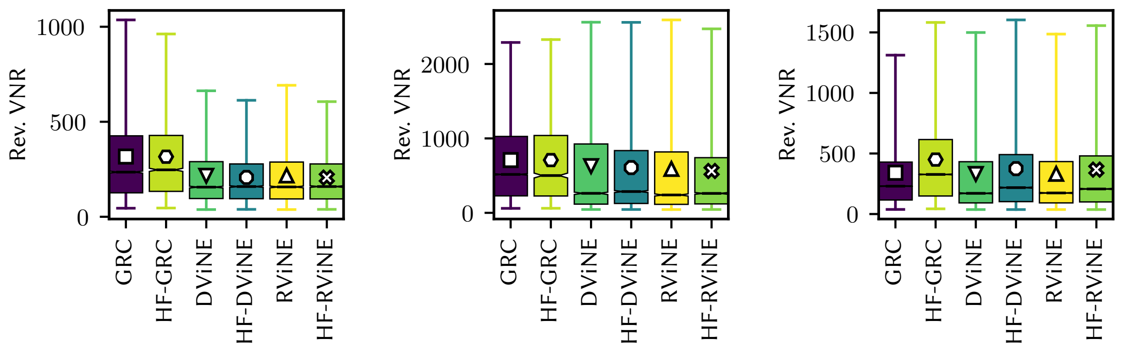
In order to illustrate the impact of different substrate topology models, Fig. 5.4, Fig. 5.5 and 5.6 show the performance of all algorithms for BA, ER and substrates with more than 50 nodes from TZ. Each figure shows the results of one metric (AR, REV, and RCR).

On the Acceptance Ratio (AR). Fig. 5.4a and Fig. 5.4b demonstrate that GRC accepts slightly more Virtual Networks (VNs) on random network graphs, whereas HF-GRC accepts the most VNRs on realistic network graphs (Fig. 5.4c) - HF-GRC is highly efficient when faced with sparsely connected real topologies. While DViNE and RViNE are generally not able to compete with GRC and HF-GRC



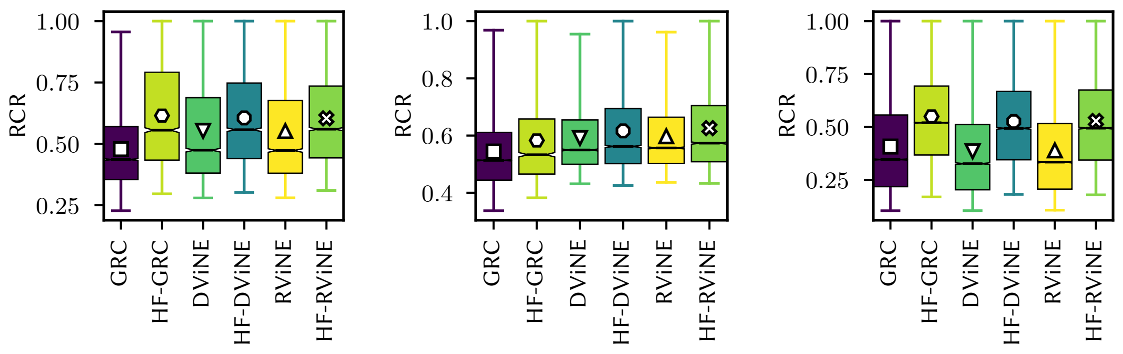
(a) Barabási-Albert (BA). (b) Erdős-Rényi (ER). (c) Topology Zoo (TZ).

Figure 5.4: AR boxplots over VNE algorithms for three topologies (BA, ER, TZ). Subfigures show boxplots of AR over algorithms. VNE algorithms: GRC, DViNE, RViNE. Hopfield variants: HF-GRC, HF-DViNE, HF-RViNE.



(a) Barabási-Albert (BA). (b) Erdős-Rényi (ER). (c) Topology Zoo (TZ).

Figure 5.5: REV boxplots over VNE algorithms for three topologies (BA, ER, TZ). Subfigures show boxplots of REV over algorithms. VNE algorithms and Hopfield variants.



(a) Barabási-Albert (BA). (b) Erdős-Rényi (ER). (c) Topology Zoo (TZ).

Figure 5.6: RCR boxplots over VNE algorithms for three topologies (BA, ER, TZ). Subfigures represent results of all accepted VNRs over algorithms. Comparison between VNE algorithms and their Hopfield variants. *NeuroViNE* achieves higher RCR values on all topology types.

in terms of AR, *NeuroViNE* still slightly improves the AR for both algorithms, as indicated by HF-RViNE and HF-DViNE.

On the Revenue (REV) per VNR. For BA topologies (Fig. 5.5a), *NeuroViNE* shows the same revenue

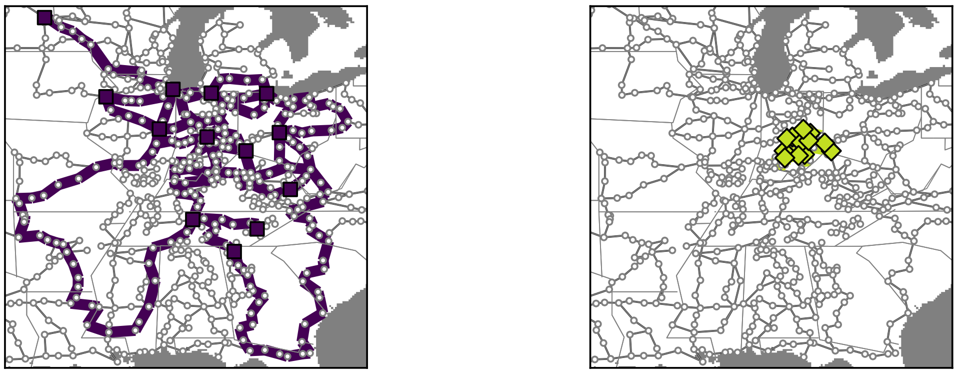

(a) GRC. (b) HF-GRC.

Figure 5.7: Comparison of node locations of a single VNR on the Kentucky Datalink (KDL) [KNF+11] between GRC and HF-GRC. Violet-filled squares show the node placement for GRC (left figure), yellow-filled diamonds for HF-GRC (right figure).

distributions as all other algorithms; the same holds for ER topologies (Fig. 5.5b). For real topologies, *NeuroViNE* again improves the revenues; it even accepts more larger VNRs on real network topologies on average and for 50 % of VNRs (shown by the black median line). *NeuroViNE* does not only accept more VNRs, it even accepts larger VNRs.

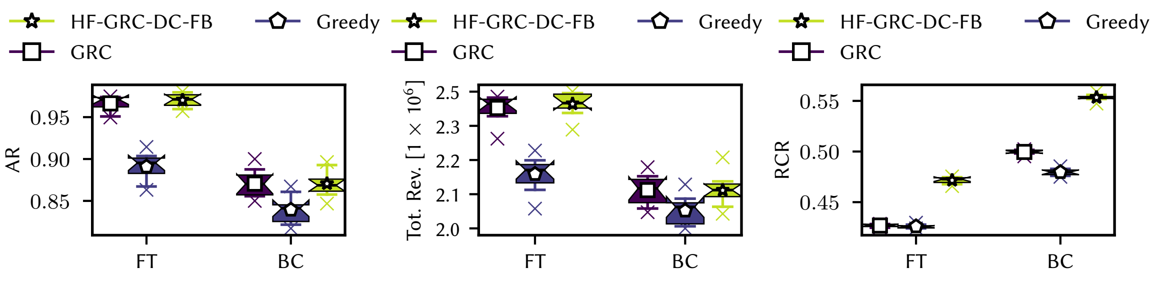
On the Revenue-Cost-Ratio (RCR) per accepted VNR. As Fig. 5.6 illustrates for RCR, *NeuroViNE* outperforms all other VNE algorithms executed without preprocessing, independent of the topology type: HF-GRC, HF-DViNE and HF-RViNE achieve always the highest RCR values when compared to the VNE algorithms without preprocessing. *NeuroViNE* selects for every algorithm a subgraph that provides a RCR that is on average always higher than 0.5. We conclude that *NeuroViNE* can on average improve the RCR independent of the topology.

An illustrative example. Fig. 5.7 illustrates an exemplary embedding for one VNR for both GRC (Fig. 5.7a) and HF-GRC (Fig. 5.7b) on the Kentucky Datalink (KDL) graph (709 nodes and 815 links) from the TZ: this should better explain the working behavior of *NeuroViNE.* As we see, GRC embeds virtual nodes to central nodes on this topology; unfortunately, these central nodes do not need to be near each other, as the spatially distributed green squares illustrate in Fig. 5.7a; long paths need to be taken in order to connect the virtual nodes. In contrast to GRC, HF-GRC selects nodes that are close to each other and have high capacities in their vicinities, as indicated by the orange-filled nodes centrally located in the network in Fig. 5.7b. Since all nodes are close, the paths between those nodes are also shorter, improving not only the embedding quality (RCR) but also the AR and TR.

We conclude that *NeuroViNE* is indispensable on realistic network topologies; it finds valuable nodes in sparsely connected network topologies. Furthermore, independently of the VNE algorithm, it can help to improve the RCR; as a consequence, it reduces cost.

5.2.3.4 Do the Benefits Extend to Data Centers?

Fig. 5.8 reports on the results for data center topologies when using GRC, HF-GRC-DC-FB, and GDY. The Hopfield variant HF-GRC-DC-FB slightly increases AR for the FatTree (FT) topology, whereas



(a) Acceptance ratio. (b) Total revenue. (c) Revenue-cost-ratio..

Figure 5.8: Absolute performance comparison between three embedding algorithms for data centers. Metrics are AR (left figure), TR (middle figure), RCR (right figure).

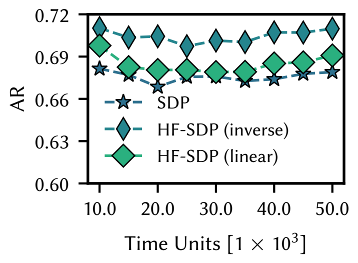
almost no change is observable for the BCube (BC) topology (Fig. 5.8a). Note the high AR values that are already obtained without applying *NeuroViNE*; the values actually range from 0.95 to 0.98 for the FT topology. As a consequence, the total revenue (Fig. 5.8b) is rather unaffected, because of the low blocking probability: both version yield the same total revenue. However, the improvements achieved by *NeuroViNE* in RCR are significant (Fig. 5.8c); HF-GRC-DC-FB improves the performance by 10 % for FT and by 7 % for BC. The cost of all embeddings is reduced by placing clusters within racks where nodes are close to each other. This brings numerous advantages: the clusters might yield lower inter-communication latency or the provider can operate the network more energy efficiently.

5.2.3.5 Can We Speed Up Optimal (Exact) Algorithms?

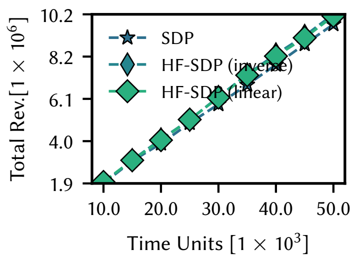
To answer this question, we compare the performance of SDP and the Hopfield (HF) variants HF-SDP with the *linear* and the *inverse* node selection function. For the linear function, κ is set to 2.5 and for the inverse function to 300. Fig. 5.9 shows all subfigures presenting the results for the individual metrics and measures. Fig. 5.9a-5.9c show the results over time: either they show the average or the sum for the particular metric from the beginning of the simulation until the specific point in simulation time. The x-axis of the Figures 5.9d-5.9f are showing the number of nodes of the VNRs: the y-axis shows the mean with the 95 % confidence interval of the given metric over all networks with the given node size.

Fig. 5.9a demonstrates that both HF variants always achieve a higher acceptance ratio. Whereas HF-SDP (linear) only shows a minor improvement, HF-SDP (inverse) achieves a 6 % higher acceptance ratio. As it can be seen in Fig. 5.9b, *NeuroViNE* improves the total revenue over time. Note how the gap between the HF variants and SDP is slowly increasing over time: the longer *NeuroViNE* is used, the higher is the revenue gain.

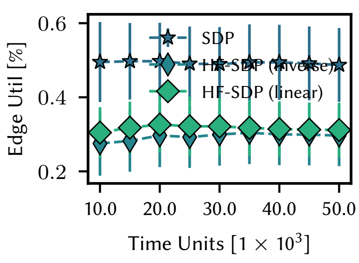
***NeuroViNE* selects closer subgraphs.** Fig. 5.9c illustrates again *NeuroViNE*'s working behavior: *NeuroViNE* selects subgraphs with nodes that are located close to each other. In contrast to SDP, the average edge utilization is decreased for both variants by more then 50 %, while HF-SDP shows the lowest edge utilization. Interestingly, this contradicts the intuition that a more efficient network utilization leads to an increase of the acceptance ratio. The reason for this behavior is due to the fact that the network consists of bottleneck links, which determine the overall embedding perfor-



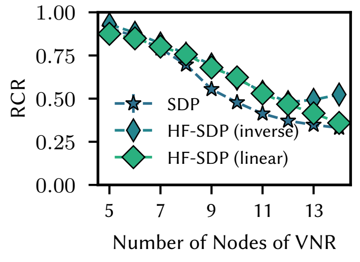
(a) Acceptance Ratio.



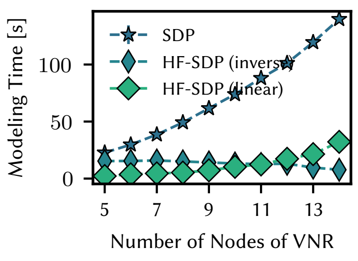
(b) Total revenue over time.



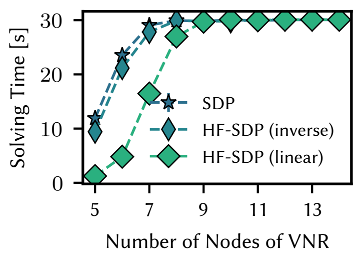
(c) Edge Utilization.



(d) Revenue-cost-ratio.



(e) Model Creation Time.



(f) Solver Time.

Figure 5.9: Performance comparison between SDP and HF-SDP with linear and inverse selection function. ER substrate graph with 100 nodes. Note that AR, TR and edge utilization are plotted over simulation time, whereas RCR, model creation time and solver time are plotted over number of VNR nodes. Hopfield solutions are superior.

mance. However, embedding networks more efficiently, i.e., with nodes closer to each other, helps to circumvent this fact.

Fig. 5.9d fortifies this fact: *NeuroViNE* improves the RCR over the number of nodes. Both Hopfield variants are better than SDP, demonstrating again how *NeuroViNE* selects subgraphs with nodes that are physically close to each other. In particular for larger networks, the RCR can be improved by more than 10 %. This improvement is significant as larger networks are more complex to be embedded.

NeuroViNE shows another interesting angle for performance improvement: the preprocessing efficiently decreases the model creation time and model solving time of the optimal algorithm while preserving its solution quality. The model creation time encompasses the time to create the model for the solver used by the optimal algorithm, e.g., to acquire memory and to set all variables and all constraints. For the HF variants, it also involves the creation and execution of the Hopfield networks.

The linear selection function decreases the model creation and solving time the most, as shown in Fig. 5.9e-5.9f: the solver spends the least time to find a feasible solution; the solver even achieves a ten times lower model creation time. In contrast, SDP and HF-SDP (inverse) are consuming the whole given processing time (30 s) for VNRs having more than 8 nodes when solving the model. Generally, time savings could be spent on improving already embedded VNs or they could be spent to find solutions for larger network requests.

We conclude that using *NeuroViNE* in combination with an optimal algorithm is very beneficial; *NeuroViNE* achieves higher acceptance ratios and total revenues while it actually lowers the cost by decreasing the edge utilization of the networks.

5.3 System for Data-Driven Network Algorithm Optimization and Design: *o'zapft is*

Networking algorithms play an important role in essentially any aspect of operating and managing virtualized networks: from the design of the virtual network topologies, the efficient operation of the virtual network resources, etc. The design of networking algorithms, however, can be challenging as the underlying problems are often computationally hard, and at the same time, need to be solved fast. For example, traffic engineering or admission control problems, or the problem of embedding entire virtual networks, underly hard unsplittable network flow allocation problems; or the problem of placing a virtual SDN controller or virtualization functions can be seen as k-center clustering and facility location problem. Most networking algorithms share a common paradigm: they are executed frequently producing a big data set of problem-solution pairs, which are not yet used efficiently.

In this chapter, we want to demonstrate how to exploit the wealth of data generated by networking algorithms. We propose a system concept that enhances the lifetime of networking algorithms; we thus promote a radically different approach to designing networking algorithms. In particular, this section outlines an ML-based approach, relying on supervised learning, that taps into the produced data of networking algorithms. Thereby, we demonstrate how the approach can improve the algorithm efficiencies for two use cases: facility location and Virtual Network Embedding (VNE).

5.3.1 Related Work

Obviously, the potential of ML in the context of networked and distributed systems optimization has already been observed in many publications. To just name a few examples, Gao [Gao14] showed that a neural network framework can learn from actual operations data to model plant performance and help improving energy consumption in the context of Google data centers.

Another emerging application domain for artificial intelligence is the optimization of networking protocols. Many Internet protocols come with several properties and parameters which have not necessarily been optimized rigorously when they were designed, but which offer potential for improvement. To just name one example, [WB13] have proposed a computer-driven approach to design congestion control protocols as they are used by TCP.

The concept of deep reinforcement learning has been applied to resource management of cloud network resources [MAM+16]. The proposed system learns to manage resources from experience. In a wider sense, Bello et al. [BPL+16] propose Neural Combinatorial Optimization, a framework that relies on reinforcement learning and neural networks to solve combinatorial optimization problems, as demonstrated for the Traveling Salesman Problem. In contrast to our approach, these concepts do not rely on labeled data, i.e., operate in an unsupervised fashion.

The potential of learning methods has been demonstrated in different domains already, beyond distributed systems and networks. A particularly interesting example is the parameterization of MILP solvers, such as CPLEX, Gurobi, and LPSOLVE [HHLB10; Hut14], or the design of branching strategies [KBS+16].

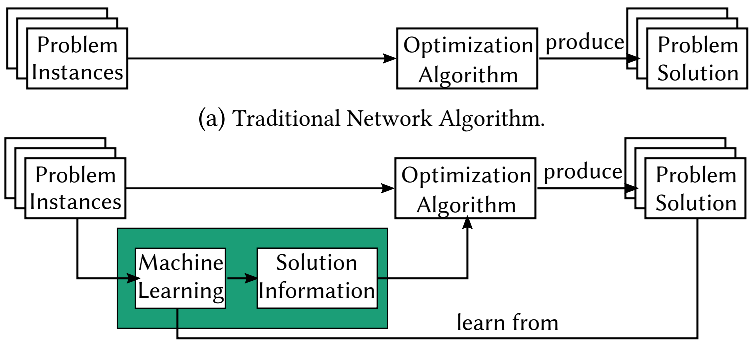


Figure 5.10: Traditional networking algorithm vs. *o'zapft is*. The input to both systems are *Problem Instances*. Traditionally, an *Optimization Algorithm* provides, if possible, a problem solution for each problem instance independently. *o'zapft is* leverages ML to learn from prior problem solutions, to compute additional *Solution Information* (e.g., reduced search space, upper and lower bounds, initial feasible solutions, good parameters, etc.) and hence to optimize the networking algorithm.

