Technische Universität München
TUM School of Computation, Information and Technology

Quality-of-Service-Aware Virtualization of Programmable Networks

Nemanja Đerić, M.Sc.

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

genehmigten Dissertation.

Vorsitz:		Prof. Dr.-Ing. Thomas Eibert
Prüfer*innen der Dissertation:	1.	Prof. Dr.-Ing. Wolfgang Kellerer
	2.	Prof. Dr.-Ing. Ralf Steinmetz

Die Dissertation wurde am 14.11.2022 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 22.03.2023 angenommen.

Quality-of-Service-Aware Virtualization of Programmable Networks

Nemanja Đerić, M.Sc.

22.03.2023

Abstract

Communication network are one of the main pillars of our digital society. That is, they enable the exchange of information between humans, machines, computers, mobile phones, satellites and so forth over both, short and long distances. In addition, communication networks and the corresponding technologies are also always evolving in order to support the requirements of cutting-edge services, or to utilize the available physical resources in a more efficient manner.

Particularly, to utilize the resource more efficiently, network operators are currently striving to utilize two novel concepts in their networks, i.e., network programmability through Software-Defined Networking (SDN) and network virtualization. Firstly, by decoupling the Control Plane (CP) from the Data Plane (DP), SDN offers network operators the opportunity to program the behaviour of their network dynamically from a logically centralized place (i.e., through SDN controller) according to the current traffic demands. Therefore, with SDN, network operators can utilize their resources in a more optimal way. Secondly, motivated by the cloud revolution, network operators are additionally aiming to utilize virtualization technologies within the physical network itself. That is, they aim to provide Network as a Service (NaaS) through sharing of the phyiscal infrastructure on-demand between multiple tenants. By utilizing softwarized network hypervisor (or SDN hypervisor), it becomes possible to combine both virtualization and network programmability at the same time, thus obtaining the benefits of both technologies. That is, virtualized programmable (or SDN) networks enable tenants and operators at the same time: *(1)* network programability from a centralized location and *(2)* sharing of the physical infrastructure.

On the other hand, future networks are envisioned to support novel services coming from technological fields such as Industry 4.0 and Multi-access Edge Computing (MEC). Services such as flexible factories, automated manufacturing, smart cities, and virtualized manufacturing control are particularly attractive. However, these services often have very high Quality of Service (QoS) traffic requirements which networks should satisfy. For example, to avoid downtime in flexible factories, the DP traffic often must be delivered with no packet loss and bounded end-to-end delay. However, doing both at the same time, i.e., supporting services with high QoS requirements in virtualized programmable networks, is still challenging. Firstly, the data plane devices (e.g., swithces) and resources (e.g., links, buffers, queues) are shared between tenants. Thus, it is cruical to ensure that the tenants do not overutilize the resources and degrade the performance of traffic with high QoS requirements. In addition, the forwarding behaviour of programmable networks is often updated, therefore, networks have to be updated in a consistent and predictable manner (e.g., within a certain time).

Therefore, this thesis designs, develops, and investigates new concepts which enable supporting services with high QoS requirements in virtualized programmable networks. In addition to network virtualization concepts, also the generic networking concepts which aim to enable such realization are studied. The major contributions of this thesis can be divided into two groups. The first group studies how to provide QoS guarantees in the CP. That is, initially, we present a novel methodology which provisions the CP hardware resources (i.e., of network hypervisors) while ensuring that the expected QoS is achieved. This was achieved by desiging the accurate resources estimation models based on the two comprehensive measurement campaigns. In addition, in this thesis, a novel network update mechanism relying on in-band CP is presented (part of the introduced `NAGA` system). It enables reliable delivery of CP messages to the physical devices and consistent network updates. Hence, the combination of the two previous approaches enable network operators to achieve high *end-to-end* QoS in CP while managing their resources efficiently.

The second major group aims to study concepts needed for practical realization of a virtualized programmable network with deterministic DP guarantees and isolated traffic. To do so, initially a novel traffic policing measurement methodology is presented. By utilizing it, it becomes possible to model the worst-case traffic policing performance of programmable switches and end-hosts. Afterwards, by utilizing the previously introduced measurement methodology, we were able to design a system (i.e., `NAGA`) which is capable of providing deterministic DP guarantees in programmable networks. In addition to relying on the measurement and modeling methodology, it uses centralized network control, and only priority queuing and label-based forwarding capabilities of programmable switches. Therefore, it can be deployed in virtualized programmable networks. By performing real implementations and deployments in multiple testbeds, in this thesis, it was demonstrated that the proposed solutions perform as expected and are able to provide guaranteed performance in the considered scenarios.

Kurzfassung

Kommunikationsnetze sind eine der wichtigsten Säulen unserer digitalen Gesellschaft. Sie ermöglichen den Austausch von Informationen zwischen Menschen, Maschinen, Computern, Mobiltelefonen, Satelliten usw., sowohl über kurze als auch über lange Entfernungen. Darüber hinaus entwickeln sich die Kommunikationsnetze und die entsprechenden Technologien ständig weiter, um den Anforderungen modernster Dienste gerecht zu werden oder die verfügbaren physischen Ressourcen effizienter zu nutzen.

Um die Ressource effizienter zu nutzen, bemühen sich die Netzbetreiber derzeit um den Einsatz zweier neuartiger Konzepte in ihren Netzen, d. h. die Programmierbarkeit des Netzes durch Software-Defined Networking (SDN) und die Netzvirtualisierung. Erstens bietet SDN durch die Entkopplung des Control Plane (CP) vom Data Plane (DP) den Netzbetreibern die Möglichkeit, das Verhalten ihres Netzes dynamisch von einer logisch zentralen Stelle aus (d.h. durch den SDN-Controller) entsprechend den aktuellen Verkehrsanforderungen zu programmieren. Mit SDN können Netzbetreiber ihre Ressourcen daher optimaler nutzen. Zweitens streben Netzbetreiber, motiviert durch die Cloud-Revolution, zusätzlich die Nutzung von Virtualisierungstechnologien innerhalb des physischen Netzes selbst an. Das heißt, sie wollen Network as a Service (NaaS) durch die gemeinsame Nutzung der physischen Infrastruktur auf Abruf zwischen mehreren Mietern bereitstellen. Durch den Einsatz eines softwarisierten Netzwerk-Hypervisors (oder SDN-Hypervisors) wird es möglich, Virtualisierung und Netzwerkprogrammierbarkeit gleichzeitig zu kombinieren und so die Vorteile beider Technologien zu nutzen. Das heißt, virtualisierte programmierbare (oder SDN) Netze ermöglichen es Mietern und Betreibern gleichzeitig: *(1)* Netzwerkprogrammierbarkeit von einem zentralen Standort aus zu nutzen und auch die *(2)* gemeinsame Nutzung der physischen Infrastruktur.

Andererseits sollen künftige Netze neuartige Dienste unterstützen, die aus Technologiebereichen wie Industrie 4.0 und Multi-access Edge Computing (MEC) stammen. Dienste wie flexible Fabriken, automatisierte Fertigung, intelligente Städte und virtualisierte Fertigungssteuerung sind besonders attraktiv. Diese Dienste haben jedoch oft sehr hohe Anforderungen an den Datenverkehr, denen die Netze gerecht werden müssen. Zum Beispiel, um Ausfallzeiten in flexiblen Fabriken zu vermeiden, muss der DP-Verkehr oft ohne Paketverlust und mit begrenzter Ende-zu-Ende-Verzögerung zugestellt werden. Darüber hinaus müssen Netzwerkaktualisierungen konsistent und zeitnah geliefert und ausgeführt werden. Beides gleichzeitig zu tun, d. h. Dienste mit hohen Quality of Service (QoS)-Anforderungen in virtualisierten programmierbaren Netzen zu unterstützen, ist jedoch immer noch eine Herausforderung. Erstens werden die Geräte der Datenebene (z. B. Switches) und die Ressourcen (z. B. Links, Puffer, Warteschlangen) von verschiedenen Benutzern gemeinsam genutzt. Daher muss sichergestellt werden, dass die Benutzer die Ressourcen nicht übermäßig nutzen und die Leistung

von Datenverkehr mit hohen Anforderungen an die QoS beeinträchtigen. Darüber hinaus wird das Weiterleitungsverhalten programmierbarer Netze häufig aktualisiert, so dass die Netze auf konsistente und vorhersehbare Weise (z. B. innerhalb einer bestimmten Zeit) aktualisiert werden müssen.