Liu et al. [LAL+14] propose that data mining can be used to reduce the search space of high-dimensional problems, hence speeding up the solution of combinatorial optimization and continuous optimization problems.

5.3.2 *o'zapft is*: Challenges, Approach, and Exemplary Implementations

In this section, we identify challenges of our data-driven network optimization system, discuss the design space and optimization opportunities, and point out limitations. In particular, we describe a general framework, called *o'zapft is*[1], and provide two concrete examples, the first one representing a packing optimization problem and the second one representing a covering optimization problem.

***o'zapft is*: tapping network algorithm's big data.** We envision history-aware systems enhancing networking algorithms, which learn from the outcomes of networking algorithms; we propose such system called *o'zapft is*, see Figure 5.10. For a given network problem, the target of *o'zapft is* is to learn from problem solutions that were obtained earlier by an optimization algorithm, in order to improve the future executions of new problem instances. When faced with new problem instances, the optimization algorithm can make use of solution information that is provided by ML models.

The focus of *o'zapft is* are network problems which can be expressed in terms of *graphs*. Graphs are used to describe physical network topologies, but also traffic demands, routing requests and virtual networks (e.g., virtual private networks, virtual clusters, etc.) can be described in terms of graphs.

5.3.2.1 Challenges

When facing such problem types and in order to automatically learn solutions to networking algorithms, we identify four main challenges that need to be solved by a concept such as *o'zapft is*:

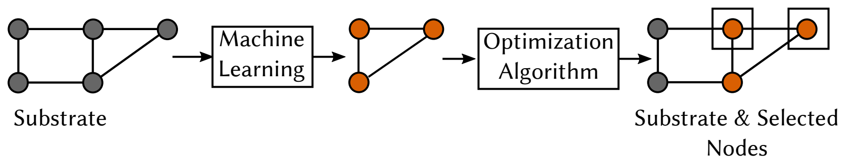
[1] Inspired by the Bavarian expression for "[the barrell] has been *tapped*" (cf. Fig. 5.10), used to proclaim a famous festival in October.

- **Common patterns (C1):** A model from one set of network optimization problems should be generalizable and applicable also to other, similar networking problems: e.g., solutions from smaller networks should be useful to predict solutions for larger networks as well; routing problems could be classified based on requests between similar areas (e.g., subnets) of the network, or reflect distance measures.
- **Compact representations (C2):** It is often infeasible to store the entire history of problem solutions (e.g., the costs and embeddings of all past virtual networks) in a learning framework explicitly. An efficient way to store and represent problem solutions could rely on probabilistic vectors, e.g., capturing likelihoods of nodes visited along a route computed by a routing algorithm.
- **Data skewness and bias (C3):** Making good predictions for highly-biased problems (e.g., containing 1% yes-instances for which feasible virtual network embeddings actually exist and 99% no-instances) is trivial: simply always output *no*. Clearly, such solutions are useless and mechanisms are needed which account for skewness and bias in the data and solution space. Note that the problem arises both when quantifying the performance and also when training the models.
- **Data training/test split (C4):** In terms of methodology, a good split between training and test set must be found. The training/test set typically cannot be split by taking network node samples from the overall data randomly, as each node sample belongs to a unique graph. Taking the example of graphs further, the data set should be split by the nodes belonging to the individual graphs. In case of graphs of different sizes, test and training data set should contain graphs that mimic the overall distribution of the overall data set.

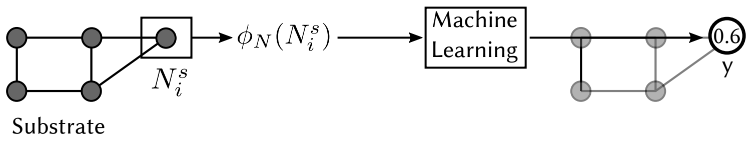
5.3.2.2 Approach

In order to address the challenges (C1-C4) identified above, *o'zapft is* builds upon (and tailors to the networking use case) a number of existing concepts.

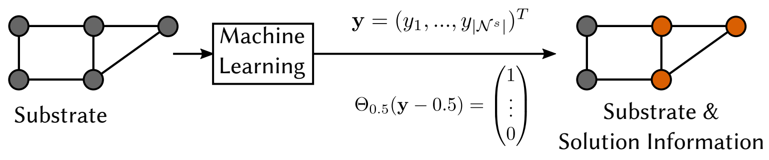
- **C1:** A common method to classify graphs resp. networks (Challenge C1) is to employ *graph kernels*. However, kernels are expensive to compute [LSY+12; GFW03]. An interesting alternative applied in *o'zapft is* is to use node, edge, and graph *features*, e.g., based on existing network centrality concepts [BKER+12], to efficiently classify graphs resp. networks to represent problem-solutions pairs.
- **C2:** We consider fixed-length real-valued feature vectors instead of storing whole adjacency matrices for solutions, which can potentially even outperform alternative methods such as graph kernels [GFW03] as shown in [LSY+12]. For the facility location problem, we also introduce the respective node features per substrate node. Furthermore, we additionally use minimum, maximum, average, and standard deviations of all graph/node features.
- **C3:** There exist standard techniques to overcome bias and data skewness, e.g., *misclassification cost assignment*, *under-sampling*, or *over-sampling* [MB02]. Misclassification costs can



(a) Facility location procedure: the *Machine Learning* module makes a pre-selection among all nodes (black nodes). The *Optimization Algorithm* searches for a feasible solution among pre-selected nodes (orange nodes).



(b) Facility location procedure for a single node: the function $\phi_N(N_i^s)$ converts the single node (surrounded by a square) into a representation for the *Machine Learning* module. The *Machine Learning* module outputs a probability ($y = 0.6$) for this node.


Machine Learning
$\mathbf{y} = (y_1, ..., y_{|\mathcal{N}^s|})^T$
$\Theta_{0.5}(\mathbf{y} - 0.5) = \begin{pmatrix} 1 \\ \vdots \\ 0 \end{pmatrix}$
Substrate
Substrate & Solution Information

(c) Facility location procedure for all nodes: the *Machine Learning* module makes predictions for all nodes; the vector $\mathbf{y}$ contains all each probability y_i. The Heaviside function maps these node probabilities into a binary vector, which results in the orange-colored nodes.

Figure 5.11: Facility location: subfigures illustrate the different steps involved in the overall optimization process. Fig. 5.11a gives an abstract overview. Fig. 5.11b shows how the prediction for one node works, Fig. 5.11c depicts how a pre-selection is created among all nodes.

be assigned if the real cost, e.g., the impact of a wrong prediction on the overall embedding performance, is known. To not loose valid data as in case for under-sampling, we use over-sampling of instances from the underrepresented class. Note that sampling can only be applied to training data, as, of course, the true/false values are not known for test data beforehand.

- **C4:** We split the overall data based on entire graphs, and hence avoid, e.g., problems of node-based samples. That is to say, training data and test data both contain all samples of a substrate graph. As an example for VNE, all VNRs that were embedded on a substrate graph are either in the training or the test data.

5.3.2.3 Example I: Placement Prediction for Facility Location-related Optimization Problems

We consider an archetypal network optimization problem: facility location. Many existing network optimization problems find their roots in the facility location problem: content placement, cache placement, proxy placement, SDN controller placement, virtual SDN controller placement etc. Because of this, we see the study of the facility location problem as fundamental for future ML-based network optimization approaches.

Concept. Fig. 5.11 provides an overview about the system. The system consists of an ML part and a placement algorithm, e.g., a virtual SDN controller placement algorithm (Fig. 5.11a). Generally, the input of an uncapacitated simple facility location problem is a substrate network $G^s = (\mathcal{N}^s, \mathcal{L}^s)$, a set of demands $\mathcal{D}$ that need to be connected to k to-be-allocated facilities $\mathcal{F}$. Every demand needs to be connected to at least one facility. The minimum k-center variant of this problem is already NP-hard. The ML part predicts for each substrate node whether it will host a facility (Fig. 5.11b). Every time, the input of the ML algorithm is specific to a node $N_i^s \in \mathcal{N}^s$. There are different examples for how to use the prediction output: search space reduction, initial starting point, or even as a solution.

Network node representation. The features of a network node $N_i^s \in \mathcal{N}^s$ for a given substrate graph $G^s \in \mathcal{G}^s$ must allow a distinction between nodes that will host a facility and the ones that do not. In order to have a representation that is independent of the graph size, the node presentation for facility location problems in this chapter uses graph and node features. For studying the impact of the use of features of different types and complexity on the ML results, three feature settings are compared, namely the low complex (LC) feature setting, the node features-related (NF) one, and the node plus graph features-based (NF+GF) one. The LC feature setting uses all graph and node features with linear complexity; NF uses all node-related measures, e.g., betweenness centrality as well as the min/max values of those measures corresponding to the inspected graph; NF+GF uses all available node and graph features, i.e., also pure graph features like energy or average path length (see Table 5.1). Consequently, ϕ_N is defined as $\phi_N : \mathcal{N}^s \rightarrow \mathbb{R}^n$, where n is the number of features contained by the used feature setting.

Machine learning input. Formally, an ML algorithm learns a mapping $m : \mathbb{R}^n \rightarrow z$ with $z \in \{0, 1\}$: a node either hosts a facility (1) or not (0). The used classifiers actually predict $y \in [0, 1)$. This allows a direct probabilistic interpretation of the prediction y: $P(z_i = 1|\phi_N(N_i^s)) = y$ (node i hosts a facility) and $P(z_i = 0|\phi_N(N_i^s))$ (node i does not host a facility).

Fig. 5.11c illustrates the next steps. Making a prediction per node, the overall output of the ML system is a vector $\mathbf{y} = (y_1, ..., y_{|\mathcal{N}^s|})^T$ providing the probability of hosting a facility for all nodes. Based on a threshold, e.g., $\theta = 0.5$, the optimization system makes for each node a decision $z_i = 1$ if $P(z_i) = y > \theta = 0.5$. Such decision can be implemented by using the Heaviside function with $\Theta_{0.5}(y - 0.5)$. Note that this might classify more than k nodes as facilities.

Hence, different ways to process $\mathbf{y}$ can be applied. For instance, the vector can be sorted and the best k nodes can be chosen; or the sorted vector can be used to reduce the search space. A facility location algorithm will only search among the candidate nodes that have either a higher chance (e.g., $P(z) > 0.3$) or it will look among the best $50\,\%$ of the nodes (see again Fig. 5.11a).

5.3.2.4 Example II: Admission Control System for Virtual Network Embedding

The embedding of virtual nodes and routes between them is an archetypal problem underlying many resource allocation and traffic engineering problems in computer networks. The common theme of these problems is that network resource consumption should be minimized, while respecting ca-

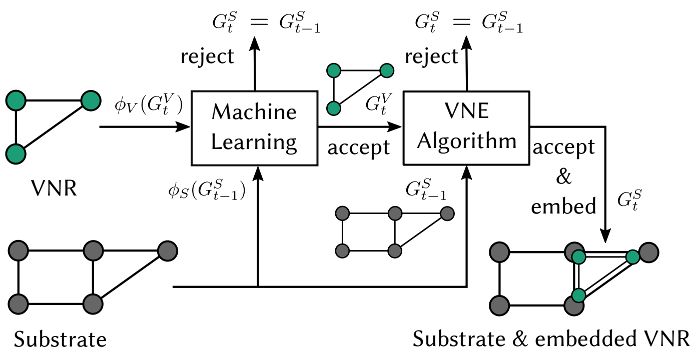


Figure 5.12: Admission control for an algorithm for the VNE problem. A *Machine Learning* module takes the vector-based representations of a VNR and a substrate as input in order to predict the feasibility of the request, i.e., to reject or to accept and forward the request a VNE algorithm. If the VNR is accepted, the substrate state is updated; otherwise, the substrate remains in the old state (before the arrival of the VNR).

pacity constraints. We investigate whether ML can be used to *predict the embedding costs* or the feasibility of serving VNRs.

Concept. Figure 5.12 shows a schematic view of the proposed system. The system consists of two main blocks: the ML part, e.g., an admission control, and the VNE algorithm. It is the goal of the ML part to assess the VNR before processing it by the VNE algorithm. The ML part predicts various VNE performance measures: it either predicts the chance for a successful generation of a solution by the VNE algorithm, or it predicts measures such as the cost of a VNR. Guessing the potential success means that the ML part classifies the VNR that arrived at time t. A VNR G_t^v arriving at point in time t can be either *infeasible*, unsolvable in acceptable time (*no solution*), or *accepted*. Predicting measures means the ML implements a regression task. Any appropriate ML algorithm can be used to implement the classification/regression functionality.

The input of the ML algorithm needs to be compact and independent of the size of the substrate G^s and the to-be-embedded VNR G_t^v. For this, the functions ϕ_S (*Substrate Network Representation*) and ϕ_V (*Virtual Network Request Representation*) create feature vector representations of the substrate network respectively the VNR, which will be explained in the following paragraphs.

Substrate network representation. Features for substrate graphs $G^s \in \mathcal{G}^s$ must allow the distinction of states in which a request can be embedded. Hence, we consider the inverse graph $G^{s\prime} := (\mathcal{N}^{s\prime}, \mathcal{L}^{s\prime})$ induced by the embedded requests on the actual graph G^s. The set of edges is defined as $\mathcal{L}^{s\prime} := \{l^s \in \mathcal{L}^s \mid 0 <| f_{\mathcal{L}}^{-1}(l^s) |\}$ and the node set is defined as $\mathcal{N}^{s\prime} := \{n^s \in \mathcal{N}^s \mid 0 <| f_{\mathcal{N}}^{-1}(n^s) |\} \cup \{\{u, v\} \mid (u, v) \in \mathcal{L}^{s\prime}\}$. As $f_{\mathcal{L}}^{-1}$ and $f_{\mathcal{N}}^{-1}$ change over time, we obtain a network with a changing topology. All graph features are recalculated (see Table 5.1) whenever the used resources of the substrate network changes: for instance, it captures the changing amount of used nodes and links by considering only nodes and links that are actually used. Note however that the current procedure works only for a connected graph $G^{s\prime}$. Whereas the inverse network has never been disconnected in any simulation, a procedure for unconnected graphs is needed for future.

In addition, so called VNE resource features (RF) are introduced to enrich the network representations: three CPU features (*free CPU*, *occupied CPU*, and *total CPU*), three bandwidth features (*free* *bandwidth*, *occupied bandwidth*, and *total bandwidth*) as well as two embedding features (*total number of currently embedded edges* and *total number of currently embedded nodes*). All resource features have linear complexity.

Finally, $\phi_s(\mathcal{G}^s) : \mathcal{G} \to \mathbb{R}^n$ defines the mapping from a graph $G^s \in \mathcal{G}^s$ to the graph and resource features. The final feature vector is given by $\mathbf{x}^s := (x_1^{GF}, \dots, x_i^{GF}, x_{i+1}^{RF}, \dots, x_n^{RF})$. For proof of concept, all ML algorithms for VNE use all 21 graph features $x_1^{GF}, \dots, x_{21}^{GF}$ and the eight resource features $x_{22}^{RF}, \dots, x_{29}^{RF}$. However, the analysis in this chapter will also use other feature settings in order to investigate the potential of features with linear or log-linear complexity.

Virtual network request representation. The features used to represent VNRs need to fulfill in particular one task: the distinction between VNRs following different graph types. Alike, the representation of similar graphs should be similar. For graphs G^1, G^2 and G^3 where $\mathcal{N}_1^v = \mathcal{N}_2^v = \mathcal{N}_3^v$ but $\mathcal{E}^{G_1}, \mathcal{E}^{G_2} \sim B(n, p_1)$ and $\mathcal{E}^{G_3} \sim B(n, p_2)$ with p_1 sufficiently different from p_2, it is required that $d\left(\phi\left(G^{N_1^s}\right), \phi\left(G^{N_2^s}\right)\right) \leq d\left(\phi\left(G^{N_1^s}\right), \phi\left(G^{N_3^s}\right)\right) \simeq d\left(\phi\left(G^{N_2^s}\right), \phi\left(G^{N_3^s}\right)\right)$ for some distance function d defined on $\mathbb{R}^n$.

To obtain a compact but preliminary representation, a Principal Component Analysis (PCA) was performed on a set of feature vectors $\{\phi_V(G^1), \dots, \phi_V(G^m)\}$ consisting of all graph features. The features are extracted from a set of graphs $\{G^1, \dots, G^m\}$ generated according to different models: ER, BA, and WAX. The features were then selected based on high coefficients in the linear transformation of the input data obtained via the PCA. The features with the highest load are *number of nodes*, *spectral radius*, *maximum effective eccentricity*, *average neighbor degree*, *number of eigenvalues*, *average path length* and *number of edges*. They allow the recovery of the models using Multinomial Linear Regression with an accuracy of 99.0 % and thus satisfy the requirements on the representation stated above. The VNR representation is then given by the mapping $\phi_V : \mathcal{G}^v \to \mathbb{R}^n$ with $n = 7$ and $\phi_V(G^V) = \mathbf{x}^V = (x_1, \dots, x_7)$ where x_i corresponds to one of the features listed above.

Machine learning input. The input to the ML algorithms is the vector $\mathbf{x} := (\mathbf{x}^s, \mathbf{x}^v)$ with $\mathbf{x}^s = \phi_S(G^s(t))$ and $\mathbf{x}^v = \phi_V(G^v))$, which is a combination of $\mathbf{x}^s$ and $\mathbf{x}^v$ for the substrate and the request as defined in the previous section. Formally, an ML algorithm learns either the *classification* mapping $c : \mathbb{R}^n \to z$ with $z \in \{0, 1\}$ or the *regression* mapping $r : \mathbb{R}^n \to o$ with $o \in \mathbb{R}$ where $n = 36$ (the dimension of $\phi_S(\mathcal{G}^s)$ is 29 and 7 for $\phi_V(\mathcal{G}^v)$). For z, 0 represents *reject* and 1 *accept*. For o, a value represents either the cost or the potential revenue, which can also be negative.

5.3.3 Case Study I: Predicting Facility Locations

The first case focuses on predicting facility locations. It analyzes the potential of using the data of exact placement solutions for (1) exactly predicting the placement of facilities on unseen graph instances and (2) for reducing the search space of exact algorithms.

5.3.3.1 Settings

In order to provide insights into the performance of an ML-based optimization system, three facility location algorithms are compared with three ML-based algorithms. The comparison is conducted on different graph types.

Simulation data. The following analysis uses random graphs and graphs taken from the Topology Zoo (TZ). The random graph data consists of random graphs generated with the ER and BA models. All graphs have 40 nodes and a connection probability of 0.11, which results in graphs that have a comparable node-to-edge ratio as graphs generated with the GT-ITM tool [ZCB96]. The TZ data contains all graphs from the TZ having at least 20 nodes.

Facility location algorithms. The task of the algorithms is to choose up to k facilities out of $\mathcal{F}$ facility locations on a graph $G^s = (\mathcal{N}^s, \mathcal{L}^s)$; in our study $\mathcal{F} = \mathcal{N}^s$. We use three baseline algorithms in our study: an optimal Mixed Integer Linear Programming (MILP) algorithm, a Greedy Algorithm (GDY), and a Random Algorithm (RND). GDY solves the facility location by iteratively taking the next node which maximizes the reduction of the objective function; GDY executes k iterations. RND just chooses k nodes from $\mathcal{N}^s$ randomly.