Daher werden in dieser Arbeit neue Konzepte entworfen, entwickelt und untersucht, die es ermöglichen, Dienste mit hohen QoS-Anforderungen in virtualisierten programmierbaren Netzwerken zu unterstützen. Zusätzlich zu den Netzwerkvirtualisierungskonzepten werden auch die generischen Netzwerkkonzepte untersucht, die eine solche Realisierung ermöglichen sollen. Die Hauptbeiträge dieser Arbeit lassen sich in zwei Gruppen unterteilen. Die erste Gruppe untersucht, wie man QoS-Garantien in CP bereitstellen kann. Das heißt, wir stellen zunächst eine neuartige Methodik vor, die die CP-Hardware-Ressourcen (d. h. von Netzwerk-Hypervisoren) bereitstellt und gleichzeitig gewährleistet, dass die erwarteten QoS erreicht werden. Dies wurde durch die Entwicklung genauer Modelle zur Ressourcenschätzung auf der Grundlage der beiden umfassenden Messkampagnen erreicht. Darüber hinaus wird in dieser Arbeit ein neuartiger Netzaktualisierungsmechanismus vorgestellt, der sich auf in-band CP stützt (Teil des eingeführten `NAGA` systems). Es ermöglicht die zuverlässige Zustellung von CP-Nachrichten an die physischen Geräte und konsistente Netzwerkaktualisierungen. Durch die Kombination der beiden vorangegangenen Ansätze können Netzbetreiber also eine hohe *end-to-end* QoS in CP erreichen und gleichzeitig ihre Ressourcen effizient verwalten.

Die zweite Hauptgruppe zielt darauf ab, Konzepte zu untersuchen, die für die praktische Realisierung eines virtualisierten programmierbaren Netzwerks mit deterministischen DP-Garantien und isoliertem Verkehr erforderlich sind. Zu diesem Zweck wird zunächst eine neuartige Messmethodik des traffic-policing vorgestellt. Mit ihrer Hilfe ist es möglich, die Worst-Case Traffic-Policing-Performance von programmierbaren Switches und End-Hosts zu modellieren. Durch die Verwendung der zuvor vorgestellten Messmethodik konnten wir anschließend ein System (d. h. `NAGA`) entwerfen, das in der Lage ist, deterministische DP-Garantien in programmierbaren Netzwerken bereitzustellen. Es stützt sich nicht nur auf die Mess- und Modellierungsmethodik, sondern nutzt auch eine zentralisierte Netzwerksteuerung und ausschließlich die Prioritätswarteschlangen- und Label-basierten Weiterleitungsfunktionen programmierbarer Switches. Daher kann es auch in virtualisierten programmierbaren Netzwerken eingesetzt werden. Durch die Durchführung realer Implementierungen und Einsätze in mehreren Testumgebungen wurde in dieser Arbeit gezeigt, dass die vorgeschlagenen Lösungen wie erwartet funktionieren und in der Lage sind, in den betrachteten Szenarien eine garantierte Leistung zu erbringen.

Contents

1 Introduction **1**
- 1.1 Research Challenges . . . 3
- 1.2 Contributions . . . 5
- 1.3 Outline . . . 7

2 Background **9**
- 2.1 Programmable Networks . . . 9
  - 2.1.1 Software-Defined Networking . . . 9
  - 2.1.2 Programming Protocol-independent Packet Processors (P4) . . . 15
- 2.2 Virtualization . . . 16
  - 2.2.1 Server Virtualization and Containerization Approaches . . . 16
  - 2.2.2 NFV Architecture . . . 18
  - 2.2.3 Virtualization of Programmable SDN Networks . . . 18
- 2.3 Deterministic Performance in SDN Networks . . . 21
  - 2.3.1 General Notion of Determinism in Networking and Literature Survey . . . 21
  - 2.3.2 DNC Theory . . . 22
  - 2.3.3 Chameleon System . . . 23
- 2.4 Summary . . . 26

3 QoS-Aware Network Hypervisor Resource Provisioning **29**
- 3.1 Function-Aware Virtual Network Embedding Problem . . . 31
  - 3.1.1 Related Work . . . 31
  - 3.1.2 Topology Abstraction Measurements . . . 32
  - 3.1.3 Modelling Central Processing Unit (CPU) Estimation . . . 36
  - 3.1.4 VNE Problem Formulation . . . 36
  - 3.1.5 Simulation Scenario . . . 41
  - 3.1.6 Simulation Results . . . 42
  - 3.1.7 Insights and Discussion . . . 45
- 3.2 QoS-Aware Network Hypervisor Resource Provisioning . . . 47
  - 3.2.1 Motivation: Predictable Virtual Network Performance . . . 47
  - 3.2.2 Related Work . . . 48
  - 3.2.3 Network Hypervisor Benchmarking . . . 50
  - 3.2.4 Hypervisor CPU Prediction Model . . . 59

3.2.5 Model Evaluation . . . 62
3.2.6 Insights and Discussion . . . 69
3.3 Demonstration of Application-Aware NH Reconfiguration . . . 70
3.3.1 Demo Setup and Scenario . . . 70
3.3.2 Demo Presentation . . . 72
3.3.3 Insights and Discussion . . . 72
3.4 Conclusion . . . 73
3.5 Future Work . . . 74

4 Traffic Policing 75
4.1 Background . . . 77
4.1.1 The Need for Traffic Policing . . . 77
4.1.2 Leaky Token Bucket Algorithm . . . 78
4.2 Related Work . . . 80
4.3 Procedure for Measuring Traffic Policing Accuracy . . . 81
4.3.1 Generic Measurement Setup . . . 81
4.3.2 Standard Measurement Procedure . . . 82
4.3.3 DNC-Based Measurement Procedure . . . 83
4.3.4 Deriving Rate and Burst . . . 84
4.4 Switch Measurements . . . 84
4.4.1 OpenFlow Meters . . . 85
4.4.2 Setup . . . 85
4.4.3 Results . . . 86
4.5 End-Host Policing . . . 91
4.5.1 DPDK Application . . . 92
4.5.2 Measurement Setup . . . 95
4.5.3 Results . . . 96
4.6 Discussion . . . 96
4.7 Conclusion . . . 98
4.8 Future Work . . . 99

5 Providing Control and Data Plane Guarantees in Programmable Networks 101
5.1 Motivation and Contribution . . . 102
5.2 System Model . . . 103
5.2.1 Achieving CP Consistency . . . 105
5.2.2 Scheduling Algorithm . . . 108
5.3 Edge Switch Predictability Measurement . . . 111
5.3.1 Edge Switch Deployment . . . 111
5.3.2 Data Plane Processing Time . . . 112
5.3.3 Traffic Metering . . . 114
5.3.4 Buffer Management and Priority Queuing . . . 114
5.3.5 Control Plane Predictability . . . 115
5.4 Performance Evaluation . . . 118

5.4.1 System Deployment . . . 118
5.4.2 Verifying Network Update Consistency . . . 121
5.4.3 Cost of In-Band Control Plane . . . 123
5.4.4 Scalability Analysis . . . 123
5.5 Related Work . . . 125
5.6 Conclusion . . . 126

6 Conclusion and Future Work **127**
6.1 Summary . . . 128
6.2 Future Work . . . 130

Bibliography **133**

Acronyms and Abbreviations **147**

List of Figures **151**

List of Tables **157**

Chapter 1

Introduction

Communication networks represent the backbone of our society. By connecting people, machines, data centers, and other devices they enable the exchange of information. Throughout history, computer networks were always evolving to support novel services and to satisfy upcoming requirements. For instance, during the 60s, to support services with high survivability (requirement of US Air Force) *packet*-switched networks were developed and they slowly started replacing *circuit*-switched networks. Over the last few years, novel requirements are emerging from the industry, and they require further evolvement of computer networks [Jab+16]; [Wan+20]; [BAP17].

The fourth industrial revolution [BAP17], or Industry 4.0, strives to utilize information and communication technologies, to enable automated manufacturing, convertible factories, and intelligent logistics. To enable such services, the networks are now supposed to support Machine-to-Machine (M2M) communication [Zha+17] at the same time as human-based communication. M2M communication typically has high Quality of Service (QoS) requirements [LC11] as it can involve the exchange of data between sensors, actuators, controllers, cloud systems, and 5G networks. For instance, one of the recent trends in the industry is virtualizing manufacturing hardware controllers (e.g., Programmable Logic Controller (PLC)) and moving them to the cloud [Giv+14]. The virtual controllers (e.g., vPLCs) are then connected to sensors and actuators over a shared network and can control the manufacturing processes. To enable such scenarios, the networks must provide *high* per-packet QoS guarantees such as bounded packet delay and loss, and high reliability [MN22]. Failing to ensure that these guarantees are met can cause catastrophic failures in industrial networks. For example, losing control of a manufacturing robot in a factory even for a split second could cause serious injuries to the people working besides it.

On the other hand, changing, updating, and upgrading computer networks whenever a new service or application emerges is rather expensive. Therefore, during the last couple of years, programmable networks are gaining more traction to enable faster innovation and adaptability [Nun+14]; [KCB21]. Technologies such as Software-Defined Networking (SDN) [McK+08] and Programming Protocol-independent Packet Processors (P4) [Bos+14] are particularly attractive as they offer programmability in the Control Plane (CP) and Data Plane (DP) of a network, respectively. That is, SDN decouples the CP from the DP devices (e.g., switches, routers) and it places it in a logically centralized controller. In such a network, the controller is responsible for managing the network, i.e., it handles the networking traffic and generates the forwarding rules for the underlying

DP devices. The rules can be delivered to SDN-enabled devices with an open and standardized CP protocol such as Open Flow (OF) [Fou21].

On top of that, virtualized programmable networks are emerging. That is, the recent cloud computing revolution demonstrated that the benefits of virtualization such as cost reduction and scalability far outweigh the drawbacks [SUK19]; [Mas+14]. Therefore, intending to achieve the same benefits, solutions for virtualizing programmable networks started emerging [She+09]; [Al-+14]. In virtualized programmable networks [She+09] (or virtualized SDN networks), tenants (or virtual network users) can request virtual networking resources, and later control them with their SDN controller. The integral component realizing virtualization of a programmable network is a Network Hypervisor (NH) [She+09]. It is logically located between the tenants' SDN controllers and the physical infrastructure (e.g., forwarding devices/switches). Moreover, it processes all the CP messages (e.g., switch forwarding rules) exchanged by the tenants' SDN controllers and the physical switches. Since virtualization is a complex process, NHs are often developed as a software application running on commodity hardware.

Virtualized programmable networks offer benefits of the both worlds. On one hand, virtualization enables network operators to share the physical network between multiple tenants. Hence, the physical resources can be more efficiently used, and virtual resources can be more easily scaled up or down. On the other hand, programmable technologies offer network operators and tenants the ability to control their resources dynamically. Thus, the network operation can be tailored to the current demands.

However, virtualized programmable networks are still not able to support high QoS requirements of the aforementioned services (e.g., from Industry 4.0). Therefore, the main goal of this thesis is to investigate, develop, and deploy novel concepts for enabling QoS-aware virtualization of programmable networks. In addition to improving virtualization concepts, this thesis also enhances general mechanisms and technologies which enable the realization of programmable network virtualization with QoS guarantees.

Fig. 1.1 illustrates the architecture of virtualized programmable (i.e., SDN) networks. In addition, it also highlights the mapping of open reasearch questions and the main contributions of this thesis to the architecture. The open research questions are discussed in more detail in Section 1.1, while, the contributions are presented in Section 1.2. The outline and structure of this thesis is shown in Section 1.3.

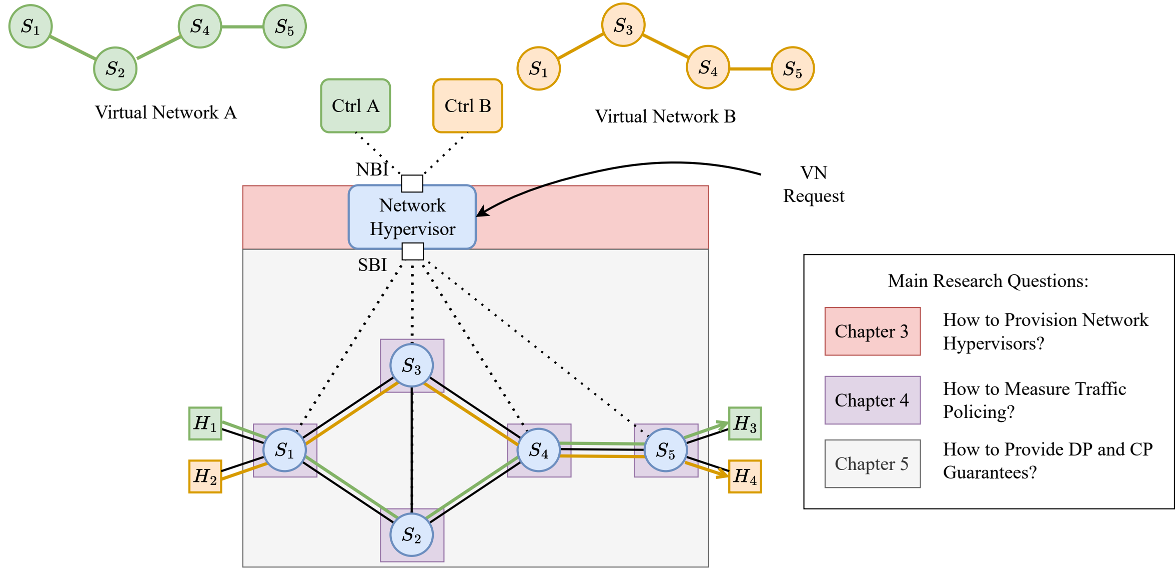


Figure 1.1: Architecture of virtualized SDN network. Additionally, the main research areas of the contributions presented in this thesis are highlighted. In this scenario, the NH virtualizes the underlying physical network, and it provides two virtual SDN networks to two different tenants (i.e., to tenants *A* and *B*). Thus, enabling the tenants to control their own virtual network through their SDN controller.

1.1 Research Challenges

In this section, the main research challenges of designing and deploying QoS-aware virtualized programmable networks are presented.

Provisioning and estimating the required NH resources.

In virtualized SDN networks, the performance of an NH can influence the underlying DP network [Ble+19]. For instance, if an NH loses a CP packet containing a forwarding rule, it might mean that two hosts might not be able to communicate with each other. Therefore, to ensure that the performance of an NH meets certain QoS, it is crucial to correctly provision it. However, doing so is challenging as it requires deep knowledge of NH requirements.