Machine learning-based facility location algorithms. As stated in Sec. 5.3.2.3, the outcome of the ML part is a vector $\mathbf{y} = (y_1, ..., y_{|\mathcal{N}^s|})^T$ containing probabilities for each node to host a facility. We propose different procedures how to use $\mathbf{y}$.

The exact algorithm (*o'zapft is*-Exact) uses the k nodes with the highest probability ranking. The pre-selection-based algorithms (*o'zapft is*-MILPX) use a $ratio$ (either 0.25 or 0.5) parameter to determine the number of nodes to select among all substrate nodes. For instance, for a network of 40 substrate nodes and a ratio of 0.25, *o'zapft is*-MILP0.25 selects the ten nodes with the highest probabilities. Then for *o'zapft is*-MILPX, the MILP is executed on the preselected subset of nodes to find the best solution.

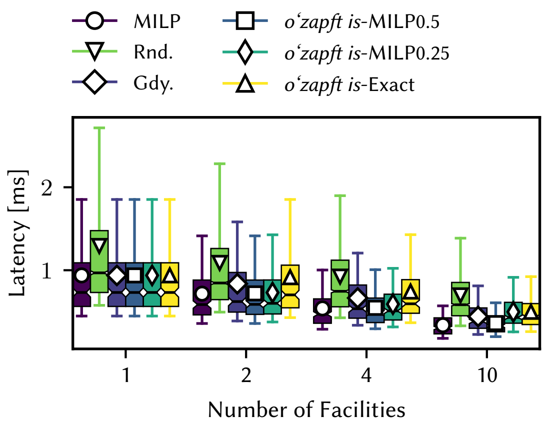
Machine learning algorithms. Four classifiers are compared: Logistic Regression Classifier (LRC), Extra Trees Classifier (ETC), AdaBoost Classifier (ABC), and Support Vector Classifier (SVC). For all models, a grid search is applied on the parameters to identify the best performing one. When presenting results, either an explicit classifier is mentioned or the results show the outcome of the best performing classifier.

For training/testing the ML algorithms, both random and TZ data are split with a 80/20 ratio: $80\,\%$ train/test data and $20\,\%$ for actually validating the facility location algorithms. To train and validate the ML algorithms, the exact solutions for up to 10 facilities are used; the MILP is actually executed on all topologies with $k = 1, ..., 10$, which produces the overall data. An ML algorithm is always trained for one set of graph types: one model for each random graph and one for the TZ graphs.

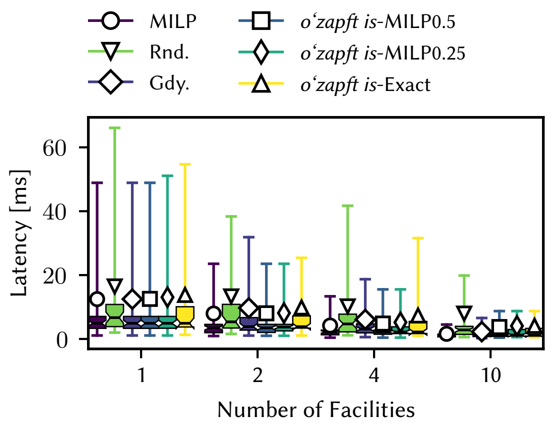
5.3.3.2 Evaluation

The evaluation part targets at answering the following questions:

- Using exact solutions, can we exactly predict facility locations?
- Can we efficiently reduce the search space?
- Which features are useful for prediction?

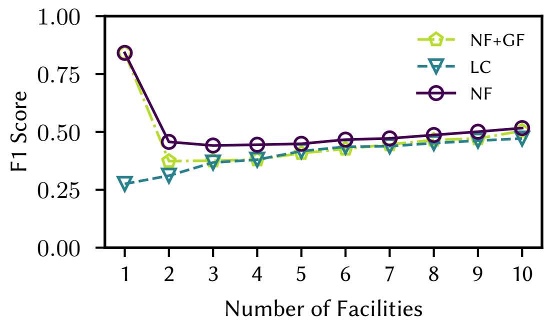


(a) Barabási-Albert (BA).

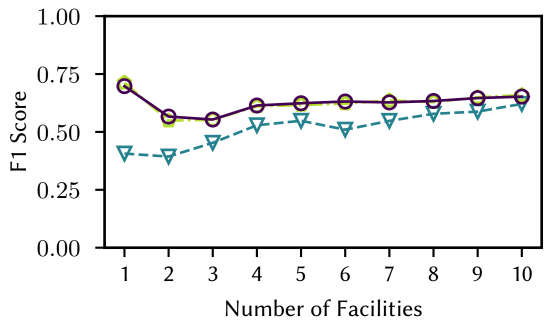


(b) Topology Zoo (TZ).

Figure 5.13: Subfigures show the latency values of the facility location algorithms among an increasing k (1,2,4,10). Left subfigure shows the results for BA substrates, right subfigure for the TZ substrates.



(a) Barabási-Albert (BA).



(b) Topology Zoo (TZ).

Figure 5.14: F1 score comparison for three feature complexities: NF, NF+GF, and LC. Increasing the number of facilities increases the F1 score.

Exact is hard. Learning optimal placements is a challenging goal, as shown for the latency values in Fig. 5.13. All results show that the MILP achieves the best performance as expected, and RND the worst; however, *o'zapft is*-Exact already comes next to RND among all scenarios. This means that the facility locations cannot be exactly learned and more research is required. Nevertheless, as we will see in the following, *o'zapft is* can be attractive for supporting exact algorithms.

Search space can be reduced. Fig. 5.13 illustrates that *o'zapft is*-MILP0.5 performs as good as the MILP for a small number of facilities (1-4) and as good as or better than GDY, while only scanning half of the substrate nodes. This holds for most combinations of substrate, facility number, and objective. *o'zapft is*-MILP0.25 shows a similar behavior, except for 10 facilities. The reason is that for 10 facilities, *o'zapft is*-MILP0.25 works like *o'zapft is*-Exact. While this only affects *o'zapft is*-MILP0.25 for BA topologies (Fig. 5.13a), *o'zapft is*-MILP0.5's's performance is worse for 10 facilities on real network topologies (Fig. 5.13b). The reason is that the TZ dataset contains also topologies with 20 nodes. For these topologies, even a ratio of 0.5 selects always only 10 nodes. Thus, the algorithms work like *o'zapft is*-Exact, which is in general not able to keep pace with GDY in such cases.

Potential for speed-up. We find that graph and complex node features are not important when predicting facility locations for random networks. Fig. 5.14 compares the F1 score of the true data for three feature settings: low complexity (LC), node (NF), and node+graph features (GF). The results demonstrate the need to differentiate among topology types and sources. While the F1 score for random networks is less affected, the TZ results show a more clear gap between low complexity features and more advanced graph/node features, where, e.g., betweenness centrality and eigenvector centrality are added.

We conclude that pre-selection with ML does not diminish the performance of optimal algorithms: an interesting angle to improve networking algorithms' efficiencies. However, the mechanisms cannot be simply generalized as shown for 10 facilities and small substrate networks; simply taking ML here might even diminish the performance when compared to the solution of a greedy algorithm.

5.3.4 Case Study II: Predicting the Costs of Virtual Network Requests

When facing the VNE problem, various measures are used to quantify VNE algorithms. Predicting such measures can be useful in many ways: for instance, when guessing the cost of a VNR, is it worth to execute an algorithm and accept it [GWZ+14].

5.3.4.1 Settings

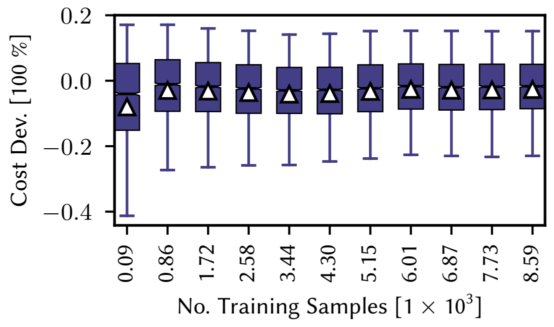
In the second case study, we analyze the potential of *o'zapft is* to automatically learn and predict the embedding costs of VNRs. Again, this section first introduces the analyzed VNE algorithms and provides information on the overall simulation and ML process and data.

Virtual network embedding process & data. We consider five frequently used substrate network types to evaluate VNE algorithms: graphs generated with the models ER and BA and three real topologies (a TZ graph, and the two data center topologies FT and BC). The parameter for the random network models are like introduced in Table 5.3. From the TZ [KNF+11], we choose the KDL topology: This network consists of 734 nodes and serves as an example of a large scale IP-layer access network deployed in Kentucky, USA. To study data center networks, we consider a 6-ary FT and a BC_2 with 4^2-port switches (DC-BC).

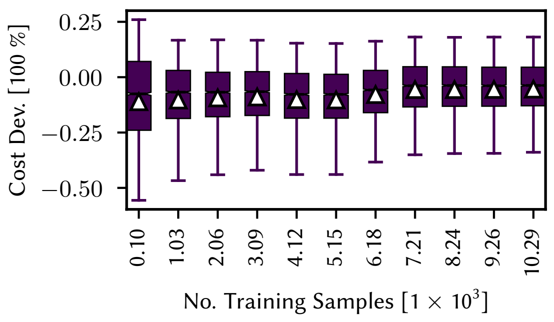
The VNRs arrive according to a Poisson process with average rate of 5 per 100 time units. The service processing times are exponentially distributed either with mean 500 time units for ER, BA and TZ graphs or with mean 1 000 time units for the data center topologies. The service times for data center topologies are increased to achieve higher and more realistic utilizations of the networks. For BA and ER substrate networks, VNRs have 2 to 20 nodes; for data center topologies, VNRs have 3 to 10 nodes; whereas when analyzing optimal algorithms, the number of nodes of VNRs ranges from 5 to 14.

For each setup, i.e., combination of VNE algorithm and embedding process, at least 5 runs with 2 500 VNRs each are generated to create the training data. For training and testing, only the accepted VNRs are used, as only those provide information on the true embedding cost.

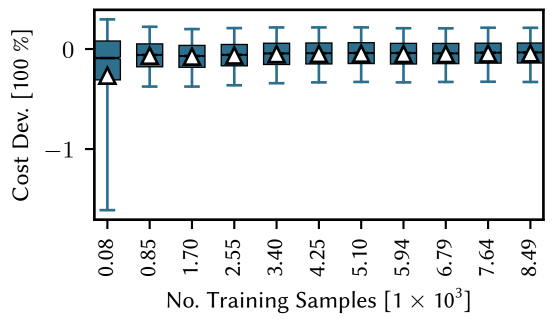
Virtual network embedding algorithms. We compare the two heuristics, GDY [YYR+08] and GRC [GWZ+14], and the optimal algorithms SDP and LB [MSK+13]. These algorithms are frequently



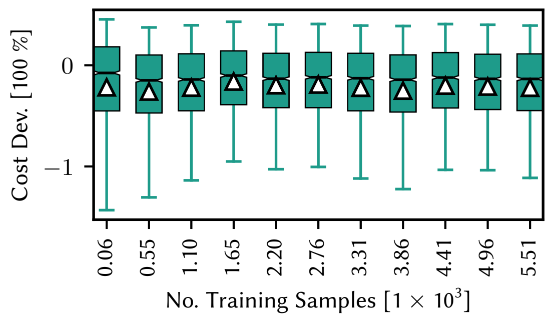
(a) Greedy.



(b) GRC.



(c) SDP.



(d) LB.

Figure 5.15: Boxplots of virtual network embedding cost prediction deviations (Cost Dev.) with random forest model in percentage as a function of training samples. *VNE algorithms:* GDY, GRC, SDP and LB. Note the different x-axis and y-axis scales. The more training data is available, the better becomes the performance for GDY, GRC and SDP. However, learning for the load balancing algorithm LB is harder, as more load-balanced-optimal solutions might generally exist.

studied in literature and hence serve as good initial representatives to demonstrate the feasibility of *o'zapft is*. Since the underlying problem is NP-hard and the optimal algorithms slow in large network topologies ($>$ 50 substrate nodes), we prematurely interrupted computations after timeouts of 30, 60, and 90 seconds.

Machine learning algorithms. We compare four ML regressors: Linear Regression (LR), Bayesian Ridge Regressor (BRR), Random Forest Regressor (RF), and Support Vector Regression (SVR).

Strawman. For comparison, we implemented a Strawman approach (SM). SM is based on the intuition that the number of VNR nodes and links are mostly related to the cost. As equal-sized VNRs are not always embedded similarly, we use the average among all VNRs of the training data. To predict the embedding cost, SM takes the number of virtual (requested) links and virtual nodes as input and calculates the cost based on a steadily interpolated 2D surface, which has been fitted to the training data.

5.3.4.2 Evaluation

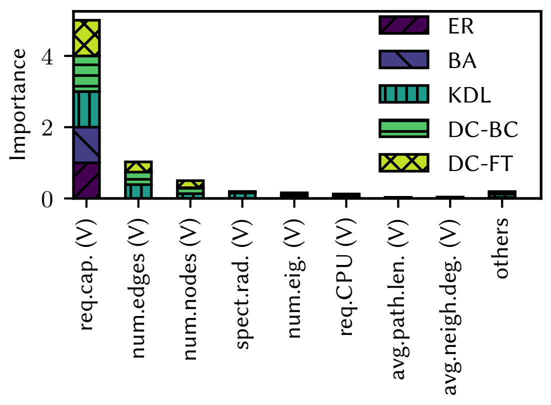
Tapping data is useful. We first investigate how much learning is required to make useful predictions of VNE costs. Our experiments show that already after a short training period, VNE costs can be estimated well. Fig. 5.15 shows boxplots of the (embedding) cost deviation over an increasing amount of training samples for all VNE algorithms. The whiskers of the boxplots show 90 % of the

Table 5.4: Regressors' performances for all VNE algorithms. Performance given via the achieved mean cost, Coefficient of determination (R^2) score and Root Mean Squared Error (RMSE) for ER substrates. ML models: Bayesian Ridge Regressor (BRR), Linear Regression (LR), Random Forest Regressor (RF), and Support Vector Regression (SVR).

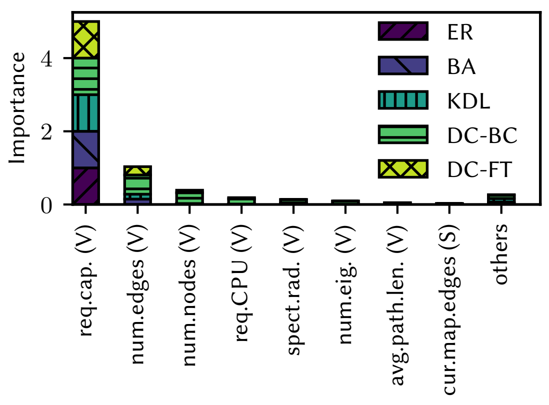
VNE Alg.	Mean Perf.	Regressors	R^2	RMSE
Greedy	1274.55	BRR	0.989	131.8
		RF	0.990	127.8
		LR	0.989	131.8
		SVR	0.990	125.1
		SM	-0.860	1710.3
GRC	1571.68	BRR	0.974	249.5
		RF	0.979	224.2
		LR	0.974	249.9
		SVR	0.983	203.3
		SM	-0.920	2129.3
SDP	1026.54	BRR	0.893	278.5
		RF	0.907	259.7
		LR	0.892	279.2
		SVR	0.898	272.2
		SM	-0.923	1180.4
LB	1803.45	BRR	0.635	670.2
		RF	0.638	666.7
		LR	0.620	683.8
		SVR	0.605	697.0
		SM	-0.964	1553.9

test data; we do not depict outliers.

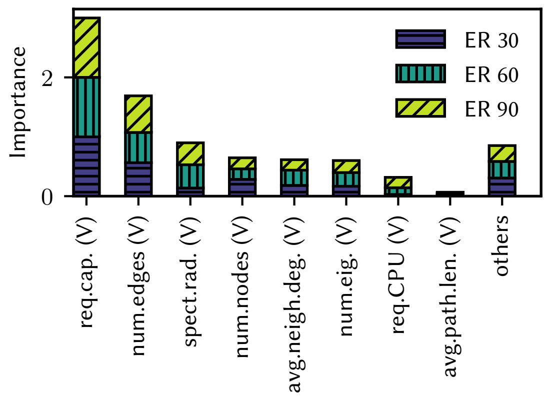
Interestingly, while the mean values are quite stable, models typically underestimate the cost of VNRs. The prediction quality of all models increases with more training data. For GDY, the accuracy for 90 % of the data improves from an underestimation of 40 % to an underestimation of 20 %. In case of the optimal algorithms SDP and LB, Figures 5.15c-5.15d demonstrate that even when increasing the number of training samples, both are suffering from outliers. For 50 % of the data, however, the deviations are comparable to the ones of the heuristic algorithms: they are in the range from an underestimation of 20 % to an overestimation of roughly 5 %.



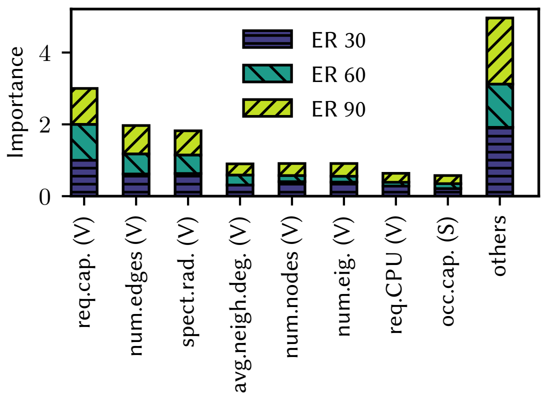
(a) Greedy.



(b) GRC.



(c) SDP.



(d) LB.

Figure 5.16: Feature importance of random forest models. VNR features are indicated via (V), substrate ones via (S). All remaining importances are accumulated in "others". For GDY and GRC, the results are shown for different substrate types. For SDP and LB, the results are shown for the timeouts of 30/60/90 seconds. The requested capacities are the dominating feature in case of GDY and GRC. For SDP, feature importance is more distributed while requested capacity is still very important. In case of LB, feature importance values are much more distributed; again, more good load-balanced solutions might exist, which makes it hard to predict the cost.

Let us next investigate to what extent the performance depends on the ML model, the substrate network and the VNE algorithm. To provide a more general comparison of models for different substrate networks and VNE algorithms, we report on the R^2, the goodness of fit of an ML model, and the RMSE in Tab. 5.4. In our experiments, we find that the ML models can achieve high R^2 scores between 0.8 and 0.99 across all substrate network types, while RF and SVR always achieve the highest values.

The Strawman (SM) approach generally shows the worst performance. Note the lower R^2 scores for SDP. The same observations hold for RMSE values, where values are in $30\,\%$ range of the mean performance. We find that while *o'zapft is* can predict the performance of heuristic algorithms well, the performance is worse for optimal algorithms: given the high complexity of the MILP solutions, this is not surprising (it is impossible to perfectly and quickly guess an optimal solution to an NP-hard problem). Yet, the solutions are promising, and they show the potential of the approach (and for the design of more advanced ML algorithms).