In the DP, deriving the requirements (e.g., bitrate) for applications is often easy as they are usually already known. For instance, YouTube recommendation for streaming 1080p video at standard frame rate is 8$Mbps$ [You]. Although CP requirements can be expressed with certain parameters such as network update rate, deriving the required NH resources is still challenging. That is, as NHs are realized as software instances, it is challenging to map these CP requirements to the corresponding hardware resource requirements (e.g., Central Processing Unit (CPU), memory). Moreover, NHs offer many additional and optional functionalities [Ble+16a] (e.g., topology abstraction) which might have an impact on the required resources. Even though there are approaches in the literature that focus on NH resource estimation [Sie+16]; [SOK17], they are not suitable for resource provisioning. To elaborate, they estimate only the average resource utilization and they do not consider various NH functions. Therefore, novel models for estimating the real hardware requirements are needed, and they should be NH aware. In addition to the estimation models, novel solutions are needed which can

utilize these models to provision the resources of an NH while ensuring that there is no performance degradation.

CP aware virtual network embedding algorithms.

Most of the *state-of-the-art* (*SotA*) Virtual Network Embedding (VNE) algorithms consider only the DP networking resources (e.g, link bandwidth, delay) and generic compute (e.g., CPU) resources on virtualized servers. However, the separation of CP and DP and the introduction of NHs adds a new dimension to the VNE algorithm – control plane. Failing to consider CP requirements and resources might result in a sub-optimal VNE algorithms. However, including them is challenging as NHs offer different functionalities which can have an impact of the resource requirements [Ble+19]. Additionally, different NHs might have different architectures, thus, making the problem even more complex. Therefore, novel control plane aware VNE algorithms are needed, and they should include different NH functionalities and architectures.

Modeling and measuring the performance of traffic policing.

In virtualized environments, the DP is shared between tenants. Thus, if two tenants share the same resources (e.g., tenants A and B share link S_4, S_5 in Fig. 1.1), it can happen that one tenant overutilizes the resources at the expense of the other one. Therefore, it is cruical to provide DP isolation between tenants. To do so, NHs can rely on the plethora of available solutions which provide high QoS guarantees in the literature. Most of these solutions rely on either in-network or host-based traffic policing to ensure that the hosts are not exceeding the agreed rate [Jan+13]; [Jan+15]. In addition, they often assume that the offered performance of such traffic policing entities is ideal. However, even though there are many measurement studies of OF-enabled devices [Van+19a]; [Kuź+18]; [Bau+18]; [DBK15], none of them evaluated the performance of in-network traffic policing. Therefore, novel measurement studies are needed to understand how the devices police the traffic and what is the offered accuracy. In addition, novel modeling methodologies which enable extracting the crucial data from the measurements in an automatic manner are necessary as well.

Achieving DP guarantees and consistent and timed network updates.

To support services with high QoS requirements (or to enable DP isolation), network operators could utilize one of the already available *SotA* solutions providing deterministic guarantees in the DP. However, the best performing systems are tailored to Data Center (DC) networks [Jan+15]; [Van+20] and are not suitable for virtualized networks. For instance, currently the best performing system is *Chameleon* [Van+20] and it achieves 50% higher network utilization compared to the other solutions. Unfortunately, *Chameleon* is not usable in virtualized networks as it offloads certain functionalities to the applications running on end-hosts. However, in virtualized networks, end-hosts belong to the tenants or virtual network users and not to the network operator (see Fig. 1.1). Therefore, either the current *SotA* systems have to be enhanced or novel systems have to be designed to achieve DP isolation and high network utilization in virtualized scenarios.

In addition to isolation, in virtualized environments, forwarding devices (e.g., switches) are more often reconfigured as the physical network is shared between the tenants. Hence, the CP of the corresponding hardware devices can be overloaded, and this can cause unexpected device behavior.

For example, certain *Pica8* switches may ignore certain CP messages (e.g., adding a forwarding rule) if the CP at that time is overloaded [Van+19a]. Therefore, it is crucial to schedule the updates to ensure that the forwarding devices are updated consistently and correctly without impacting the already embedded flows. Even though many solutions aim to achieve consistent network updates [Zho+21]; [NCC17], they do not consider DP with high QoS requirements and they support only low update rates. Therefore, novel and faster network update solutions that ensure consistency and correctness are needed.

1.2 Contributions

The contributions presented in this thesis are summarized in this section. The main objective of this thesis is to enable the development of QoS-aware virtualized programmable networks by developing and designing either new technological concepts or enhancing the already existing ones. An overview of the structure of this thesis is illustrated in Fig. 1.2. It highlights the three main research directions in addition to the methodologies, concepts, and corresponding publications from the author. Fig. 1.1 presents the mapping of the contributions to the research areas of virtualized SDN networks.

To begin with, the background chapter (i.e., Chapter 2) sets the scene for the following chapters by introducing the necessary basic information regarding the current *SotA* networking concepts. It introduces the concept of programmable networks (including both SDN and P4) and it provides an overview of the realization of virtualized programmable networks. In addition, it also provides detailed insights regarding architectures of virtualized programmable network, performance issues of SDN-enabled switches, and the best performing deterministic systems at the current time (i.e., *Chameleon*).

The first group of contributions of this thesis are the development of a new NH provisioning methodology and the investigation of VNE approaches (see Chapter 3). Network provisioning and VNE solutions in the literature both neglected the NH requirements (e.g., in terms of resources such as CPU). For instance, most of the VNE algorithms only considered the DP requirements when embedding Virtual Network (VN)s. Therefore, in this part, to understand the possible impacts, we present and study the results of two comprehensive NH measurement campaigns. To be precise, we measure the resource utilization of two most famous NHs, i.e., *FlowVisor* [She+09] and *OpenVirtex* [Al-+14], in difference scenarios. By studying the measurement traces, it was possible to develop very accurate models capable of estimating the required CPU resources based on the various scenario parameters. The evaluations showed that the developed models exhibit a relative average error of around 4%. By using these models, firstly, we were able to design a novel VNE algorithm based on a carefully designed Integer Linear Programming (ILP) model which includes both DP and CP requirements. By comparing the developed model with a baseline, it was possible to demonstrate that including NH requirements is crucial for developing an optimal VNE algorithm. Secondly, we have developed and evaluated a novel measurement-based methodology that can provision the resources of an NH for almost arbitrary networks, while ensuring that there is no significant performance degradation. The performance of this provisioning approach was demonstrated on a real test-bed available at our chair. One of the main benefits of this approach is the reduction of resource over-provisioning

which could be particularly useful for network operators as they could utilize their resources more optimally. Since both of the previously mentioned approaches focus on the CP aspect of virtualized programmable networks, this group of contributions can can be mapped to the research questions related to the NH area (see Fig. 1.1).

The second major group of contributions presented in this thesis is related to the modeling, measuring, and realization of traffic policing in *in-network* OF-enabled switches and *end-host* based solutions (see Chapter 4). The challenges in this field are often overlooked in *SotA*. For example, the *SotA* measurement studies such as [DBK15] only measure and model the average traffic policing accuracy, which is insufficient for many systems and mathematical frameworks providing strict QoS guarantees [Van+20]. To be precise, they often require upper bounds and not average values. Therefore, to resolve this issue, we have developed and presented a measurement and modeling methodology that enables deriving more detailed traffic policing performance models. These models are rather focused on obtaining and modeling worst-case traffic policing performance (e.g., including all deviations). Hence, they can be used as input parameters to the mathematical frameworks which provide deterministic guarantees such as Deterministic Network Calculus (DNC) [LT01]. By utilizing the presented measurement methodology, it was possible to measure and model the accuracy of both *in-network* OF-enabled switches and *end-hosts* based solutions. This measurement study unraveled many interesting effects. For instance, *in-network* OF-enabled switches in certain scenarios exhibit very inaccurate traffic policing (e.g., the relative error reaches 60%). This motivated us to investigate the impact of such inaccuracies on the performance (e.g., number of accepted flows) of one *SotA* system which provides deterministic QoS guarantees. Overall, this group of contributions (e.g., methodology and measurements of the switches) enables network operators to model networking devices and deploy accurately various DP isolation strategies directly in their physical networking infrastructure (see Fig. 1.1).

The third major contribution of this thesis is system `NAGA` (see Chapter 5). It provides deterministic DP guarantees and consistent *on-demand* network updates with timing guarantees in programmable networks. Deterministic DP guarantees are provided by relying on a DNC-based admission control algorithm initially presented in [GVK17] and already deployed in a DC network in [Van+20]. However, to realize such a system in networks without *end-hosts* control, we have offloaded certain networking functions to the P4 hardware devices located at the network edge. That is, by utilizing the previously introduced traffic policing measurement methodology, we were able to measure the preformance of traffic policing in P4 switches, and to model it `NAGA`. Therefore, from a DP perspective, the combination of contributions presented in Chapters 4 and 5 enabled the practical realization of a DP supporting (without the need for end-host control) deterministic services. To provide the aforementioned CP benefits (i.e., consistent and timed network updates), `NAGA` relies on two mechanisms. Firstly, it uses an in-band CP embedded over an already discussed deterministic DP to ensure that the CP messages are delivered within a certain time. Secondly, by utilizing the developed network update scheduling algorithm, network updates are performed consistently and with timing guarantees. Whereas the real implementation of `NAGA` in a P4-based testbed demonstrates that applications indeed received guaranteed performance in terms of latency, data rate and network update timing guarantees, simulation studies show the ability of `NAGA` to be even deployed in large scale scenarios, which are common for virtualized networks. Therefore, in virtualized architecture

(see Fig. 1.1), in order to support services with high QoS, `NAGA` could be used to realize: *(1)* the CP network connected to the Southbound Interface (SBI) of an NH and *(2)* the physical DP network.

The CP focused contributions presented in Chapter 3 and 5 take an integral step towards providing *end-to-end* CP guarantees. Firstly, as mentioned, the CP aspects of `NAGA` can be used for the realization of control network with update timing guarantees between an NH SBI and physical devices (see Fig. 1.1). Secondly, the main contribution of Chapter 3 (i.e., QoS-aware resource provisioning) enables providing predictable processing time between the two NH interfaces (i.e., Northbound Interface (NBI) and SBI in Fig. 1.1) in dynamically changing scenarios.

1.3 Outline

The outline of this thesis is presented in this section. An overview is shown in Fig. 1.2.

Chapter 2 presents the necessary information regarding the related technologies and it sets the scene for this thesis. Section 2.1 presents two main branches of programmable networks, i.e., SDN and P4. Afterward, in Section 2.2, concepts such as server and network virtualization are introduced. Section 2.3 presents the details of one *SotA* DNC-based system (i.e., *Chameleon*) which provides DP deterministic guarantees.

Chapter 3 presents novel concepts which aim to improve the virtualization of programmable networks. Section 3.1 presents a measurement-based and CP aware VNE algorithm. This algorithm also includes the effect of different NH functions. QoS-aware NH CPU provisioning procedure is introduced, presented, and evaluated in Section 3.2. Section 3.3 demonstrates the benefits of VN reconfigurations in virtualized programmable networks.

Chapter 4 investigates the problems related to *in-network* and *end-host* based traffic policing. The proposed modeling and measurement methodology based on DNC mathematical framework is presented in Section 4.3. Section 4.4 and 4.5 present two measurement surveys aiming to shed light on the accuracy of traffic policing in hardware OF-enabled switches and software *end-host* based solutions. The impact of traffic policing inaccuracies on network utilization achieved by one *SotA* solution is discussed in Section 4.6.

Chapter 5 presents a novel system (i.e., `NAGA`) capable of providing deterministic data and CP guarantees. Additionally, it also enables consistent network updates. The system model and network architecture are introduced in Section 5.2. Section 5.3 presents in-depth measurement study of one P4-enabled hardware device. Section 5.4 presents the performance evaluation of `NAGA`system. Section 5.4.1 provides the details related to the deployment of the `NAGA` system in the test-bed available at our chair (i.e., at Chair of Communication Networks, Technical University of Munich). Section 5.4.2 verifies that `NAGA` updates the network consistently. Section 5.4.3 present the performance comparison of `NAGA` with the other *SotA* systems. Section 5.4.4 studies the scalability of `NAGA`.

Finally, this thesis is concluded in Chapter 6. Additionally, the potential interesting future research directions are included in the same chapter.

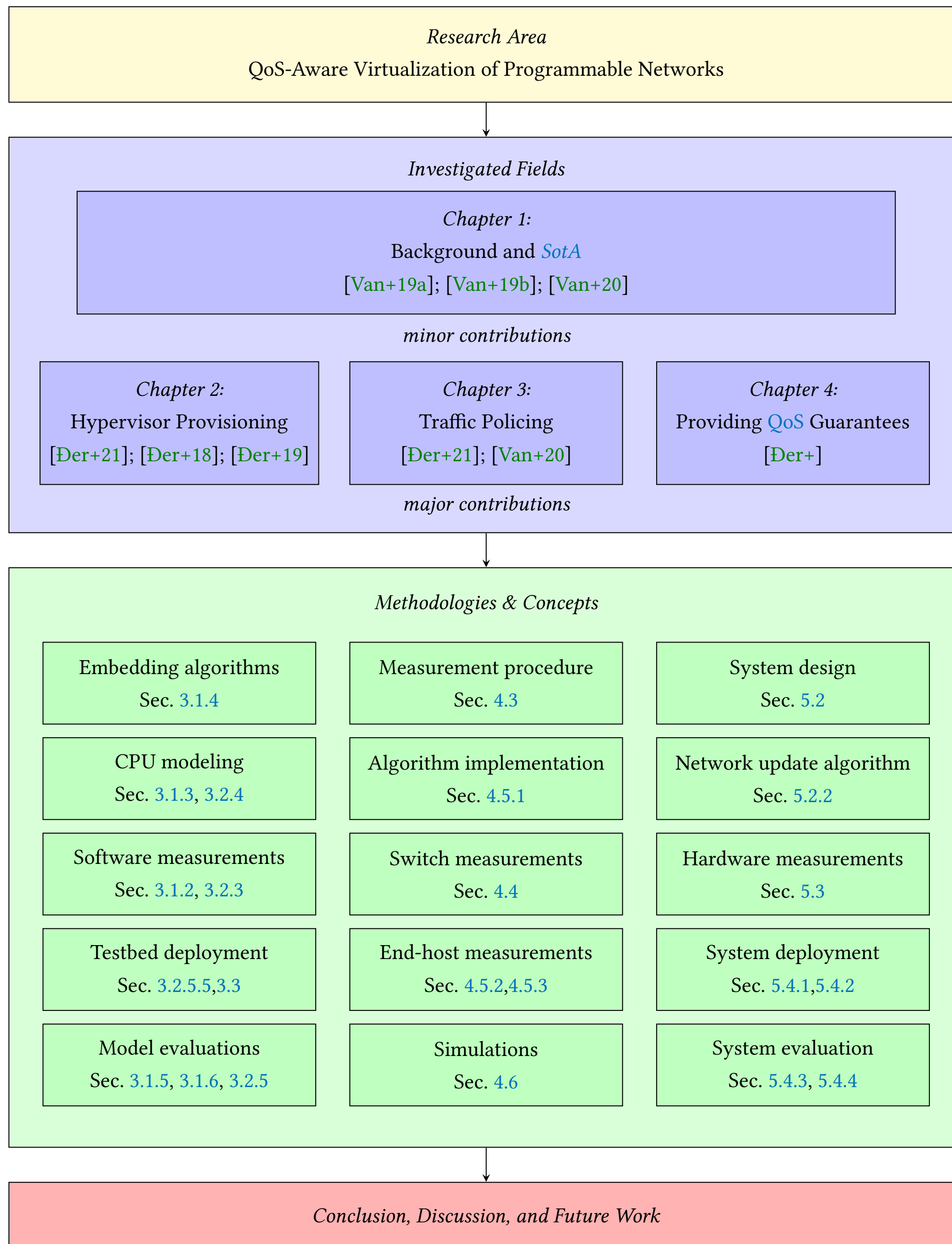


Figure 1.2: An overview of the structure of this thesis. In this thesis, three different main research direction related to the QoS-aware virtualization of programmable networks are investigated. The first direction focuses on provisioning of the CP of virtualized SDN networks. While the following two directions are focused on the challenges of achieving guaranteed QoS in the DP.