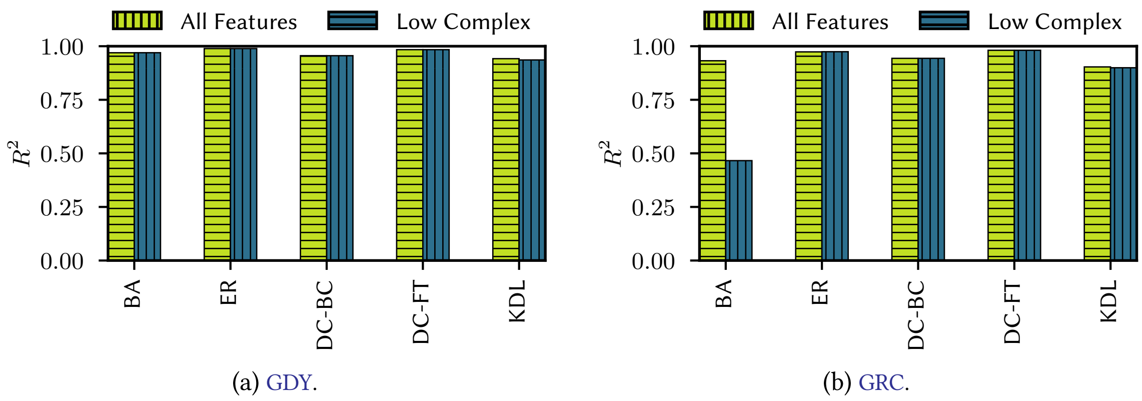


Figure 5.17: Comparison of R^2 of Linear Regression (LR) model for taking all features (*All Features*) versus taking only features with low complexity (*Low Complex*) for all substrate networks for GDY and GRC embedding algorithms. The linear model cannot learn with low complex features for GRC and BA topologies anymore. Missing features like the closeness centrality make it harder to achieve high R^2 values.

Speeding-up machine learning computations. *o'zapft is* comes with interesting flexibilities to speed up the networking algorithms further: perhaps a small number of low-complexity features are sufficient to obtain good approximations? First, to study which features are actually important for the solution learning, let us focus on the Random Forest (RF) model, which allows the examination of the importances of features. For GRC and GDY, we report on different substrate network types. For SDP and LB, we report on the feature importance for all timeout settings of 30/60/90 seconds.

Fig. 5.16 shows that the most important feature is the requested link capacity. For the BC data center topology, the number of edges and the number of nodes have a significant influence. Here, we cannot note that the timeout settings differ in the feature importances. Comparing SDP and LB to the heuristics, the feature importance is more distributed across all features. This is to be expected as optimal algorithms face a larger search space and hence the variation of solutions is larger.

So we may ask: is it even sufficient to focus on linear-time graph features to obtain accurate cost predictions? To find out, we trained our regressors with features that have complexity $O(n)$, $O(n+m)$ and $O(n \cdot \log n)$. As Fig. 5.17 shows for the R^2 score of LR, interestingly, even the low complex features provide high R^2 scores in particular for GDY (Fig. 5.17a). In case of GRC (Fig. 5.17b), however, the BA substrate prediction is negatively affected by the features choice and the simple linear regression model cannot compensate for some of the high-complexity features.

5.3.5 Case Study III: Predicting the Acceptance Probabilities of Virtual Network Requests

Alongside predicting the cost, there is another useful measure to predict: the acceptance of a VNR. When using optimal and time-costly algorithms, predicting the outcome of algorithms can save runtime. If a VNR is actually infeasible, it is not needed to create a potentially big problem instance of an optimal model. Moreover, computational resources can be saved if knowing that a VNR solution cannot be found in acceptable algorithm runtime. Saved resources could rather be spent on improving already embedded VNs.

Table 5.5: Grid Search parameters for training the recurrent neural network. Chosen values are printed in bold.

Parameter	Values
Optimizer	adam [KB14], RMSprop [JCS+15]
Size of hidden layer	100, **200**, 300
RMSprop step-rate	0.1, 0.01, **0.001**
RMSprop momentum	0.3, 0.6, **0.7**, 0.8, 0.9
RMSprop decay	0.3, 0.6, 0.7, 0.8, **0.9**

5.3.5.1 Settings

Virtual network embedding process & data. The substrate graphs are generated with the ER model with $50, 75$ and 100 nodes. The connection probability is 0.11. Substrate node and edge capacities are unit-less; their capacity is following a uniform distribution between 50 and 100. The VNRs are also generated with the ER model. Their connection probability is 0.5. The number of nodes are uniformly distributed between 5 and 14. Node and edge capacities are also uniformly distributed between 50 and 100. The inter-arrival time of the VNRs follow an exponential distribution (exp. distr.) with arrival rates $\lambda = \frac{1}{100}, \frac{3}{100}, \frac{5}{100}$. Each accepted VNR stays for an exponentially distributed lifetime with mean $1\,000$. This results in an overall training of 18 ML models. All chosen parameter settings in this thesis are in accordance with existing VNE research [CRB09; MSK+13; RB13]. For generating training data, 10 runs are conducted. The final admission control system is tested for at least one run.

Virtual network embedding algorithms. We investigate the SDP algorithm [MSK+13], which showed generally the best performance among all VNE algorithms (Tab.5.4). In addition, we use a modified version of the Melo algorithm which just seeks the first feasible solution, which we call Feasibility Seeking VNE Algorithm (FEAS). By using both algorithms, we want to investigate the ability of the ML model to predict VNE measures for two completely different algorithm behaviors. The algorithms are interrupted after pre-determined timeouts: 30, 60, and 90 seconds. Because of this, a VNR is classified as *feasible*, *infeasible*, or as *no solution* if a solver cannot find a feasible solution in time T. Note that this does not imply that the VNR is generally infeasible; it might be possible to find a solution after the time constraint T. An operator can use the time constraint T to trade off embedding quality and needed computational runtime. Adding the time to the algorithm name denotes the different variants; SDP30 has a timeout of 30 seconds.

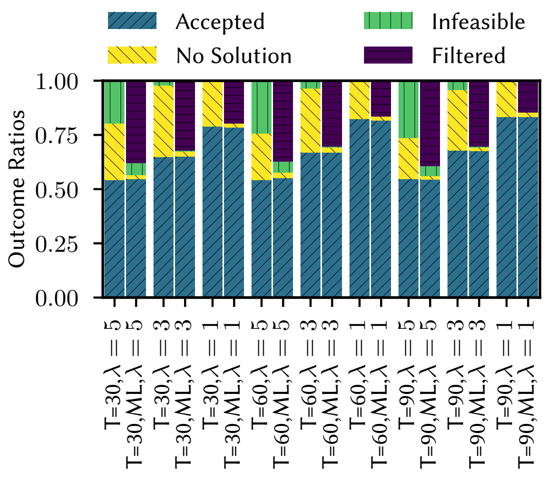
Machine learning algorithm. The admission control systems uses a Recurrent Neural Network (RNN) for classifying VNRs. The RNN consists of one hidden layer with 200 hidden units. The output transfer function is the sigmoid function $\sigma(x) = (1 + e^{-x})^{-1}$, the hidden transfer function is the hyperbolic tangent function $\tanh(x) = (e^{2x} - 1)(e^{2x} + 1)^{-1}$, the loss optimized on is the Bernoulli cross-entropy loss $L(c \mid y) = -z \log(y) - (1-z) \log(1-y)$, where $z \in \{0, 1\}$ is the class label and y

Table 5.6: Prediction model performance for SDP and FEAS VNE algorithms for three substrate and timing settings. ER50-SDP30 denotes an ML model trained for ER-based substrate networks with 50 nodes, SDP algorithm, and $T = 30\,s$. All models show an accuracy that is higher than the Majority Vote (MV). High True Positive Rates (TPR) and True Negative Rates (TNR) confirm that the model can learn the classification task very well.

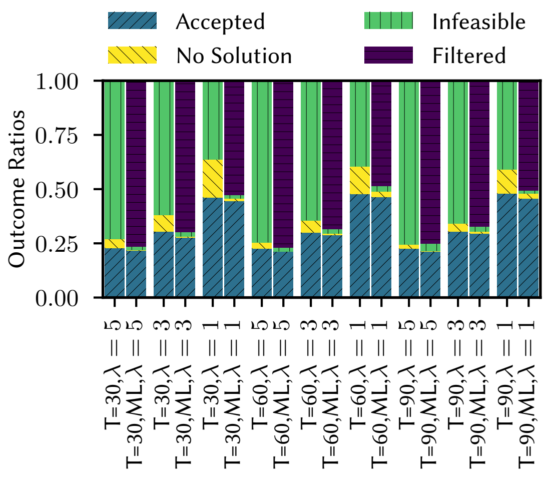
RNN Model	**Accuracy [%]**	**TPR [%]**	**TNR [%]**	**MV [%]**
ER50-SDP30	92.70	86.57	95.35	69.86
ER50-SDP60	92.60	87.88	94.98	66.53
ER50-SDP90	92.91	89.71	94.59	65.61
ER75-SDP30	90.57	88.68	92.39	50.89
ER75-SDP60	90.52	91.88	89.16	50.03
ER75-SDP90	89.64	89.85	89.44	50.23
ER100-SDP30	90.61	92.36	87.85	61.26
ER100-SDP60	90.30	92.97	85.87	62.45
ER100-SDP90	90.15	91.82	87.31	62.91
ER50-FEAS30	97.77	92.30	98.78	84.38
ER50-FEAS60	98.02	92.88	99.01	83.93
ER50-FEAS90	97.68	92.33	98.68	84.17
ER75-FEAS30	96.24	88.98	98.29	77.97
ER75-FEAS60	96.34	90.10	98.03	78.62
ER75-FEAS90	96.22	89.10	98.17	78.46
ER100-FEAS30	94.21	87.52	96.70	72.87
ER100-FEAS60	93.48	86.39	96.26	71.79
ER100-FEAS90	92.86	87.61	94.89	72.01

the prediction. Two optimizers are used to train the RNN: the adam optimizer or the RMSprop optimizer. The RMSprop optimizer is taken from the publicly available *climin* [JCS+15] library. The RNN is implemented in the publicly available *breze* library [JCM+16]. Based on a grid search among the parameters that Table 5.5 describes, the best setting of RMSprop is a step-rate of 0.001, a momentum of 0.7, and a decay of 0.9.

Runtime measure. The optimal VNE algorithm uses Gurobi [GO15] to solve the mathematical model. This process consists of two potentially time-consuming parts: the model creation and the actual model solving. In case of using the admission control system, the overall runtime also includes the feature calculations $\phi_V(G_t^v)$ and $\phi_S(G_{t-1}^s)$ and the calculation of the prediction value.



(a) SDP VNE Algorithm.



(b) Feasibility VNE Algorithm.

Figure 5.18: Outcome ratios (accepted, no solution, infeasible, filtered) of VNE algorithms with and without admission control. The four colors represent the ratios of accepted (blue), no solution (red), infeasible (green), and filtered (purple) ratios of VNRs by the system, i.e., by the admission control and the VNE algorithm. ER substrate network with size 100. For both algorithms, the admission control filters requests that do not provide a solution in acceptable time (*no solution*) or that are *infeasible*.

5.3.5.2 Evaluation

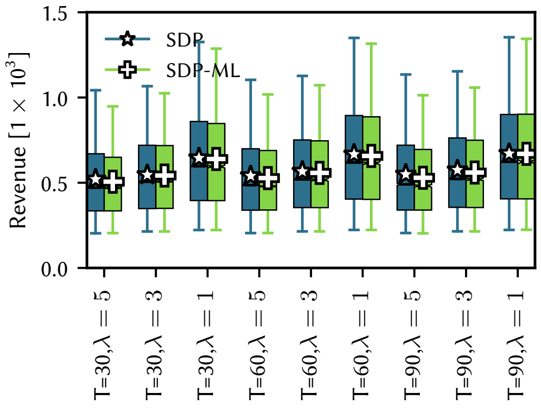
The evaluation answers the question whether ML can actually help to improve algorithm efficiency. As a proof of concept, it targets at improving the performance of an optimal algorithm; optimal VNE algorithms have the highest potential for runtime improvement. In contrast to optimal algorithms, heuristic algorithms may already have a quite low runtime; however, the question whether ML is beneficial here has initially been addressed in [HKB+17].

Are we better than a majority vote? We first inform on the performance of our ML algorithm. The question is whether the ML algorithm is better than a majority vote, i.e., either simply rejecting or accepting all VNRs. Table 5.6 summarizes the results of all trained models. It shows the accuracy, the true positive rate and the true negative rate. The accuracies of the models are between 89 % and 98 % in all scenarios: they are better than the respective majority votes - the ML predicts successfully the outcome of the algorithms.

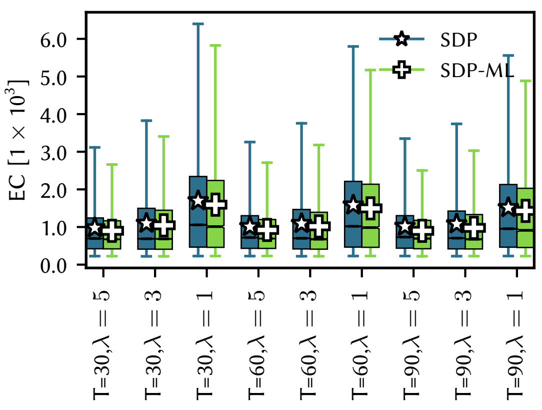
Generally, the datasets are very skewed: the majority vote (MV) can already achieve up to 84.38 % as shown for ER50-FEAS30. Nevertheless, the ML algorithm has a higher accuracy; it beats the majority vote. Moreover, we observe that the results are similar for the different network sizes. Hence, we focus on a substrate size of 100 nodes.

Figs. 5.18a-5.18b differentiate between the different classes of the VNR classification. The different colors represent the ratios of accepted (blue), no solution (yellow) and infeasible (green) VNRs. The ratio of VNRs, which are filtered by the admission control system, is shown in purple. We first note general observations for increasing timeouts and lambda values.

An increasing timeout parameter does not significantly improve the acceptance ratio for the same lambda values; the optimal algorithm does not benefit from additional solver time here. Besides,



(a) Revenue REV.



(b) Virtual Network Embedding Cost EC.

Figure 5.19: VNE metrics of SDP for ER substrate with network size 100. The graphs show boxplots with whiskers ranging from the lower 2.5 % to the upper 97.5 % of the REV and EC values; outliers are not shown. Results with and without admission control are similar. Using ML does not decrease the VNE performance in terms of revenue and cost.

decreasing the lambda values leads to more accepted VNRs; the substrate is less populated, hence, more VNRs can be accepted. A larger lambda value leads to more infeasible VNRs as the substrate is more densely populated.

Independent of this fact, the ML algorithm achieves a high accuracy among all classes; it forwards *feasible solutions* and successfully filters *infeasible solutions* and *no solutions*. The acceptance ratio slightly varies around the true acceptance ratio; it is either a bit higher with 2.37 % for SDP, $T = 60, \lambda = 5$ or slightly lower with 7.01 % for FEAS, $T = 60, \lambda = 5$. As FEAS generally shows a low AR, we continue our study with the SDP algorithm as we are generally interested in speeding up algorithms solving the VNE problem with good performance.

Do we keep the virtual network embedding performance? Figures 5.19a-5.19b provide a comparison between SDP and the admission control system for the two metrics REV and EC. Whereas the parameter λ affects the absolute metrics, the additional timeout parameter shows only a minor affect. Independent of both variables, the ML algorithm does not significantly diminish REV or EC.

When looking at the RCR illustrated in Fig. 5.20, it can be noted that using the prediction improves the RCR slightly. The ML algorithm tends to reject time-consuming VNRs; in addition, it rejects the VNRs which potentially use the available solving time T, i.e., which might come close to the threshold. Generally, requests that tend to consume more solving time also have a lower RCR. An interesting aspect that should be investigated deeper in future studies of systems combining optimal algorithms and ML.

What is the runtime gain? In order to provide deeper insights into the runtime gain, Fig. 5.21 reports on the mean modeling times and the mean solving times for all VNRs. As Fig. 5.21a depicts, the gain in modeling time is significant: this is again independent of the timeout T and the arrival rate λ. When looking at the solving time in Fig. 5.21b, the previous observation can be confirmed:

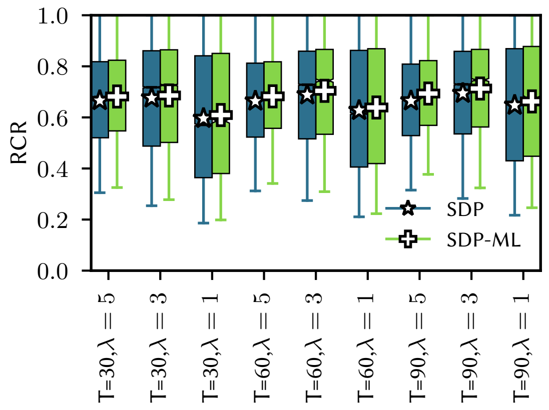
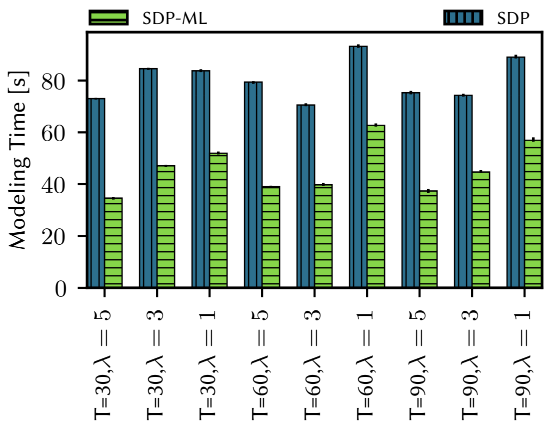
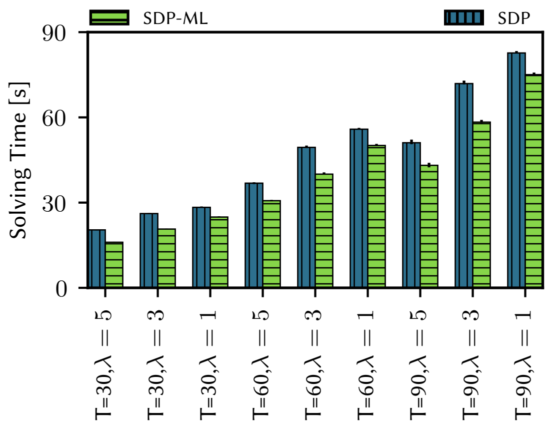


Figure 5.20: Boxplots of RCR over combinations of timeout settings and arrival processes. Boxplots show the RCR of SDP with ML (SDP-ML) and without SDP. Interestingly, the ML system slightly increases the average RCR. Apart from that, it does not significantly improve or worsen the overall RCR.



(a) Modeling Time.



(b) Solving Time.

Figure 5.21: Modeling time and solving time over combinations of timeout settings and arrival processes for SDP with and without ML. The left subfigure shows the average model creation time (Modeling Time in seconds); the right subfigure shows the average runtime of the solver (Solving Time in seconds) to process a VNR. Using ML can reduce modeling time by 1.4 times ($T = 60, \lambda = 1$) up to 2.1 times ($T = 30, \lambda = 5$). On average, the algorithms generally do not use the available solving time.

the admission control system filters time-consuming VNRs, i.e., the ones that exhaust the available timeout threshold. It can be noted that by using the admission control system, up to 50 % of overall computational time can be saved without affecting the overall VNE metrics.

To summarize, the admission control improves algorithm efficiencies while it does not significantly diminish the overall system performance in terms of VNE metrics.

5.4 Summary

Rapidly and efficiently solving network optimization problems is inevitable for next generation networks. With the increase in flexibility due to SDN and NV, resource management of those systems faces new challenges: changing resource assignments at runtime on short timescales. As a consequence, we see ML as one elementary concept that helps improve the efficiency of systems facing complex optimization and resource management problems.

The first system concept, called *NeuroViNE*, is a preprocessor for VNE algorithms. *NeuroViNE* uses a Hopfield network to subtract subgraphs that have a high potential to host VNRs efficiently. Any VNE algorithm can be used to conduct the final embedding on the deduced subgraph. Specifically real network topologies, such as wide area networks or data center networks, show structures which *NeuroViNE* can exploit efficiently to extract better subgraphs. Further, because *NeuroViNE* relies on a neural network data structure that can be parallelized, it can be implemented with a high computational efficiency.

The second system concept, called *o'zapft is*, shows that a supervised learning approach can be used to improve the efficiency of networking algorithms. By simply considering the produced data of networking algorithms, which are represented compactly by graph and node features, *o'zapft is* can predict the outcome of time consuming optimization processes: e.g., the potential objective values or the feasibilities of an optimization problem. In summary, *o'zapft is* improves the computational efficiency of optimization systems, e.g., it decreases the runtime by more than $50\,\%$ for optimal algorithms.

Chapter 6

Conclusion and Outlook

Network operators need mechanisms that allow communication networks to be adapted in case of new demands arising from new services, applications, and user expectations. The virtualization of software-defined networks is a tremendous step towards making today's networks ready for adaptation. Virtualized software-defined networks combine the benefits of Network Virtualization (NV) and Software-Defined Networking (SDN). Whereas NV runs multiple virtual networks whose resources are tailored towards different service and application demands in parallel on one infrastructure, SDN makes it possible to easily program virtual networks at runtime.

While both concepts NV and SDN proclaim to broaden the dimensions of adaptability and flexibility, combining them comes with new challenges. Within the framework of virtualizing software-defined networks, tenants do not only share resources on the data plane, but also on the control plane; therefore, tenants can not only interfere on the data plane resources of networks, but also on the control plane resources. Measurement procedures need to identify existing shortcomings of SDN-based architectures realizing the sharing of networking resources; at the same time new system architectures are needed to mitigate the potentially existing shortcomings.

With the virtualization of software-defined networks comes an additional layer; this layer is usually implemented by software-based network hypervisors, which provide the control logic for the virtualization tasks. Since any commodity server can host network hypervisors, the control logic can be distributed and even migrated among a network at runtime. However, exploiting this new freedom of placement is algorithmically hard; in contrast to the placement problem of SDN controllers in non-virtualized networks, the placement of network hypervisors adds the dimension of serving multiple vSDNs at the same time.

Virtualizing software-defined networks does not only require efficient implementations and algorithms for control plane resource management, but also fast and efficient provisioning of virtual networks on the data plane. Since solving the problem of sharing resources among virtual networks is NP-hard, algorithm designers need new ways to speed up provisioning times in order to take advantage of the operational flexibility of SDN and NV. Mechanisms that exploit the produced data of algorithms, such as Machine Learning (ML), might open new paths towards improving the efficiency of networking algorithms.

6.1 Summary

This work addresses three important challenges of virtualizing software-defined networks. The first part studies the real SDN network hypervisor implementations and proposes mechanisms to circumvent existing shortcomings of these implementations. The second part studies the placement problem of SDN virtualization layers. The third part focuses on the study of neural computation and ML-based systems for improving the efficiency of network algorithms for virtual network environments.

Measurement and design of virtualization layer. The virtualization layer with its realization via SDN network hypervisors is the integral part when virtualizing software-defined networks. In order to analyze the existing SDN network hypervisor deployments, we propose a benchmarking tool for SDN network hypervisors: a missing ingredient for benchmarking virtual SDN environments so far. Whereas existing benchmark tools for OpenFlow (OF)-based networks are not designed for virtualized SDN environments, our new tool puts efficient benchmarks into effect. Deep benchmarks on two existing hypervisors, FlowVisor (FV) and OpenVirteX (OVX), present important characteristics and uncover potential issues. Both hypervisors show varying performance trade-offs with respect to how they realize the virtualization of SDN networks. Whereas FV provides a more predictable operation, i.e., its processing shows less variation with respect to control plane latency, OVX offers higher scalability at the cost of less predictability.

Furthermore, we identify control plane isolation and adaptability as missing pieces of existing hypervisor concepts. Therefore, we propose HyperFlex - an SDN virtualization framework towards flexibility, adaptability and improved control plane predictability. By realizing virtualization functions in software or/and in hardware, HyperFlex increases the flexibility when operating a virtualization layer. Using network or software isolation, HyperFlex improves the predictability of the control plane operation of virtual network tenants connected to FV. Moreover, we identify a new challenge when adapting virtualization layers: reconfiguration events that increase the control plane latency and, as a consequence, worsen the predictability of operating virtual networks.

Deployment optimization of virtualization layer. Putting the control logic into software, virtualizing SDN networks offers a new dimension of where to place the control logic. With SDN, any commodity server can host software running the control logic of SDN virtualization layers. In order to analyze the various trade-offs between existing hypervisor architectures, we provide a first model for the hypervisor placement problem facing static network scenarios. With this model targeting at minimum control plane latencies, we analyze design trade-offs: e.g., from the number of hypervisor instances to the deployment of multi-controller switches. As a result, up to $50\,\%$ multi-controller switches and $k = 5$ hypervisor instances are enough to provide an optimal control plane latency.

In order to accommodate the full flexibility of virtual SDN networks, e.g., the embedding of virtual SDN networks at runtime, we extend our model for static network use cases towards dynamic traffic scenarios. Beside targeting control plane latency, our model also takes reconfigurations of the virtualization layer into account; minimizing reconfigurations yields more predictable network operations. Based on a Pareto analysis, we show the trade-offs between control plane latency and the total number of reconfigurations. To summarize, accepting up to 8 % higher latency can already put the amount of needed reconfigurations close to 0.

Improving virtual network provisioning. Combining SDN and NV offers new dimensions of faster and more flexible network operation. Hence, mechanisms are needed that improve the efficiencies of VNE algorithms to fast and quickly provide virtual networks. We propose systems using neural computation and ML for improving the efficiency of networking algorithms in general, and VNE algorithms in particular. Our first system *NeuroViNE* neurally computes subgraphs yielding a high embedding efficiency; virtual nodes and links can be efficiently placed on these subgraphs. Hence, *NeuroViNE* improves embedding footprints and shortens the runtime of optimal algorithms by shrinking the search space.

Our second system is *o'zapft is* - a data-driven approach to designing network algorithms. *o'zapft is* learns in a supervised fashion from solutions of network problems; thereby, it can guess good initial solutions for unseen problem instances. *o'zapft is* reduces the search space and/or predicts the feasibilities of problem instances to save runtime. For the VNE problem, we demonstrate how *o'zapft is* helps optimal algorithms to gain speed ups by up to 50 %.

6.2 Future Work

We believe that the following future work might be of particular interest.

Reconfiguration analysis of SDN network hypervisors for complex network topologies. Virtualization of SDN networks require complex operations: topology abstraction, data plane addressing isolation, state and diverse hardware abstractions etc. So far, the benchmarks consider rather simple topologies with one switch only, while they have analyzed a wide range of OF message types, hypervisor implementations, and multi-tenant setups. Yet, benchmarking hypervisors and, in particular, virtualization functions demands varying performance benchmark setups such as complex network topologies. Since the number of different setups is large, smart mechanisms are needed that can still provide useful insights into the performance of hypervisors.

Capacitated hypervisor placement problems. Although control plane latency is seen as one major impact factor when managing SDN networks, placement optimizations must be extended to scenarios where network and computational capacities are limited. Understanding the behavior of capacity limitations is an important step towards improving virtualization layers for SDN networks.

Preparedness of placements - towards flexibility and empowerment. The proposed placement models can be used to investigate the trade-off between control plane latency and reconfigurations. Beside analyzing this trade-off, virtualization layers should be optimized with respect to flexibility; it is expected that a more flexible hypervisor design yields advantages when adapting to demand changes or new incoming virtual network requests. Instead of steering a hypervisor placement always to a latency optimal solution, it could also be steered towards integrating Empowerment [SGP14]: empowered solutions are expected to react to changes more efficiently.

Problem and solution representation for learning. Interesting future work might dig deeper into the way how to represent structured data, i.e., problems and solutions, for learning as many proposals already exist in literature: graph kernels, random walks, or graph embeddings that are inspired by graphical model inference algorithms. Furthermore, the data-driven optimization should be extended to other algorithms for SDN: e.g., an initial proposal has demonstrated its extension to the SDN controller placement problem [HKB+17]. Above all, the proposed system should be extended to further network optimization problems and research areas including routing, QoS management, wireless network resource management etc.

From offline to online learning. The proposed ML models so far were only trained offline: they are not able to capture online changes of data patterns. Accordingly, mechanisms are needed that can efficiently update the ML models at runtime. Other ML concepts, such as reinforcement learning or advanced neural network architectures, open further potential optimization improvements. At the same time, it is important to better understand the limitations of ML-based approaches.

Bibliography

Publications by the Author

Book Chapters

[SBD+18] Susanna Schwarzmann, Andreas Blenk, Ognjen Dobrijevic, Michael Jarschel, Andreas Hotho, Thomas Zinner, and Florian Wamser. "Big-Data Helps SDN to Improve Application Specific Quality of Service." In: *Big Data and Software Defined Networks*. Ed. by Javid Taheri. The Institution of Engineering and Technology (IET), January 2018, pp. 433–455.

Journal Papers

[BBD+18] Arsany Basta, Andreas Blenk, Szymon Dudycz, Arne Ludwig, and Stefan Schmid. "Efficient Loop-Free Rerouting of Multiple SDN Flows." In: *IEEE/ACM Trans. on Netw. (ToN)* 26.2 (April 2018), pp. 948–961.

[BBH+17] Arsany Basta, Andreas Blenk, Klaus Hoffmann, Hans Jochen Morper, Marco Hoffmann, and Wolfgang Kellerer. "Towards a Cost Optimal Design for a 5G Mobile Core Network based on SDN and NFV." In: *IEEE Trans. on Netw. and Serv. Manage.* 14.4 (December 2017), pp. 1061–1075.

[BBR+16] Andreas Blenk, Arsany Basta, Martin Reisslein, and Wolfgang Kellerer. "Survey on Network Virtualization Hypervisors for Software Defined Networking." In: *IEEE Commun. Surveys & Tutorials* 18.1 (2016), pp. 655–685.

[BBZ+16] Andreas Blenk, Arsany Basta, Johannes Zerwas, Martin Reisslein, and Wolfgang Kellerer. "Control Plane Latency With SDN Network Hypervisors: The Cost of Virtualization." In: *IEEE Trans. on Netw. and Serv. Manage.* 13.3 (September 2016), pp. 366–380.

[KBB+18] Wolfgang Kellerer, Arsany Basta, Péter Babarczi, Andreas Blenk, Mu He, Markus Kluegel, and Alberto Martínez Alba. "How to Measure Network Flexibility? - A Proposal for Evaluating Softwarized Networks." In: *IEEE Commun. Mag.* PP.99 (2018), pp. 2–8.

[WBS+15] Florian Wamser, Andreas Blenk, Michael Seufert, Thomas Zinner, Wolfgang Kellerer, and Phuoc Tran-Gia. “Modelling and performance analysis of application-aware resource management.” In: *International Journal of Network Management* 25.4 (May 2015). nem.1894, pp. 223–241.

Conference Papers

[BBB+15] Arsany Basta, Andreas Blenk, Hassib Belhaj Hassine, and Wolfgang Kellerer. “Towards a dynamic SDN virtualization layer: Control path migration protocol.” In: *Proc. IFIP/IEEE CNSM*. Barcelona, Spain: IEEE, November 2015, pp. 354–359.

[BBK15] Andreas Blenk, Arsany Basta, and Wolfgang Kellerer. “HyperFlex: An SDN virtualization architecture with flexible hypervisor function allocation.” In: *Proc. IFIP/IEEE Int. Symp. IM Netw.* Ottawa, ON, Canada: IEEE, May 2015, pp. 397–405.

[BBL+15] Arsany Basta, Andreas Blenk, Yu-Ting Lai, and Wolfgang Kellerer. “HyperFlex: Demonstrating control-plane isolation for virtual software-defined networks.” In: *Proc. IFIP/IEEE Int. Symp. IM Netw.* Ottawa, ON, Canada: IEEE, May 2015, pp. 1163–1164.

[BBZ+15] Andreas Blenk, Arsany Basta, Johannes Zerwas, and Wolfgang Kellerer. “Pairing SDN with network virtualization: The network hypervisor placement problem.” In: *Proc. IEEE NFV-SDN*. San Francisco, CA, USA, November 2015, pp. 198–204.

[BK13] Andreas Blenk and Wolfgang Kellerer. “Traffic pattern based virtual network embedding.” In: *Proc. ACM CoNEXT Student Workshop*. Santa Barbara, California, USA: ACM, December 2013, pp. 23–26.

[BKJ+18] Andreas Blenk, Patrick Kalmbach, Michael Jarschel, Stefan Schmid, and Wolfgang Kellerer. “NeuroViNE: A Neural Preprocessor for Your Virtual Network Embedding Algorithm.” In: *Proc. IEEE INFOCOM*. Honolulu, HI, USA, April 2018, pp. 1–9.

[BKS+16] Andreas Blenk, Patrick Kalmbach, Patrick Van Der Smagt, and Wolfgang Kellerer. “Boost Online Virtual Network Embedding : Using Neural Networks for Admission Control.” In: *Proc. IFIP/IEEE CNSM*. Montreal, QC, Canada, October 2016, pp. 10–18.

[BKS+17] Andreas Blenk, Patrick Kalmbach, Stefan Schmid, and Wolfgang Kellerer. “*o’zapft is:* Tap Your Network Algorithm‘s Big Data!” In: *Proc. ACM SIGCOMM Workshop Big-DAMA*. Los Angeles, CA, USA: ACM, August 2017, pp. 19–24.

[BKW+13] Andreas Blenk, Wolfgang Kellerer, Florian Wamser, and Thomas Zinner. “Dynamic HTTP download scheduling with respect to energy consumption.” In: *Proc. Tyrrhenian Int. Workshop on Dig. Commun. - Green ICT (TIWDC)*. Genoa, Italy: IEEE, September 2013, pp. 1–6.

[DBK15] Raphael Durner, Andreas Blenk, and Wolfgang Kellerer. “Performance study of dynamic QoS management for OpenFlow-enabled SDN switches.” In: *Proc. IEEE IWQoS*. Portland, OR, USA: IEEE, June 2015, pp. 177–182.

[HBB+17a] Mu He, Arsany Basta, Andreas Blenk, and Wolfgang Kellerer. “How flexible is dynamic SDN control plane?” In: *Proc. IEEE INFOCOM Workshop (INFOCOM WKSHPS)*. Atlanta, GA, USA, April 2017, pp. 689–694.

[HBB+17b] Mu He, Arsany Basta, Andreas Blenk, and Wolfgang Kellerer. “Modeling flow setup time for controller placement in SDN: Evaluation for dynamic flows.” In: *Proc. IEEE ICC*. Paris, France, May 2017, pp. 1–7.

[HKB+17] Mu He, Patrick Kalmbach, Andreas Blenk, Stefan Schmid, and Wolfgang Kellerer. “Algorithm-Data Driven Optimization of Adaptive Communication Networks.” In: *Proc. IEEE ICNP Workshop on Machine Learning and Artificial Intelligence in Computer Networks*. Toronto, ON, Canada: IEEE, October 2017, pp. 1–6.

[KBB16] Wolfgang Kellerer, Arsany Basta, and Andreas Blenk. “Using a flexibility measure for network design space analysis of SDN and NFV.” In: *Proc. IEEE INFOCOM Workshop*. IEEE, April 2016, pp. 423–428. arXiv: 1512.03770.

[KBK+17] Patrick Kalmbach, Andreas Blenk, Markus Klügel, and Wolfgang Kellerer. “Generating Synthetic Internet- and IP-Topologies using the Stochastic-Block-Model.” In: *Proc. IFIP/IEEE Int. Workshop on Analytics for Netw. and Serv. Manage. (AnNet)*. Lisbon, Portugal, May 2017, pp. 911–916.

[KBK+18] Patrick Kalmbach, Andreas Blenk, Wolfgang Kellerer, and Stefan Schmid. “Themis: A Data-Driven Approach to Bot Detection (Short Abstract).” In: *Proc. IEEE INFOCOM*. Honolulu, HI, USA, 2018, pp. 1–2.

[KZB+18] Patrick Kalmbach, Johannes Zerwas, Andreas Blenk, Wolfgang Kellerer, and Stefan Schmid. “Empowering Self-Driving Networks.” In: *Proc. ACM SIGCOMM Workshop on Self-Driving Networks (SelfDN)*. accepted for publication. Budapest, Hungary, 2018, pp. 1–6.

[SBB+16a] Christian Sieber, Arsany Basta, Andreas Blenk, and Wolfgang Kellerer. “Online resource mapping for SDN network hypervisors using machine learning.” In: *Proc. IEEE NetSoft Conf. and Workshops (NetSoft)*. April. Seoul, South Korea: IEEE, June 2016, pp. 78–82.

[SBB+16b] Christian Sieber, Andreas Blenk, Arsany Basta, and Wolfgang Kellerer. “hvbench: An open and scalable SDN network hypervisor benchmark.” In: *Proc. IEEE NetSoft Conf. and Workshops (NetSoft)*. Seoul, South Korea: IEEE, June 2016, pp. 403–406.

[SBB+16c] Christian Sieber, Andreas Blenk, Arsany Basta, David Hock, and Wolfgang Kellerer. “Towards a programmable management plane for SDN and legacy networks.” In: *Proc. IEEE NetSoft Conf. and Workshops (NetSoft)*. Seoul, South Korea: IEEE, June 2016, pp. 319–327.

[SBH+15a] Christian Sieber, Andreas Blenk, Max Hinteregger, and Wolfgang Kellerer. “The cost of aggressive HTTP adaptive streaming: Quantifying YouTube’s redundant traffic.” In: *Proc. IFIP/IEEE Int. Symp. IM Netw.* Ottawa, ON, Canada: IEEE, May 2015, pp. 1261–1267.

[SBH+15b] Christian Sieber, Andreas Blenk, David Hock, Marc Scheib, Thomas Hohn, Stefan Kohler, and Wolfgang Kellerer. “Network configuration with quality of service abstractions for SDN and legacy networks.” In: *Proc. IFIP/IEEE Int. Symp. IM Netw.* Ottawa, ON, Canada: IEEE, May 2015, pp. 1135–1136.