Chapter 2

Background

The first goal of this chapter is to provide the necessary basic background information needed for understanding the contributions presented in the following chapters of this thesis. The second goal of this chapter is to introduce in detail the network architectures of programmable networks and their requirements. To be precise, firstly, the architecture of non-virtualized and virtualized Software-Defined Networking (SDN) networks is presented. Afterward, the architecture and an overview of the most performing Data Center (DC) deterministic network, i.e., *Chameleon*, is presented. Understanding these architectures and their requirements is important as Chapter 3 proposes concepts that enhance the operation of the Control Plane (CP) of virtualized SDN networks, while Chapter 5 resolves certain drawbacks and improves *Chameleon* even further.

This chapter is structured as follows. Section 2.1 introduces the novel approaches in programmable networks. This includes SDN (see Sec. 2.1.1) and Programming Protocol-independent Packet Processors (P4) (see Sec. 2.1.2). Section 2.2 provides background information regarding server virtualization (see Sec. 2.2.1), network virtualization (see Sec. 2.2.3), and Network Function Virtualization (NFV) (see Sec. 2.2.2). Finally, Section 2.3 discusses the deterministic networks and it provides an overview of *Chameleon* system.

2.1 Programmable Networks

There are two main branches of programmable networks. The first branch is SDN, which aims to enable programming of the CP of a networks. While the second one is P4, which is focused on providing programmability of the Data Plane (DP).

2.1.1 Software-Defined Networking

In traditional legacy networks, CP and DP are coupled. Meaning that all of the devices (e.g., switches or routers) are in charge of both, forwarding the packets in a DP, and exchanging the control information (e.g., routing decisions) in a CP. To be precise, to exchange information, the devices relied on distributed control protocols such as Border Gateway Protocol (BGP). The legacy network architecture is illustrated in Fig. 2.1a.

SDN proposes to decouple CP and DP [McK+08], as depicted in Fig. 2.1b. In SDN, the control of a network is determined by a logically centralized SDN controller. They are typically realized

as software applications (e.g., Java programs) running on generic computing servers. A centralized SDN controller is connected via one of the SDN protocols (e.g., Open Flow (OF) [Fou21]) to all of the SDN-enabled switches, and it controls them during the network operation. For instance, it could insert routing rules to route the traffic between hosts (or users). In SDN, the networking switches have to only implement the DP functionality and an agent for communication with a controller.

Controlling the DP packets in SDN networking (hardware) switches is done with a *match* and *action* paradigm. That is, each SDN switch maintains a flow table that can contain multiple entries. A *flow* entry is typically defined with a specific *match* and one or more *actions*. *Matching* is usually done based on the packet headers (e.g., source Media Access Control (MAC) address). One example of an *action* is forwarding the packet on a certain port of a switch. To summarize the whole procedure, upon packet reception, an SDN-enabled switch firstly tries to find a *flow* to which the corresponding packet belongs (i.e., *matching*). This is done based on the header of a packet and all of the entries in its *flow* table. After finding the *matching flow*, the switch applies the *action(s)* of that *flow* to the corresponding packet.

To control the behavior of a network (e.g., routing), a centralized SDN controller populates *flow* tables of all the deployed DP switches in a network via one of the SDN-based protocols.

There are many benefits of SDN [TPR14]; [Thi+19], for example:

1. **Centralized Network Control.** In SDN [McK+08], the control of a network is logically centralized. Therefore, monitoring a network and deploying advanced traffic steering algorithms becomes easier. Utilizing such technologies enables network operators to potentially increase their network utilization.

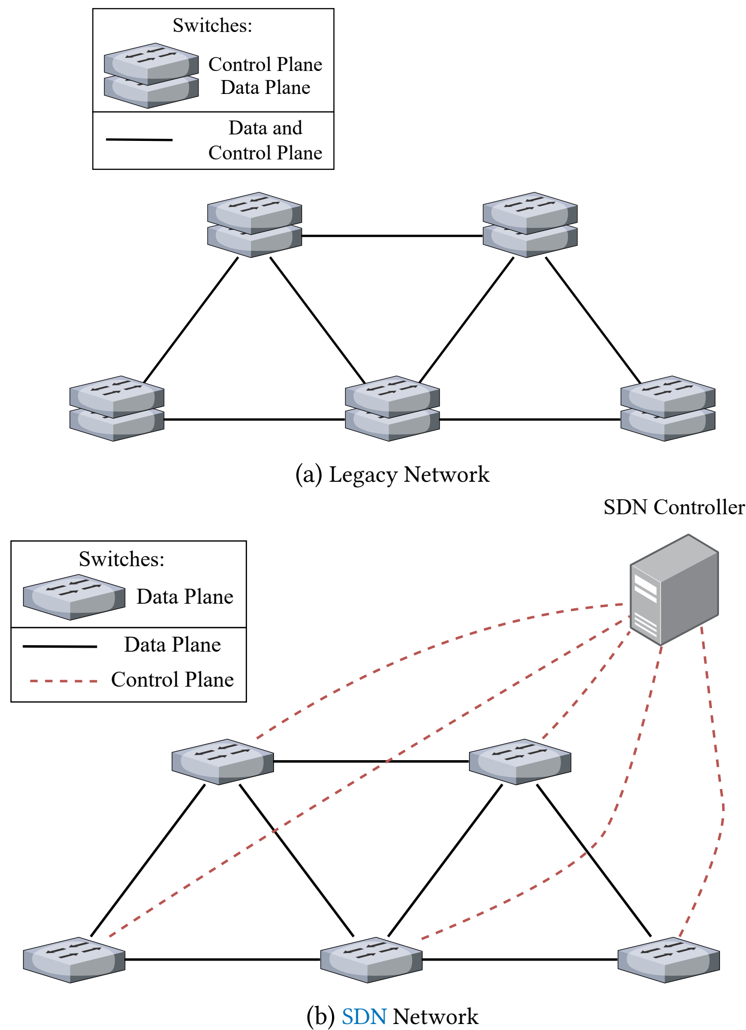
2. **Simple Switches.** In SDN networks, the switches only have to implement the DP forwarding, and an agent for communicating with a centralized SDN controller. This means that switch vendors do not have to implement and maintain all of the available CP protocols. In turn, this might make designing and developing the switches cheaper.

3. **Faster Innovation.** With SDN [McK+08], the network operators do not have to wait for switch vendors to incorporate new CP protocols into their own devices. They simply can just update their SDN controller with a new network control logic and deploy it instantly without changing the underlying switching (hardware) devices. Thus, SDN enables faster innovation [McK+08].

However, SDN also exhibits a couple of drawbacks:

1. **Single Point of Failure.** Since a logically centralized SDN controller is in charge of controlling the network, it represents a single point of failure. That is if a controller fails, the network operator loses full control of a network, and the management of a DP becomes unavailable.

2. **Scalability.** With every day, the total number of internet users is growing [For+19]. Therefore, the total number of flows in one large network can easily reach over 1 milion. Therefore, having one centralized controller can be a bottleneck for such networks.



(a) Legacy Network

(b) SDN Network

Figure 2.1: (a) Legacy and (b) SDN network architecture.

To overcome the two aforementioned issues, many *state-of-the-art (SotA)* works propose to use a distributed CP instead of a centralized one [KCG14]; [Dix+13]; [Dix+14]. While this indeed resolves *single point of failure* and *scalability* issues, it also raises a couple of novel problems. For instance, if multiple distributed SDN controllers are managing the network, they might have to be synchronized to ensure that the underlying network is updated consistently [SK18]. To enable that, novel consensus algorithms such as Raft [OO14] and Paxos [Lam01] can be deployed in various SDN network controllers [SK17]. Additionally, during network operation, distributed SDN controllers might become buggy, fail, or even malicious. Hence, to protect a network from these issues, a distributed architecture supporting Byzantine Fault Tolerance (BFT) principles [MNR19] can be used. In [SĐK18], it was demonstrated how BFT can be implemented into a distributed SDN CP architecture. Moreover, the aforementioned study also demonstrated the effectiveness of a such solution in various networking settings.

Since the focus of this thesis is Quality of Service (QoS) in programmable networks, the contributions presented later rely on a centralized SDN CP architecture. Therefore, even though *scalability*

and distributed CP are important topics, we consider them orthogonal compared to the contributions of this thesis.

2.1.1.1 OpenFlow

OpenFlow [Fou21] is one of the most famous and widely used protocols for communication between a SDN controller and OF-enabled switches. It is supported by many different switch vendors (e.g., Dell, HP, Pica8, and many others), and it is currently the de facto standard for the realization of SDN networks. OF uses both Transmission Control Protocol (TCP) and Transport Layer Security (TLS) to ensure that the packet is reliably and securely exchanged between a controller and switches.

The procedure to enable a CP OF connection based on version 1.0 between an SDN controller and a OF-enabled switch is as follows.

1. **Configuration Phase.** An SDN controller is run (typically) as an application on a general purpose server in a network. It is configured to listen on one available Internet Protocol (IP) address and one TCP port (default TCP ports for OF are 6633 and 6653). The OF-enabled switches can then initiate the connection towards the configured SDN controller.

2. **Handshake Phase.** After TCP 3-way handshake is completed, the initial handshake procedure of OF starts. The main objective of OF handshake procedure is to exchange the capabilities and supported features between an SDN controller and a switch. For example, during this phase, a controller and a switch agree on which OF version to use.

3. **Operational Phase.** After an OF handshake procedure is finished, the operational phase starts. During this phase, an SDN controller and a switch exchange keep-alive messages to ensure that there is no failure. Additionally, during this phase, an SDN controller can manage the switch as desired by following the OF specification. For instance, it can insert *flow* rules into the *flow* table of a connected switch to enable packet forwarding in the network.

Flow Table. Since OF is an SDN control protocol, it also follows the *match* and *action* paradigm. In OF 1.0, each *flow* table entry has the following attributes: *(1)* header fields, *(2)* counters, and *(3)* actions. Header fields are used for matching. Counters are used for monitoring and statistic gathering, and they are updated after each packet is processed. Finally, actions are applied accordingly based on the matching outcome. Tab. 2.1 presents the available header fields in OF 1.0 version.

In the following, the most important OF 1.0 messages are listed.

1. *FlowMod Add* (`OFPT_FLOW_MOD` message with command `OFPFC_ADD` in OF 1.0 specification [Spe09]) - This message is sent by an SDN controller towards a OF-enabled switch, and the main purpose of this message is to instruct a switch to add a new flow rule into it's flow table.

2. *FlowMod Delete* (`OFPT_FLOW_MOD` message with command `OFPFC_DELETE` or `OFPFC_DELETE_STRICT` in OF 1.0 specification [Spe09]) - This message is also sent by an SDN controller towards a OF-enabled switch, and the main purpose of this message is to instruct a switch to delete an already inserted flow rule from it's flow table.

Header	OF 1.0
Ingress Port	✓
Ether src.	✓
Ether dst.	✓
Ether type	✓
VLAN id	✓
VLAN priority	✓
IP src.	✓
IP dst.	✓
IP proto.	✓
IP ToS bits	✓
TCP/UDP src. port	✓
TCP/UDP dst. port	✓

Table 2.1: Supported header fields in OF 1.0.

3. *FlowStatsRequest* (`OFPT_STATS_REQUEST` message in OF 1.0 specification [Spe09]) - By sending this message to a OF-enabled switch, an SDN controller requests the information regarding flow statistics. It could request the statistics of all flows, or only a certain subset of flows. In OF, flow statistics are gathered by utilizing counters which are part of every entry in a flow table. There are multiple types of counters, such as packet, byte, and drop counters, and so on.

4. *FlowStatsReply* (`OFPT_STATS_REPLY` message in OF 1.0 specification [Spe09]) - With one or more *FlowStatsReply* message(s), a OF-enabled switch reports back to the controller current values of counters for all of the requested flows.

The first two messages are used to enable packet routing (or forwarding) in an SDN-enabled network, i.e., they are used by an SDN controller to insert and update the flow rules on OF-enabled switches deployed in a network. While the last two messages are mostly used for monitoring the state of a network (e.g., for calculating link utilization). Of course, OF [Fou21] specification or a standard provides many more messages. However, since in this thesis we mostly rely on these four messages, the other ones are not introduced in this section.

Newer OF Versions. Every couple of years a new version of OF is introduced. With each version, new functionalities are added. For instance, OF 1.1 version further extends the supported header fields listed in Tab. 2.1. OF 1.3 introduces traffic metering and multi-controller support (and many other things). For more information we refer readers to OF specifications (e.g., [Fou21]).

SDN Controllers. There are many different SDN controllers which support OF. The main difference between SDN controllers is the offered functionalities, performance, and the used programming language. Some of the well known controllers are Ryu [Com17], OpenDayLight [Med+14],

ONOS [Ber+14], FloodLight [Flo15], and NOX [KSG14]. For more information regarding SDN controller we refer readers to *SotA* studies [Sha+13]; [Kho+14]; [Zhu+20].

For more information about SDN and OF, we refer readers to either the original paper [McK+08] or one of the comprehensive surveys [Row+14]; [BEE16]; [SOS13].

2.1.1.2 Performance of OF-enables switches.

To control reliably the traffic traversing a network, SDN controllers must understand what performance the physical and virtual SDN switches support. For instance, if an SDN controller aims to provide *end-to-end* latency guarantees, it must be aware of the processing time of each deployed SDN switch. Therefore, many works [Van+19a]; [KPK14a]; [KPK15]; [Kuź+18]; [He+15a]; [BR13]; [DBK15]; [Laz+14]; [HYS13]; [Rot+12]; [He+15b]; [Bau+18]; [PMK13]; [Bia+10]; [Emm+14]; [Van+19b]; [HYS13]; [Jar+11]; [Nao+08]; [GYG13]; [Lin+17] in the *SotA* investigate and measure the data and CP performance of the available OF-enabled switches. Based on the literature, the following conclusions regarding the performance of hardware OF switches can be extracted. The following conclusions influenced certain system design choices which are presented later in the thesis.

Data Plane Processing Time. Most of the *carrier-grade* OF switches exhibit very deterministic processing time. For instance, the packet processing time of *Dell S4048-ON*, *Pica8 P3297*, *HP E3800*, and *NEC PF5240* was always within $0 - 6\mu s$ [Van+19a] regardless of the packet size, match type, or action type. Since the processing time is very low, this indicates that the main sources of delay in a network are propagation time and packet queuing at transit switches.