[SWD+11] Barbara Staehle, Florian Wamser, Sebastian Deschner, Andreas Blenk, Dirk Staehle, Oliver Hahm, Nicolai Schmittberger, and G Mesut. “Application-Aware Self-Optimization of Wireless Mesh Networks with AquareYoum and DES-SERT.” In: *Proc. Würzburg Workshop on IP: Joint ITG and Euro-NF Workshop "Visions of Future Generation Networks"*. Würzburg, Germany, August 2011, pp. 3–4.

[ZKF+18] Johannes Zerwas, Patrick Kalmbach, Carlo Fuerst, Arne Ludwig, Andreas Blenk, Wolfgang Kellerer, and Stefan Schmid. “Ahab: Data-Driven Virtual Cluster Hunting.” In: *Proc. IFIP Networking*. Zurich, Switzerland, May 2018, pp. 1–9.

Technical Reports

[BBK+17] Arsany Basta, Andreas Blenk, Wolfgang Kellerer, and Stefan Schmid. “Logically Isolated, Actually Unpredictable? Measuring Hypervisor Performance in Multi-Tenant SDNs.” In: *Computer Research Repository (CoRR)* (April 2017), pp. 1–7. arXiv: 1704.08958.

[ZBK16] Johannes Zerwas, Andreas Blenk, and Wolfgang Kellerer. “Optimization Models for Flexible and Adaptive SDN Network Virtualization Layers.” In: *Computer Research Repository (CoRR)* V (November 2016), pp. 1–4. arXiv: 1611.03307.

General Publications

[802] *IEEE 802.1AB - Station and Media Access Control Connectivity Discovery.* 2005.

[ACK+16] Edoardo Amaldi, Stefano Coniglio, Arie M.C.A. Koster, and Martin Tieves. “On the computational complexity of the virtual network embedding problem.” In: *Electronic Notes in Discrete Mathematics* 52.10 (June 2016), pp. 213–220.

[ADG+14] Ali Al-Shabibi, Marc De Leenheer, Matteo Gerola, Ayaka Koshibe, William Snow, and Guru Parulkar. “OpenVirteX: A Network Hypervisor.” In: *Proc. USENIX Open Netw. Summit (ONS).* Santa Clara, CA, March 2014, pp. 1–2.

[ADK14] Niels L. M. van Adrichem, Christian Doerr, and Fernando A. Kuipers. “OpenNetMon: Network monitoring in OpenFlow Software-Defined Networks.” In: *Proc. IEEE/IFIP NOMS.* IEEE, May 2014, pp. 1–8.

[AEE15] Madyan Alsenwi, Hany Elsayed, and Mona Elghoneimy. “Optimization of channel selection in cognitive heterogeneous wireless networks.” In: *11th ICENCO.* December 2015, pp. 38–43.

[AFG+09] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. *Above the Clouds : A View of Cloud Computing*. Tech. rep. 2009, pp. 1–25.

[AFG+10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. "A View of Cloud Computing." In: *Commun. ACM* 53.4 (April 2010), pp. 50–58.

[AFL08] Mohammad Al-Fares and Amin Loukissas Alexanderand Vahdat. "A scalable, commodity data center network architecture." In: *Proc. ACM SIGCOMM*. Seattle, WA, USA: ACM, August 2008, pp. 63–74.

[AIS+14] Patrick Kwadwo Agyapong, Mikio Iwamura, Dirk Staehle, Wolfgang Kiess, and Anass Benjebbour. "Design considerations for a 5G network architecture." In: *IEEE Commun. Mag.* 52.11 (2014), pp. 65–75.

[AKE+12] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, and Balaji Prabhakar. "Less is More : Trading a little Bandwidth for Ultra-Low Latency in the Data Center." In: *Proc. USENIX Symp. NSDI*. San Jose, CA: USENIX Association, April 2012, pp. 19–19.

[AKH92] Shigeo Abe, Junzo Kawakami, and Kotaroo Hirasawa. "Solving inequality constrained combinatorial optimization problems by the hopfield neural networks." In: *Neural Networks* 5.4 (1992), pp. 663–670.

[ALL+96] Karen Aardal, Martine Labbé, Janny Leung, and Maurice Queyranne. "On the Two-Level Uncapacitated Facility Location Problem." In: *INFORMS Journal on Computing* 8.3 (August 1996), pp. 289–301.

[ALS14] Ali Al-Shabibi, Marc De Leenheer, and Bill Snow. "OpenVirteX: Make Your Virtual SDNs Programmable." In: *Proc. ACM Workshop on Hot Topics in Softw. Defined Netw.* Chicago, Illinois, USA: ACM, August 2014, pp. 25–30.

[APS+05] Thomas Anderson, Larry Peterson, Scott Shenker, and Jonathan Turner. "Overcoming the Internet impasse through virtualization." In: *IEEE Computer* 38.4 (April 2005), pp. 34–41.

[BAM10] Theophilus Benson, Aditya Akella, and David a. Maltz. "Network traffic characteristics of data centers in the wild." In: *Proc. ACM SIGCOMM IMC*. New York, New York, USA: ACM Press, November 2010, p. 267.

[BBE+13] Md Faizul Bari, Raouf Boutaba, Rafael Esteves, Lisandro Zambenedetti Granville, Maxim Podlesny, Md Golam Rabbani, Qi Zhang, and Mohamed Faten Zhani. "Data center network virtualization: A survey." In: *IEEE Commun. Surveys & Tutorials* 15.2 (2013), pp. 909–928.

[BBG+10] Andrea Bianco, Robert Birke, Luca Giraudo, Manuel Palacin, Dipartimento Elettronica, and Politecnico Torino. "OpenFlow Switching : Data Plane Performance." In: *Proc. IEEE ICC*. May 2010, pp. 1–5.

[BCK+11] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. "Towards predictable datacenter networks." In: *Proc. ACM SIGCOMM*. Toronto, Ontario, Canada: ACM, August 2011, pp. 242–253.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. “Xen and the art of virtualization.” In: *Proc. ACM Symp. on Operating Systems.* Bolton Landing, NY, USA: ACM, 2003, pp. 164–177.

[BDW01] Oded Berman, Zvi Drezner, and George O. Wesolowsky. “Location of facilities on a network with groups of demand points.” In: *IIE Transactions (Institute of Industrial Engineers)* 33.8 (2001), pp. 637–648.

[BGH+14] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, William Snow, Guru Parulkar, Brian O’Connor, and Pavlin Radoslavov. “ONOS: Towards an Open, Distributed SDN OS.” In: *Proc. ACM Workshop on Hot Topics in Softw. Defined Netw.* Chicago, Illinois, USA, August 2014, pp. 1–6.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics).* Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[BKER+12] Michele Berlingerio, Danai Koutra, Tina Eliassi-Rad, and Christos Faloutsos. “NetSimile: A Scalable Approach to Size-Independent Network Similarity.” In: *Computer Research Repository (CoRR)* (September 2012), pp. 1–8. arXiv: 1209.2684.

[BP12] Zdravko Bozakov and Panagiotis Papadimitriou. “AutoSlice: Automated and Scalable Slicing for Software-defined Networks.” In: *Proc. ACM CoNEXT Student Workshop.* Nice, France: ACM, December 2012, pp. 3–4.

[BP14] Zdravko Bozakov and Panagiotis Papadimitriou. “Towards a scalable software-defined network virtualization platform.” In: *Proc. IEEE/IFIP NOMS.* Krakow, Poland, May 2014, pp. 1–8.

[BPL+16] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. “Neural Combinatorial Optimization with Reinforcement Learning.” In: *Computer Research Repository (CoRR)* (November 2016). arXiv: 1611.09940.

[BR99] Albert-László Barabási and Albert Réka. “Emergence of Scaling in Random Networks.” In: *Science* 286.5439 (October 1999), pp. 509–512.

[BRC+13] Md. Faizul Bari, Arup Raton Roy, Shihabur Rahman Chowdhury, Qi Zhang, Mohamed Faten Zhani, Reaz Ahmed, and Raouf Boutaba. “Dynamic Controller Provisioning in Software Defined Networks.” In: *Proc. IFIP/IEEE CNSM.* October 2013, pp. 18–25.

[Bre01] Leo Breiman. “Random Forests.” In: *Machine Learning* 45.1 (2001), pp. 5–32.

[BVS+17] Marcel Blöcher, Malte Viering, Stefan Schmid, and Patrick Eugster. “The Grand CRU Challenge.” In: *Proc. Workshop on Hot Topics in Container Networking and Networked Systems (HotConNet ’17).* New York, New York, USA: ACM Press, 2017, pp. 7–11.

[CB08] N. M. Mosharaf Kabir Chowdhury and Raouf Boutaba. “A survey of network virtualization.” In: *Computer Networks* 54 (2008), pp. 862–876.

[CB09] N. M. Mosharaf Kabir Chowdhury and Raouf Boutaba. “Network virtualization: state of the art and research challenges.” In: *IEEE Commun. Mag.* 47.7 (July 2009), pp. 20–26.

[CFG14] Martin Casado, Nate Foster, and Arjun Guha. "Abstractions for software-defined networks." In: *Communications of the ACM* 57.10 (September 2014), pp. 86–95.

[CGR+12] Roberto Doriguzzi Corin, Matteo Gerola, Roberto Riggio, Francesco De Pellegrini, and Elio Salvadori. "VeRTIGO: Network virtualization and beyond." In: *Proc. IEEE Eu. Workshop on Software Defined Netw.* October 2012, pp. 24–29.

[CH08] Vira Chankong and Yacov Y Haimes. *Multiobjective decision making: theory and methodology.* Courier Dover Publications, 2008.

[CKR+10] Martín Casado, Teemu Koponen, Rajiv Ramanathan, and Scott Shenker. "Virtualizing the network forwarding plane." In: *Proc. ACM Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO).* Philadelphia, Pennsylvania: ACM, November 2010, 8:1–8:6.

[CMG+14] Stuart Clayman, Elisa Maini, Alex Galis, Antonio Manzalini, and Nicola Mazzocca. "The dynamic placement of virtual network functions." In: *Proc. IEEE/IFIP NOMS.* May 2014, pp. 1–9.

[CMT+11] Andrew R Curtis, Jeffrey C Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. "DevoFlow: Scaling Flow Management for High-Performance Networks." In: *ACM SIGCOMM Computer Communication Review* 41.4 (October 2011), p. 254.

[Coh13] Jared L Cohon. *Multiobjective programming and planning.* Courier Corporation, 2013.

[Cor02] R. J. Corsini. *The Dictionary of Psychology.* Psychology Press, 2002.

[CRB09] Mosharaf Chowdhury, Muntasir Raihan Rahman, and Raouf Boutaba. "Virtual Network Embedding with Coordinated Node and Link Mapping." In: *Proc. IEEE INFOCOM.* April 2009, pp. 783–791.

[CRB12] Mosharaf Chowdhury, Muntasir Raihan Rahman, and Raouf Boutaba. "ViNEYard: Virtual Network Embedding Algorithms With Coordinated Node and Link Mapping." In: *IEEE/ACM Trans. Netw.* 20.1 (February 2012), pp. 206–219.

[CSB10] Mosharaf Chowdhury, Fady Samuel, and Raouf Boutaba. "PolyViNE." In: *Proc. ACM SIGCOMM Workshop on Virtualized Infrast. Systems and Arch. (VISA).* New Delhi, India: ACM, 2010, pp. 49–56.

[CSO+09] Ítalo Cunha, Fernando Silveira, Ricardo Oliveira, Renata Teixeira, and Christophe Diot. "Uncovering Artifacts of Flow Measurement Tools." In: *Passive and Active Network Measurement: 10th International Conference, PAM 2009, Seoul, Korea, April 1-3, 2009. Proceedings.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 187–196.

[CSZ+11] Xiang Cheng, Sen Su, Zhongbao Zhang, Hanchi Wang, Fangchun Yang, Yan Luo, and Jie Wang. "Virtual network embedding through topology-aware node ranking." In: *ACM SIGCOMM Computer Commun. Rev.* 41.2 (April 2011), p. 38.

[DAR12] Mihai Dobrescu, Katerina Argyraki, and Sylvia Ratnasamy. "Toward Predictable Performance in Software Packet-Processing Platforms." In: *Proc. USENIX Symp. NSDI.* San Jose, CA: USENIX Association, April 2012, pp. 11–11.

[DJS12] Alisa Devlic, Wolfgang John, and P Sköldström. *Carrier-grade Network Management Extensions to the SDN Framework*. Tech. rep. Erricson Research, 2012.

[DK13] Fred Douglis and Orran Krieger. “Virtualization.” In: *IEEE Internet Computing* 17.2 (2013), pp. 6–9.

[DK16] Christina Delimitrou and Christos Kozyrakis. “HCloud: Resource-Efficient Provisioning in Shared Cloud Systems.” In: *Proc. ASPLOS*. New York, New York, USA: ACM Press, 2016, pp. 473–488.

[DKR13] Dmitry Drutskoy, Eric Keller, and Jennifer Rexford. “Scalable Network Virtualization in Software-Defined Networks.” In: *IEEE Internet Computing* 17.2 (2013), pp. 20–27.

[DSG+14] Roberto Doriguzzi Corin, Elio Salvadori, Matteo Gerola, Hagen Woesner, and Marc Sune. “A Datapath-centric Virtualization Mechanism for OpenFlow Networks.” In: *Proc. IEEE Eu. Workshop on Software Defined Netw.* September 2014, pp. 19–24.

[DW00] Zvi Drezner and George O. Wesolowsky. “Location Models With Groups Of Demand Points.” In: *INFOR: Information Systems and Operational Research* 38.4 (2000), pp. 359–372.

[EABL+11] Mohammed El-Azzab, Imen Limam Bedhiaf, Yves Lemieux, and Omar Cherkaoui. “Slices Isolator for a Virtualized Openflow Node.” In: *Proc. Int. Symp. on Netw. Cloud Computing and Appl. (NCCA)*. November 2011, pp. 121–126.

[EH08] Jeremy Elson and Jon Howell. “Handling Flash Crowds from Your Garage.” In: *Proc. USENIX Annual Technical Conference*. June 2008, pp. 171–184.

[ER13] Jeffrey Erman and K.K. Ramakrishnan. “Understanding the super-sized traffic of the super bowl.” In: *Proc. ACM SIGCOMM IMC*. Barcelona, Spain: ACM, 2013, pp. 353–360.

[ER59] Paul Erdős and Alfréd. Rényi. “On random graphs.” In: *Publicationes Mathematicae* 6 (1959), pp. 290–297.

[Eri13] David Erickson. “The beacon openflow controller.” In: *Proc. ACM Workshop on Hot Topics in Softw. Defined Netw.* Hong Kong, China: ACM, August 2013, pp. 13–18.

[FBB+13] Andreas Fischer, Juan Felipe Botero, Michael Till Beck, Hermann de Meer, and Xavier Hesselbach. “Virtual Network Embedding: A Survey.” In: *IEEE Commun. Surveys & Tutorials* VLiM (2013), pp. 1–19.

[FGR07] Nick Feamster, Lixin Gao, and Jennifer Rexford. “How to lease the internet in your spare time.” In: *ACM SIGCOMM Computer Commun. Rev.* 37.1 (January 2007), p. 61.

[FHF+14] Reza Zanjirani Farahani, Masoud Hekmatfar, Behnam Fahimnia, and Narges Kazemzadeh. “Hierarchical facility location problem: Models, classifications, techniques, and applications.” In: *Comp. Ind. Eng.* 68.1 (2014), pp. 104–117.

[FKL04] Fang Yu, R.H. Katz, and T.V. Lakshman. “Gigabit rate packet pattern-matching using TCAM.” In: *Proc. IEEE ICNP*. October 2004, pp. 174–183.

[FLH+00] Dino Farinacci, Tony Li, Stan Hanks, David Meyer, and Paul Traina. *Generic Routing Encapsulation (GRE)*. RFC 2784. http://www.rfc-editor.org/rfc/rfc2784.txt. RFC Editor, March 2000.

[FLW+14] Min Feng, Jianxin Liao, Jingyu Wang, Sude Qing, and Qi Qi. "Topology-aware Virtual Network Embedding based on multiple characteristics." In: *Proc. IEEE ICC*. June 2014, pp. 2956–2962.

[FRZ14] Nick Feamster, Jennifer Rexford, and Ellen Zegura. "The Road to SDN: An Intellectual History of Programmable Networks." In: *ACM SIGCOMM Computer Commun. Rev.* 44.2 (April 2014), pp. 87–98.

[FS97] Yoav Freund and Robert E Schapire. "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting." In: *Journal of Computer and System Sciences* 55.1 (August 1997), pp. 119–139.

[Gao14] Jim Gao. "Machine learning applications for data center optimization." In: *Google White Paper* (2014).

[GDFM+12] Jose Luis Garcia-Dorado, Alessandro Finamore, Marco Mellia, Michela Meo, and Maurizio M. Munafo. "Characterization of ISP Traffic: Trends, User Habits, and Access Technology Impact." In: *IEEE Trans. on Netw. and Serv. Manage.* 9.2 (June 2012), pp. 142–155.

[GEW06] Pierre Geurts, Damien Ernst, and Louis Wehenkel. "Extremely randomized trees." In: *Machine Learning* 63.1 (April 2006), pp. 3–42.

[GFM+12] Vinicius Gehlen, Alessandro Finamore, Marco Mellia, and Maurizio M. Munafò. "Uncovering the Big Players of the Web." In: *Traffic Monitoring and Analysis: 4th International Workshop, TMA 2012, Vienna, Austria, March 12, 2012. Proceedings.* Ed. by Antonio Pescapè, Luca Salgarelli, and Xenofontas Dimitropoulos. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 15–28.

[GFW03] Thomas Gärtner, Peter Flach, and Stefan Wrobel. "On graph kernels: Hardness results and efficient alternatives." In: *Learning Theory and Kernel Machines*. Springer, 2003, pp. 129–143.

[GHJ+09] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David a. Maltz, Parveen Patel, and Sudipta Sengupta. "Vl2: A Scalable and Flexible Data Center Network." In: *ACM SIGCOMM Computer Commun. Rev.* 39.4 (2009), p. 51.

[GHM+08] Albert Greenberg, James Hamilton, David A. Maltz, and Parveen Patel. "The Cost of a Cloud: Research Problems in Data Center Networks." In: *ACM SIGCOMM Computer Commun. Rev.* 39.1 (December 2008), p. 68.

[GJN11] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. "Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications." In: *Proc. ACM SIGCOMM*. Toronto, Ontario, Canada: ACM, August 2011, pp. 350–361.

[GKP+08] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. "NOX: towards an operating system for networks." In: *ACM SIGCOMM Computer Commun. Rev.* 38.3 (2008), pp. 105–110.

[GKS+15] Milad Ghaznavi, Aimal Khan, Nashid Shahriar, Khalid Alsubhi, Reaz Ahmed, and Raouf Boutaba. “Elastic Virtual Network Function Placement.” In: *Proc. IEEE CloudNet.* October 2015, pp. 255–260.

[GLL+09] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. “BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers.” In: *ACM SIGCOMM Computer Commun. Rev.* 39.4 (October 2009), pp. 63–74.

[GMK+16] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. “Evolve or Die: High-Availability Design Principles Drawn from Googles Network Infrastructure.” In: *Proc. ACM SIGCOMM.* New York, New York, USA: ACM Press, 2016, pp. 58–72. arXiv: arXiv:1011.1669v3.

[GMM00] Sudipto Guha, A. Meyerson, and K. Munagala. “Hierarchical placement and network design problems.” In: *IEEE FOCS.* November 2000, pp. 603–612.

[GO15] Inc. Gurobi Optimization. *Gurobi Optimizer Reference Manual.* 2015.

[GO16] Inc. Gurobi Optimization. *Gurobi Optimizer Reference Manual.* 2016.

[Gol74] Robert P Goldberg. “Survey of virtual machine research.” In: *IEEE Computer* 7.6 (1974), pp. 34–45.

[Gra] *Graph isomorphism.* URL: https://en.wikipedia.org/wiki/Graph_isomorphism.

[GS89] Lars Gislén, Carsten, and Bo Söderberg. “"Teachers and Classes" with Neural Networks.” In: *International Journal of Neural Systems* 01.02 (January 1989), pp. 167–176.

[GW15] Pankaj Garg and Yu-Shun Wang. *NVGRE: Network Virtualization Using Generic Routing Encapsulation.* RFC 7637. RFC Editor, September 2015.

[GWZ+14] Long Gong, Yonggang Wen, Zuqing Zhu, and Tony Lee. “Toward profit-seeking virtual network embedding algorithm via global resource capacity.” In: *Proc. IEEE INFOCOM.* April 2014, pp. 1–9.

[HFW+13] Keqiang He, Alexis Fisher, Liang Wang, Aaron Gember, Aditya Akella, and Thomas Ristenpart. “Next stop, the cloud: Understanding modern web service deployment in ec2 and azure.” In: *Proc. ACM SIGCOMM IMC.* ACM. 2013, pp. 177–190.

[HG13] Shufeng Huang and James Griffioen. “Network Hypervisors: managing the emerging SDN Chaos.” In: *Proc. IEEE ICCCN.* July 2013, pp. 1–7.

[HHLB10] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. “Automated Configuration of Mixed Integer Programming Solvers.” In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems.* Vol. 6140 LNCS. 2010, pp. 186–202.

[HK92] Benjamin J. Hellstrom and Laveen N. Kanal. “Knapsack Packing Networks.” In: *IEEE Trans. Neural Netw.* 3.2 (March 1992), pp. 302–307.

[Hop84] John J. Hopfield. “Neurons with graded response have collective computational properties like those of two-state neurons.” In: *Proc. Natl. Acad. Sci. USA* 81.10 (May 1984), pp. 3088–3092.

[HSM12] Brandon Heller, Rob Sherwood, and Nick McKeown. “The controller placement problem.” In: *Proc. ACM Workshop on Hot Topics in Softw. Defined Netw.* Helsinki, Finland: ACM, August 2012, pp. 7–12.

[HSS+16] Luuk Hendriks, Ricardo De O Schmidt, Ramin Sadre, Jeronimo A Bezerra, and Aiko Pras. “Assessing the Quality of Flow Measurements from OpenFlow Devices.” In: *Proc. International Workshop on Traffic Monitoring and Analysis (TMA).* 2016, pp. 1–8.

[HT16] Soroush Haeri and Ljiljana Trajković. “Virtual Network Embeddings in Data Center Networks.” In: *IEEE International Symposium on Circuits and Systems.* May 2016, pp. 874–877.

[HT85] John J. Hopfield and David W. Tank. “"Neural" computations of decisions in optimization problems.” In: *Biological Cybernetics* 52 (1985), pp. 141–152.

[Hut14] Frank Hutter. *Machine Learning for Optimization: Automated Parameter Tuning and Beyond.* 2014.

[HWG+14] Yannan Hu, Wendong Wang, Xiangyang Gong, Xirong Que, and Shiduan Cheng. “On reliability-optimized controller placement for Software-Defined Networks.” In: *China Communications* 11.2 (February 2014), pp. 38–54.

[ICM+02] Gianluca Iannaccone, Chen-nee Chuah, Richard Mortier, Supratik Bhattacharyya, and Christophe Diot. “Analysis of link failures in an IP backbone.” In: *Proc. ACM SIGCOMM Workshop on Internet Measurment (IMW).* Marseille, France: ACM, November 2002, pp. 237–242.

[JCM+16] Justin Bayer, Christian Osendorfer, Max Karl, Maximilian Soelch, and Sebastian Urban. *breze.* TUM, 2016.

[JCPG14] Yury Jimenez, Cristina Cervelló-Pastor, and Aurelio J. García. “On the controller placement for designing a distributed SDN control layer.” In: *Proc. IFIP Networking.* June 2014, pp. 1–9.

[JCS+15] Justin Bayer, Christian Osendorfer, Sarah Diot-Girard, Thomas Rueckstiess, and Sebastian Urban. *climin - A pythonic framework for gradient-based function optimization.* Tech. rep. TUM, 2015.

[JGR+15] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. “CoVisor: A Compositional Hypervisor for Software-Defined Networks.” In: *Proc. USENIX Symp. NSDI.* Oakland, CA: USENIX Association, May 2015, pp. 87–101.

[JLM+12] Michael Jarschel, Frank Lehrieder, Zsolt Magyari, and Rastin Pries. “A flexible OpenFlow-controller benchmark.” In: *Proc. IEEE Eu. Workshop on Software Defined Netw.* October 2012, pp. 48–53.

[JMZ+14] Michael Jarschel, Christopher Metter, Thomas Zinner, Steffen Gebert, and Phuoc Tran-Gia. “OFCProbe: A platform-independent tool for OpenFlow controller analysis.” In: *Proc. IEEE Int. Conf. on Commun. and Electronics (ICCE).* July 2014, pp. 182–187.

[JOS+11] Michael Jarschel, Simon Oechsner, Daniel Schlosser, Rastin Pries, and Sebastian Goll. “Modeling and performance evaluation of an OpenFlow architecture.” In: *Proc. ITC*. September 2011, pp. 1–7.

[JP13] Raj Jain and Subharthi Paul. “Network virtualization and software defined networking for cloud computing: a survey.” In: *IEEE Commun. Mag.* 51.11 (November 2013), pp. 24–31.

[JS15] Brendan Jennings and Rolf Stadler. “Resource Management in Clouds: Survey and Research Challenges.” In: *Journal of Network and Systems Management* 23.3 (July 2015), pp. 567–619.

[KAB+14] Teemu Koponen et al. “Network Virtualization in Multi-tenant Datacenters.” In: *Proc. USENIX Symp. NSDI.* Seattle, WA: USENIX Association, April 2014, pp. 203–216.

[KAG+12] Eric Keller, Dushyant Arora, Soudeh Ghorbani, Matt Caesar, and Jennifer Rexford. “Live Migration of an Entire Network (and its Hosts).” In: *Princeton University Computer Science Technical Report* (2012), pp. 109–114.

[KB14] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization.” In: *CoRR* abs/1412.6980 (2014). arXiv: 1412.6980.

[KBS+16] Elias B Khalil, Pierre Le Bodic, Le Song, George Nemhauser, and Bistra Dilkina. “Learning to Branch in Mixed Integer Programming.” In: *Proc. AAAI.* Phoenix, Arizona: AAAI Press, 2016, pp. 724–731.

[KCG+10] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. “Onix: A Distributed Control Platform for Large-scale Production Networks.” In: *Proc. USENIX Conf. OSDI.* Vancouver, BC, Canada: USENIX Association, October 2010, pp. 351–364.

[KCI+15] T. Koponen, M. Casado, P.S. Ingram, W.A. Lambeth, P.J. Balland, K.E. Amidon, and D.J. Wendlandt. *Network virtualization, US Patent 8,959,215.* February 2015.

[KD05] Andreas Klose and Andreas Drexl. “Facility location models for distribution system design.” In: *European J. Operational Res.* 162.1 (2005), pp. 4–29.

[KNF+11] Simon Knight, Hung X. Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. “The Internet Topology Zoo.” In: *IEEE Journal on Selected Areas in Commun.* 29.9 (October 2011), pp. 1765–1775.

[Kog11] Jochen Kogel. “One-way delay measurement based on flow data: Quantification and compensation of errors by exporter profiling.” In: *Proc. ICOIN.* IEEE, January 2011, pp. 25–30.

[KOO+16] Kota Kawashima, Tatsuya Otoshi, Yuichi Ohsita, and Masayuki Murata. “Dynamic Placement of Virtual Network Functions Based on Model Predictive Control.” In: *Proc. IEEE/IFIP NOMS.* April 2016, pp. 1037–1042.

[KPK14] Maciej Kuzniar, Peter Peresini, and Dejan Kostic. *What you need to know about SDN control and data planes.* Tech. rep. EPFL, TR 199497, 2014.

[KPK15] Maciej Kuźniar, Peter Perešíni, and Dejan Kostić. "What You Need to Know About SDN Flow Tables." In: *Passive and Active Measurement: 16th International Conference, PAM 2015, New York, NY, USA, March 19-20, 2015, Proceedings.* Ed. by Jelena Mirkovic and Yong Liu. Cham: Springer International Publishing, 2015, pp. 347–359.

[KRV+15] Diego Kreutz, Fernando MV Ramos, PE Verissimo, C Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. "Software-defined networking: A comprehensive survey." In: *Proc. IEEE* 103.1 (2015), pp. 14–76.

[KSG+09] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. "The nature of data center traffic." In: *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference - IMC '09.* Microsoft. New York, New York, USA: ACM Press, 2009, p. 202.

[KV07] Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms.* 4th. Springer Publishing Company, Incorporated, 2007.

[KZM+17] Patrick Kalmbach, Johannes Zerwas, Michael Manhart, Andreas Blenk, Stefan Schmid, and Wolfgang Kellerer. *Data on "o'zapft is: Tap Your Network Algorithm's Big Data!"* TUM. 2017. URL: https://mediatum.ub.tum.de/1361589.

[LAL+14] Ruoqian Liu, Ankit Agrawal, Wei-keng Liao, and Alok Choudhary. "Search Space Preprocessing in Solving Complex Optimization Problems." In: *Workshop on Complexity for Big Data.* 2014.

[Lar] *Linux Advanced Routing & Traffic Control.* URL: http://lartc.org/.

[LBB+15] Marcelo Caggiani Luizelli, Leonardo Richter Bays, Luciana Salete Buriol, Marinho Pilla Barcellos, and Luciano Paschoal Gaspary. "Piecing together the NFV provisioning puzzle: Efficient placement and chaining of virtual network functions." In: *Proc. IFIP/IEEE IM.* May 2015, pp. 98–106.

[LC16] Philipp Leitner and Jürgen Cito. "Patterns in the Chaos—A Study of Performance Variation and Predictability in Public IaaS Clouds." In: *ACM Transactions on Internet Technology* 16.3 (April 2016), pp. 1–23.

[LGS+15] Stanislav Lange, Steffen Gebert, Joachim Spoerhase, Piotr Rygielski, Thomas Zinner, Samuel Kounev, and Phuoc Tran-Gia. "Specialized Heuristics for the Controller Placement Problem in Large Scale SDN Networks." In: *Proc. ITC.* September 2015, pp. 210–218.

[LGZ+15] Stanislav Lange, Steffen Gebert, Thomas Zinner, Phuoc Tran-Gia, David Hock, Michael Jarschel, and Marco Hoffmann. "Heuristic Approaches to the Controller Placement Problem in Large Scale SDN Networks." In: *IEEE Trans. on Netw. and Serv. Manage.* 12.1 (March 2015), pp. 4–17.

[LLJ10] Yunfa Li, Wanqing Li, and Congfeng Jiang. "A survey of virtual machine system: Current technology and future trends." In: *Proc. IEEE Int. Symp. on Electronic Commerce and Security (ISECS).* July 2010, pp. 332–336.

[LLN14] Lingxia Liao, Victor C M Leung, and Panos Nasiopoulos. "DFVisor: Scalable Network Virtualization for QoS Management in Cloud Computing." In: *Proc. CNSM and Workshop IFIP*. November 2014, pp. 328–331.

[LMC+13] Lei Liu, Raül Muñoz, Ramon Casellas, Takehiro Tsuritani, Ricardo Martínez, and Itsuro Morita. "OpenSlice: an OpenFlow-based control plane for spectrum sliced elastic optical path networks." In: *OSA Opt. Express* 21.4 (2013), pp. 4194–4204.

[Lox] *OpenFlowJ Loxi*. URL: https://github.com/floodlight/loxigen/wiki/OpenFlowJ-Loxi.

[LSL15] Lingxia Liao, Abdallah Shami, and V. C. M. Leung. "Distributed FlowVisor: a distributed FlowVisor platform for quality of service aware cloud network virtualisation." In: *IET Networks* 4.5 (2015), pp. 270–277.

[LSY+12] Geng Li, Murat Semerci, Bülent Yener, and Mohammed J Zaki. "Effective graph classification based on topological and label attributes." In: *Statistical Analysis and Data Mining* 5.4 (August 2012), pp. 265–283.

[LTH+14] Aggelos Lazaris, Daniel Tahara, Xin Huang, Erran Li, Andreas Voellmy, Y Richard Yang, and Minlan Yu. "Tango: Simplifying SDN Control with Automatic Switch Property Inference, Abstraction, and Optimization." In: *Proc. ACM CoNEXT*. Sydney, Australia: ACM, December 2014, pp. 199–212.

[LTM+11] Lei Liu, Takehiro Tsuritani, Itsuro Morita, Hongxiang Guo, and Jian Wu. "Experimental validation and performance evaluation of OpenFlow-based wavelength path control in transparent optical networks." In: *OSA Opt. Express* 19.27 (2011), pp. 26578–26593.

[MAB+08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. "OpenFlow: enabling innovation in campus networks." In: *ACM SIGCOMM Computer Commun. Rev.* 38.2 (2008), pp. 69–74.

[MAM+16] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. "Resource Management with Deep Reinforcement Learning." In: *Proc. ACM Workshop HotNets*. Atlanta, GA, USA: ACM, November 2016, pp. 50–56.

[MB02] Maria Carolina Monard and Gustavo E A P A Batista. "Learning with skewed class distributions." In: *Advances in Logic, Artificial Intelligence and Robotics* (2002), pp. 173 –180.

[MDD+14] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. *Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks*. RFC 7348. http://www.rfc-editor.org/rfc/rfc7348.txt. RFC Editor, August 2014.

[MJS+11] Jon Matias, Eduardo Jacob, David Sanchez, and Yuri Demchenko. "An OpenFlow based network virtualization framework for the Cloud." In: *Proc. IEEE CloudCom*. November 2011, pp. 672–678.

[MKL+12] Seokhong Min, S Kim, Jaeyong Lee, and Byungchul Kim. “Implementation of an OpenFlow network virtualization for multi-controller environment.” In: *Proc. Int. Conf. on Advanced Commun. Techn. (ICACT).* February 2012, pp. 589–592.

[MMB00] Alberto Medina, Ibrahim Matta, and John Byers. “On the Origin of Power Laws in Internet Topologies.” In: *ACM SIGCOMM Computer Commun. Rev.* 30.2 (April 2000), pp. 18–28.

[MOL+14] Lucas F. Muller, Rodrigo R Oliveira, Marcelo C Luizelli, Luciano P Gaspary, and Marinho P Barcellos. “Survivor: An enhanced controller placement strategy for improving SDN survivability.” In: *Proc. IEEE Globecom.* December 2014, pp. 1909–1915.

[MRF+13] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. “Composing Software-Defined Networks.” In: *Proc. USENIX Symp. NSDI.* Lombard, IL: USENIX Association, April 2013, pp. 1–13.

[MSK+13] Marcio Melo, Susana Sargento, Ulrich Killat, Andreas Timm-Giel, and Jorge Carapinha. “Optimal Virtual Network Embedding: Node-Link Formulation.” In: *IEEE Trans. on Netw. and Serv. Manage.* 10.4 (December 2013), pp. 356–368.

[New10] Mark Newman. *Networks: An Introduction.* Oxford University Press, March 2010. eprint: 1212.2425.

[NVN+13] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. “Deepdive: Transparently identifying and managing performance interference in virtualized environments.” In: *Proc. USENIX Annual Technical Conference.* San Jose, CA: USENIX Association, June 2013, pp. 219–230.

[Oft] *OFTest—Validating OpenFlow Switches.* URL: http://www.projectfloodlight.org/oftest/.

[Ope13] OpenDaylight. *A Linux Foundation Collaborative Project.* 2013.

[OPS93] Mattias Ohlsson, Carsten Peterson, and Bo Söderberg. “Neural Networks for Optimization Problems with Inequality Constraints: The Knapsack Problem.” In: *Neural Computation* 5.2 (March 1993), pp. 331–339.

[Os3] *Internet2 Open Science, Scholarship, and Services Exchange (OS3E).* URL: http::/www.internet2.edu/network/ose.

[OWP+10] S. Orlowski, R. Wessäly, M. Pióro, and A. Tomaszewski. “SNDlib 1.0—Survivable Network Design Library.” In: *Networks* 55.3 (2010), pp. 276–286.

[PJ98] Hasan Pirkul and Vaidyanathan Jayaraman. “A multi-commodity, multi-plant, capacitated facility location problem: formulation and efficient heuristic solution.” In: *Comp. & Op. Res.* 25.10 (1998), pp. 869–878.

[PJG12] Rastin Pries, Michael Jarschel, and Sebastian Goll. “On the usability of OpenFlow in data center environments.” In: *Proc. IEEE ICC.* IEEE, June 2012, pp. 5533–5537.

[PPA+09] Ben Pfaff, Justin Pettit, Keith Amidon, Martin Casado, Teemu Koponen, and Scott Shenker. “Extending Networking into the Virtualization Layer.” In: *HotNets-VIII.* October 2009, pp. 1–6.

[PS02] Rina Panigrahy and Samar Sharma. “Reducing TCAM power consumption and increasing throughput.” In: *Proc. IEEE High Perf. Interconnects.* August 2002, pp. 107–112.

[PS06] Kostas Pagiamtzis and Ali Sheikholeslami. “Content-addressable memory (CAM) circuits and architectures: A tutorial and survey.” In: *IEEE J. Solid-State Circuits* 41.3 (2006), pp. 712–727.

[PZH+11] Abhinav Pathak, Ming Zhang, Y. Charlie Hu, Ratul Mahajan, and Dave Maltz. “Latency inflation with MPLS-based traffic engineering.” In: *Proc. ACM SIGCOMM IMC.* Berlin, Germany: ACM, November 2011, pp. 463–472.

[Qui86] J.R. Quinlan. “Induction of Decision Trees.” In: *Machine Learning* 1.1 (1986), pp. 81–106.

[RAB+14] Charalampos Rotsos, Gianni Antichi, Marc Bruyere, Philippe Owezarski, and Andrew W Moore. “An open testing framework for next-generation OpenFlow switches.” In: *Proc. IEEE Eu. Workshop on Software Defined Netw.* September 2014, pp. 127–128.

[RB13] M. R. Rahman and R. Boutaba. “SVNE: Survivable Virtual Network Embedding Algorithms for Network Virtualization.” In: *IEEE Trans. on Netw. and Serv. Manage.* 10.2 (June 2013), pp. 105–118.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. “Learning representations by back-propagating errors.” In: *Nature* 323.6088 (October 1986), pp. 533–536.

[RNM+14] Navaneeth Rameshan, Leandro Navarro, Enric Monte, and Vladimir Vlassov. “Stay-Away , protecting sensitive applications from performance interference.” In: *Proc. International Middleware Conference - Middleware ’14.* December. New York, New York, USA: ACM Press, December 2014, pp. 301–312.

[RS18] Matthias Rost and Stefan Schmid. “NP-Completeness and Inapproximability of the Virtual Network Embedding Problem and Its Variants.” In: *CoRR* abs/1801.03162 (2018). arXiv: 1801.03162.

[RSU+12] Charalampos Rotsos, Nadi Sarrar, Steve Uhlig, Rob Sherwood, and Andrew W. Moore. “OFLOPS: An Open Framework for OpenFlow Switch Evaluation.” In: *Passive and Active Measurement: 13th International Conference, PAM 2012, Vienna, Austria, March 12-14th, 2012. Proceedings.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 85–95.

[RTP15] M. Roughan, J. Tuke, and E. Parsonage. “Estimating the Parameters of the Waxman Random Graph.” In: *Computer Research Repository (CoRR)* (June 2015).

[Run95] Carl Runge. “Ueber die numerische Auflösung von Differentialgleichungen.” In: *Mathematische Annalen* 46 (1895), pp. 167 –178.

[RVC01] E. Rosen, A. Viswanathan, and R. Callon. *Multiprotocol Label Switching Architecture.* RFC 3031. http://www.rfc-editor.org/rfc/rfc3031.txt. RFC Editor, January 2001.

[RW88] H.E. Rauch and T. Winarske. “Neural networks for routing communication traffic.” In: *IEEE Control Systems Magazine* 8.2 (April 1988), pp. 26–31.

[Ryu] *RYU SDN Framework, http://osrg.github.io/ryu.*

[SBD+18] Susanna Schwarzmann, Andreas Blenk, Ognjen Dobrijevic, Michael Jarschel, Andreas Hotho, Thomas Zinner, and Florian Wamser. "Big-Data Helps SDN to Improve Application Specific Quality of Service." In: *Big Data and Software Defined Networks.* Ed. by Javid Taheri. The Institution of Engineering and Technology (IET), January 2018, pp. 433–455.

[SC08] Ingo Steinwart and Andreas Christmann. *Support Vector Machines.* 1st. Springer Publishing Company, Incorporated, 2008.

[SCB+11] E. Salvadori, R. D. Corin, A. Broglio, and M. Gerola. "Generalizing Virtual Network Topologies in OpenFlow-Based Networks." In: *Proc. IEEE Globecom.* December 2011, pp. 1–6.

[SCS+15] Lalith Suresh, Marco Canini, Stefan Schmid, Anja Feldmann, and Telekom Innovation Labs. "C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection." In: *Proc. USENIX Symp. NSDI.* Oakland, CA: USENIX Association, May 2015, pp. 513–527.

[SDQR10] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. "Runtime measurements in the cloud." In: *Proc. VLDB Endow.* 3.1-2 (September 2010), pp. 460–471.

[SGN+12] Balazs Sonkoly, Andras Gulyas, Felician Nemeth, Janos Czentye, Krisztian Kurucz, Barnabas Novak, and Gabor Vaszkun. "OpenFlow Virtualization Framework with Advanced Capabilities." In: *Proc. IEEE Eu. Workshop on Software Defined Netw.* October 2012, pp. 18–23.

[SGP14] Christoph Salge, Cornelius Glackin, and Daniel Polani. "Changing the environment based on empowerment as intrinsic motivation." In: *Entropy* 16.5 (2014), pp. 2789–2819. arXiv: 1406.1767.

[SGY+09] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick Mckeown, and Guru Parulkar. *FlowVisor: A Network Virtualization Layer.* Tech. rep. OpenFlow Consortium, 2009, p. 15.

[SJ13] Pontus Skoldstrom and Wolfgang John. "Implementation and Evaluation of a Carrier-Grade OpenFlow Virtualization Scheme." In: *Proc. IEEE Eu. Workshop on Software Defined Netw.* October 2013, pp. 75–80.

[SKP96] Kate Smith, M. Krishnamoorthy, and M. Palaniswami. "Neural versus traditional approaches to the location of interacting hub facilities." In: *Location Science* 4.3 (October 1996), pp. 155–171.

[SML10] Jyotiprakash Sahoo, Subasish Mohapatra, and Radha Lath. "Virtualization: A survey on concepts, taxonomy and associated security issues." In: *Proc. IEEE Int. Conf. on Computer and Netw. Techn. (ICCNT).* April 2010, pp. 222–226.

[SN05] James E Smith and Ravi Nair. "The architecture of virtual machines." In: *IEEE Computer* 38.5 (2005), pp. 32–38.

[SNS+10] Rob Sherwood et al. “Carving research slices out of your production networks with OpenFlow.” In: *ACM SIGCOMM Computer Commun. Rev.* 40.1 (January 2010), pp. 129–130.

[Sof09] Rute C Sofia. “A survey of advanced ethernet forwarding approaches.” In: *IEEE Commun. Surveys & Tutorials* 11.1 (2009), pp. 92–115.

[SSH15] Afrim Sallahi and Marc St-Hilaire. “Optimal Model for the Controller Placement Problem in Software Defined Networks.” In: *IEEE Communications Letters* 19.1 (January 2015), pp. 30–33.

[Sør48] T. Sørensen. “A method of establishing groups of equal amplitude in plant sociology based on similarity of species and its application to analyses of the vegetation on Danish commons.” In: *Biol. Skr.* 5 (1948), pp. 1–34.

[TCP91] G Tagliarini, J Christ, and E Page. “Optimization using neural networks.” In: *IEEE Trans. Comp.* 40.12 (December 1991), pp. 1347–1358.

[TG10] Amin Tootoonchian and Yashar Ganjali. “HyperFlow: A distributed control plane for OpenFlow.” In: *Proc. USENIX Internet Network Management Conf. on Research on Enterprise Netw.* San Jose, CA: USENIX Association, April 2010, pp. 3–3.

[TGG+12] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood. “On Controller Performance in Software-defined Networks.” In: *Proc. USENIX Wkshp. on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services.* Berkeley, CA, USA: USENIX Association, April 2012, pp. 10–10.

[THS10] Voravit Tanyingyong, Markus Hidell, and Peter Sjödin. “Improving PC-based OpenFlow switching performance.” In: *Proc. ACM/IEEE ANCS.* New York, New York, USA: ACM Press, 2010, p. 1.

[TMV+11] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. “The impact of memory subsystem resource sharing on datacenter applications.” In: *ACM SIGARCH Computer Architecture News* 39.3 (July 2011), p. 283.

[Tri17] A. Tripathi. *Machine Learning Cookbook.* Packt Publishing, Limited, 2017.

[TT05] J.S. Turner and D.E. Taylor. “Diversifying the Internet.” In: *Proc. IEEE Globecom.* Vol. 2. December 2005, 6 pp.–760.

[TTS+11] Brian Trammell, Bernhard Tellenbach, Dominik Schatzmann, and Martin Burkhart. “Peeling Away Timing Error in NetFlow Data.” In: ed. by Nina Taft and Fabio Ricciato. Vol. 7192. March. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 194–203.

[VGM+13] Ashish Vulimiri, Philip Brighten Godfrey, Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. “Low latency via redundancy.” In: *Proc. ACM CoNEXT.* New York, New York, USA: ACM Press, 2013, pp. 283–294.

[VRV14] Allan Vidal, Christian Esteve Rothenberg, and Fábio Luciano Verdi. “The libfluid OpenFlow driver implementation.” In: *Proc. 32nd Brazilian Symp. Comp. Netw.(SBRC).* 2014, pp. 1029–1036.

[Wax88] B.M. Waxman. “Routing of multipoint connections.” In: *IEEE Journal on Selected Areas in Commun.* 6.9 (1988), pp. 1617–1622. eprint: 49.12889 (10.1109).

[WB13] Keith Winstein and Hari Balakrishnan. “TCP ex machina.” In: *Proc. ACM SIGCOMM.* Hong Kong, China: ACM, August 2013, pp. 123–134.

[XHB+00] Xipeng Xiao, Alan Hannan, Brook Bailey, and Lionel M Ni. “Traffic Engineering with MPLS in the Internet.” In: *IEEE Network* 14.2 (2000), pp. 28–33.

[YBL+14] Guang Yao, Jun Bi, Yuliang Li, and Luyi Guo. “On the Capacitated Controller Placement Problem in Software Defined Networks.” In: *IEEE Communications Letters* 18.August (August 2014), pp. 1339–1342.

[YKI+14a] Hiroaki Yamanaka, Eiji Kawai, Shuji Ishii, Shinji Shimojo, and Communications Technology. “AutoVFlow: Autonomous Virtualization for Wide-area OpenFlow Networks.” In: *Proc. IEEE Eu. Workshop on Software Defined Netw.* September 2014, pp. 67–72.

[YKI+14b] Hiroaki Yamanaka, Eiji Kawai, Shuji Ishii, and Shinji Shimojo. “OpenFlow Network Virtualization on Multiple Administration Infrastructures.” In: *Proc. Open Networking Summit.* March 2014, pp. 1–2.

[YRF+10] Minlan Yu, Jennifer Rexford, Michael J. Freedman, and Jia Wang. “Scalable flow-based networking with DIFANE.” In: *ACM SIGCOMM Computer Commun. Rev.* 40.4 (October 2010), pp. 351–362.

[YRS15] Abdulsalam Yassine, Hesam Rahimi, and Shervin Shirmohammadi. “Software defined network traffic measurement: Current trends and challenges.” In: *IEEE Instrumentation & Measurement Magazine* 18.2 (April 2015), pp. 42–50.

[YYR+08] Minlan Yu, Yung Yi, Jennifer Rexford, and Mung Chiang. “Rethinking virtual network embedding.” In: *ACM SIGCOMM Computer Commun. Rev.* 38.2 (March 2008), p. 17.

[ZA06] Y. Zhu and M. Ammar. “Algorithms for Assigning Substrate Network Resources to Virtual Network Components.” In: *Proc. IEEE INFOCOM.* IEEE, April 2006, pp. 1–12.

[ZCB96] E.W. Zegura, K.L. Calvert, and S. Bhattacharjee. “How to model an internetwork.” In: *Proc. IEEE INFOCOM.* Vol. 2. IEEE Comput. Soc. Press, 1996, pp. 594–602.

[ZJX11] Jun Zhu, Zhefu Jiang, and Zhen Xiao. “Twinkle: A fast resource provisioning mechanism for internet services.” In: *Proc. IEEE INFOCOM.* IEEE, April 2011, pp. 802–810.

[ZLW+14] Zhaoning Zhang, Ziyang Li, Kui Wu, Dongsheng Li, Huiba Li, Yuxing Peng, and Xicheng Lu. “VMThunder: fast provisioning of large-scale virtual machine clusters.” In: *IEEE Trans. Parallel and Distr. Systems* 25.12 (December 2014), pp. 3328–3338.

[ZQW+12] Sheng Zhang, Zhuzhong Qian, Jie Wu, and Sanglu Lu. “An Opportunistic Resource Sharing and Topology-Aware mapping framework for virtual networks.” In: *Proc. IEEE INFOCOM.* i. IEEE, March 2012, pp. 2408–2416.

[ZSR+16] Eitan Zahavi, Alexander Shpiner, Ori Rottenstreich, Avinoam Kolodny, and Isaac Keslassy. “Links as a Service (LaaS).” In: *Proc. ANCS.* Santa Clara, California, USA: ACM, March 2016, pp. 87–98.

[Ope09] Open Networking Foundation (ONF). *OpenFlow Switch Specifications 1.0 (ONF TS-001).* https://www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.0.0.pdf. December 2009.

[Ope11a] Open Networking Foundation (ONF). *OpenFlow Switch Specifications 1.1.0 (ONF TS-002).* https://3vf60mmveq1g8vzn48q2o71a-wpengine.netdna-ssl.com/wp-content/uploads/2014/10/openflow-spec-v1.1.0.pdf. February 2011.

[Ope11b] Open Networking Foundation (ONF). *OpenFlow Switch Specifications 1.2 (ONF TS-003).* https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.2.pdf. October 2011.

[Ope12] Open Networking Foundation (ONF). *OpenFlow Switch Specifications 1.3.0 (ONF TS-006).* https://3vf60mmveq1g8vzn48q2o71a-wpengine.netdna-ssl.com/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf. October 2012.

[Ope13] Open Networking Foundation (ONF). *OpenFlow Switch Specifications 1.4 (ONF TS-012).* https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf. October 2013.

[Ope14a] Open Networking Foundation (ONF). *OpenFlow Switch Specifications 1.5.0 (ONF TS-020).* https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf. December 2014.

[Ope14b] Open Networking Foundation (ONF). *SDN Architecture Overview, Version 1.0, ONF TR-502.* https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR_SDN_ARCH_1.0_06062014.pdf. June 2014.

[Ope14c] Open Networking Foundation (ONF). *SDN Architecture Overview, Version 1.1, ONF TR-504.* https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR_SDN-ARCH-Overview-1.1-11112014.02.pdf. November 2014.

List of Figures

List of Tables

Acronyms

R^2 Coefficient of determination. 116, 117, 147–149, 183

A-CPI Application-Controller Plane Interface. 12, 14

AI Artificial Intelligence. 2, 114

ANN Artificial Neural Network. 110, 118, 123

AR Acceptance Ratio. 121, 130, 131

BA Barabási-Albert. 113, 114, 129–132, 142–145, 149, 182, 183

BC BCube Data Center Topology. 113, 114, 129, 130, 134, 145, 149

CPP Controller Placement Problem. 4, 70, 71

CPS Cyber-Physical System. 1

CPU Central Processing Unit. 5, 10, 16, 18

D-CPI Data-Controller Plane Interface. 11, 13, 14, 24, 26

DC Data Center. 114

DHPP Dynamic Hypervisor Placement Problem. 7, 75, 77, 78, 82–84, 182, 185

DViNE Deterministic Virtual Network Embedding Algorithm. 122, 129, 131, 132

EC Virtual Network Embedding Cost. 121

ER Erdős-Rényi. 113, 114, 129–133, 135, 142, 143, 145, 147, 150–153, 182, 183

FEAS Feasibility Seeking VNE Algorithm. 150, 151, 153, 185

FLP Facility Location Problem. 70

FN False Negative. 115

FP False Positive. 115

FT FatTree Data Center Topology. 113, 114, 129, 130, 133, 134, 145

FV FlowVisor. 21–23, 35–48, 53, 55–57, 62, 67, 158, 181

GDY Greedy Algorithm. 122, 129, 133, 143–149

GRC Global Resource Capacity Algorithm. 122, 129–133, 145, 146, 148, 149, 183

HF Hopfield. 134, 135

HF-DViNE Hopfield-enhanced Deterministic Virtual Network Embedding Algorithm. 129, 132, 133

HF-GRC Global Resource Capacity Algorithm with Hopfield Preprocessor. 129–133, 183

HF-GRC-DC-FB Global Resource Capacity Algorithm with Hopfield Preprocessor for Data Center Topologies. 129, 133, 134

HF-RViNE Hopfield-enhanced Randomized Virtual Network Embedding Algorithm. 129, 132, 133

HF-SDP Hopfield-enhanced Shortest Distance Path Algorithm. 129, 134, 135, 183

HPP Network Hypervisor Placement Problem. ii, 4, 7, 65, 66, 69, 70, 72, 74, 75, 77, 79, 81, 82, 87, 92, 185

InP Infrastructure Provider. 2, 10, 13

IP Internet Protocol. 10–12, 20, 23, 39, 52, 145

ISP Internet Service Provider. 9, 10

LB Load Balancing Algorithm. 122, 145–149

LLDP Link Layer Discovery Protocol. 21, 23

MAC Media Access Control. 23

McSDP Multi-controller Switch Deployment Problem. ii, 65, 66, 72, 75, 77, 79, 81, 185

MILP Mixed Integer Linear Programming. 7, 8, 66, 72, 79, 86, 107, 110, 136, 143, 144, 148

ML Machine Learning. 3, 5, 7, 8, 110, 111, 114–116, 118, 136, 137, 139–143, 145–148, 150–155, 157–160, 183, 185

MPLS Multiprotocol Label Switching. 10, 17

NFV Network Function Virtualization. 28

NOS Network Operating System. 2, 14, 16

NV Network Virtualization. 1–3, 6, 7, 9, 10, 12–15, 19, 109, 154, 157, 159

OF OpenFlow. 12, 13, 17–27, 29, 31–36, 38–40, 42, 44, 50–56, 58–63, 67, 158, 159, 181, 182

ONF Open Networking Foundation. 20

OSI Open Systems Interconnection. 51, 52

OvS Open vSwitch. 37–39, 41, 53, 54, 57, 60

OVX OpenVirteX. 22, 23, 35–48, 62, 158, 181

PCA Principal Component Analysis. 142

PCP Priority Code Point. 22

QoS Quality-of-Service. 1, 21, 71, 160

RCR Revenue-Cost-Ratio. 121, 123, 130, 131, 133

REV Revenue. 121, 131, 132

RMSE Root Mean Squared Error. 117, 147, 148

RND Random Algorithm. 143, 144

RViNE Randomized Virtual Network Embedding Algorithm. 122, 129, 131, 132

SDN Software-Defined Networking. 1–27, 29–35, 37, 38, 41, 46–51, 53, 58–60, 62, 65–68, 70–79, 81–83, 86, 87, 92, 93, 97, 102, 107, 109, 136, 139, 140, 154, 157–160, 181, 185

SDP Shortest Distance Path Algorithm. 122, 129, 134, 135, 145–151, 153, 154, 183, 185

SLA Service Level Agreement. 19, 54

SP Service Provider. 2, 9, 10

TCAM Ternary Content-Addressable Memory. 12, 16, 22, 49

TCP Transmission Control Protocol. 12, 20, 22, 32, 41, 42, 52, 55, 58, 59, 70, 136

TN True Negative. 115

TP True Positive. 115

TR Total Revenue. 121, 130

TZ Topology Zoo. 86, 113, 114, 130–133, 143–145, 182, 183

UDP User Datagram Protocol. 33, 34

VCP Virtual Control Path. 69, 70, 76, 77, 80, 97, 98, 103–105, 107

VLAN Virtual Local Area Network. 9, 10, 21, 22

VM Virtual Machine. 2, 28

VN Virtual Network. 131, 135, 149

VNE Virtual Network Embedding. 5, 7, 109, 110, 119–124, 128–133, 136, 139, 141, 142, 145–148, 150–155, 159, 182, 183

VNF Virtual Network Function. 71, 72

VNR Virtual Network Request. 119–121, 124, 126, 129–135, 139, 141, 142, 145–150, 152–155, 183

vSDN Virtual Software-Defined Network. 2–4, 12–14, 17, 20, 22, 29–32, 34, 36, 41, 47, 50–52, 58, 62, 65–67, 69–81, 83–94, 96–100, 102–107, 157, 181, 182, 185

WAX Waxman. 113, 114, 130, 142