Throughput. Almost all of the *carrier-grade* OF switches support full line rate throughput even if all of their physical ports are fully used [Van+19a]. Additionally, packet size, match type, and action type did not produce any impact on the throughput. Following two OF-enables switches failed to support do the same: *HP E3800* and *HP 2920*. However, the newer version of *HP* device might perform better and potentially offer full line rate.

Traffic Manager and Packet Queuing. Traffic manager is an entity that is in charge of forwarding the packets between ports. Therefore, it implements various queuing strategies (e.g., strict priority queuing [BP02] or weighted fair queuing [BP02]), and it allocates the available buffer space to queues [Van+19a] based on the configured strategy. Additionally, this entity is also in charge of traffic replication. Traffic manager is not part of OF protocol and standard, however, all OF-enabled switches to have to implement it. Therefore, it is important to understand what are the offered functionalities of traffic manager and the corresponding performance. In [Van+19a], the authors analyzed the features offered by traffic managers deployed in 5 different OF switches and they tried to evaluate if their behavior is predictable. The overall conclusion was that the provided features vary from switch to switch. However, the observed performance was predictable. For example, for a given scenario, it is possible to measure and estimate how much buffer space will be allocated to a certain priority queue [Van+19a].

Traffic Policing. In OF, traffic policing is realized with the *metering* feature [McK+08]. Metering was introduced as part of the OF v1.3 [McK+08] specifications and it is supported by most of the *carrier-grade* switches. However, surprisingly, the performance of traffic policing in OF-enabled

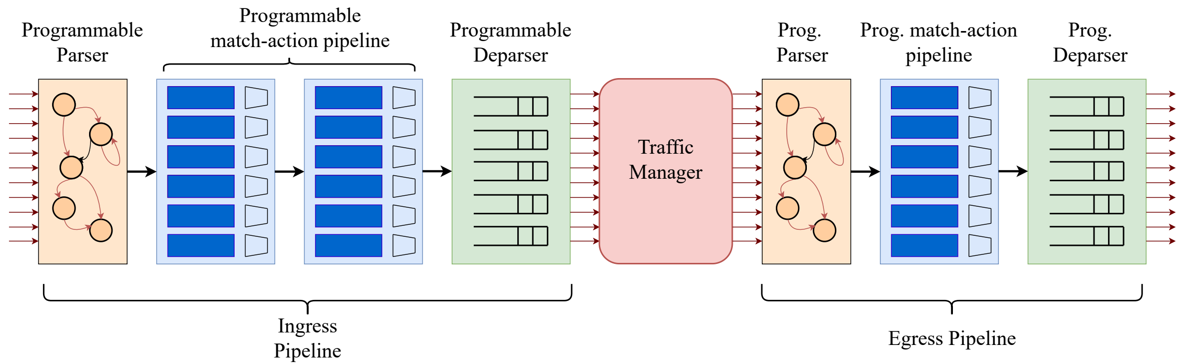


Figure 2.2: P4 Processing Pipeline.

switches was never comprehensively evaluated. Therefore, one of the main contributions presented in Chapter 4 is the corresponding comprehensive measurement results.

Flow Management. Many works in the literature demonstrated that inserting the rules into a *flow* table of a *carrier-grade* switch offers unpredictable and unexpected performance [Van+19a]; [KPK14a]; [KPK15]; [Kuź+18]. For instance, in [Van+19a], it was shown that inserting the flow rules can be surprisingly slow – the time needed to insert 1000 flow rules into a flow table of *NEC PF5240* took over 16 minutes. Additionally, in some scenarios, some switches even ignore adding the flow rules sent by an SDN controller [Van+19a]; [Kuź+18]. Therefore, to avoid reconfiguring often SDN switches, some works proposed to use source routing [Van+20]. With the same goal (avoiding often reconfigurations), the presented SDN-based system and the main contribution of Chapter 5 routes the packets based on the decision made by edge-switches.

2.1.2 Programming Protocol-independent Packet Processors (P4)

Even though SDN architecture and OF protocol enable network operators to have CP programmability, at that time, there was still one glaring problem. That is, each OF version had a fixed set of functionalities. For instance, in OF 1.0, it is possible only to match 13 different header field values. Of course, new OF versions always extended the set of functionalities (e.g., header field list). However, developing a new version always took a couple of years, and updating the software (or agent) of the hardware switches deployed in a network also took at least a couple of months. Therefore, to increase flexibility and speed up the innovation, P4 [Bos+14] programming language was developed.

The main objective of P4 is to provide DP programmability [Bos+14]. With P4 it becomes possible to program how the switches (or SmartNICs) process the packets in the DP. It is based on *C* programming language, thus, enabling network operators to quickly and easily develop new applications. Fig. 2.2 illustrates the architecture with the corresponding stages of one P4 processing pipeline. This pipeline represents the logical pipeline of one P4-enabled (hardware) switch or Network Card Interface (NIC), thus, packets enter it on the left-hand side, and sequentially traverse the depicted stages before being forwarded further (alternatively they could also be intentionally or unintentionally dropped).

In the following, P4 stages are discussed and presented in more detail [Bos+14].

Programmable Parser. Upon packet reception, most of the switches first extract headers from a packet. For instance, SDN-enabled switch might extract the header fields listed in Tab. 2.1 (e.g., Virtual Local Area Network (VLAN) id or IP src.). In a programmable parser of P4 pipeline, it is possible to define arbitrary headers and specify how they should be extracted from a (received) packet that just started traversing the pipeline. This allows us to design new headers, and deploy rapidly novel protocols. The parsed headers and packet metadata are later on forwarded to the programmable match-action pipeline. Packet metadata usually contains additional information regarding packet processing (e.g., ingress port of the packet, or timestamp representing packet reception). Additionally, usually, metadata also contains information about how the packet should be processed further (e.g., it usually contains a flag to indicate if a packet should be dropped). Packet metadata is usually forwarded across multiple stages.

Programmable match-action pipeline. In this part of the pipeline, similarly as in SDN, packets are matched (based on the extracted arbitrary defined headers) to flows, and actions are applied to the traversing packets. With P4 it is possible to program different matching strategies (e.g., we could match on already arbitrary defined headers), and different actions. Additionally, there could be multiple stages in a programmable match-action pipeline.

Programmable Deparser. The main goal of the code located in the programmable deparser is to define how the packet is later on assembled from the extracted headers and packet metadata. Therefore, based on the defined logic, for instance, not all headers have to be appended to the packet during the assembly in deparser. The previous three stages together are called an ingress pipeline.

Traffic Manager. As explained in Sec. 2.1.1.1, the main goal of the traffic manager entity is to forward the packets between ports. Since traffic manager is not included in the current version of P4, it is not possible to program its behavior of it with P4.

Ingress and Egress Pipeline. After the packet traverses a traffic manager, it gets processed by an egress pipeline. Conceptually, an egress pipeline performs the same tasks as ingress pipeline. The main purpose of as an egress pipeline is to provide additional options for the programming of the DP, and it is can be potentially useful in certain scenarios.

Supported devices. During the last couple of years, many vendors such as Intel, Netronome, Xilinx have developed hardware devices (including both switches and SmartNICs) that support P4 programming language. Therefore, network and data center operators are now able to use P4-enabled devices in their networks and reap the benefits provided by P4 programming language.

2.2 Virtualization

In this section, initially, a short background regarding server virtualization, containerization, and NFV is provided (see Sec. 2.2.1 and Sec. 2.2.2). Afterward, background regarding network virtualization in SDN networks is presented.

2.2.1 Server Virtualization and Containerization Approaches

In computing, virtualization enables the creation of virtual resources from the underlying physical resources. Therefore, it is often said that virtualization abstracts physical resources and provides a

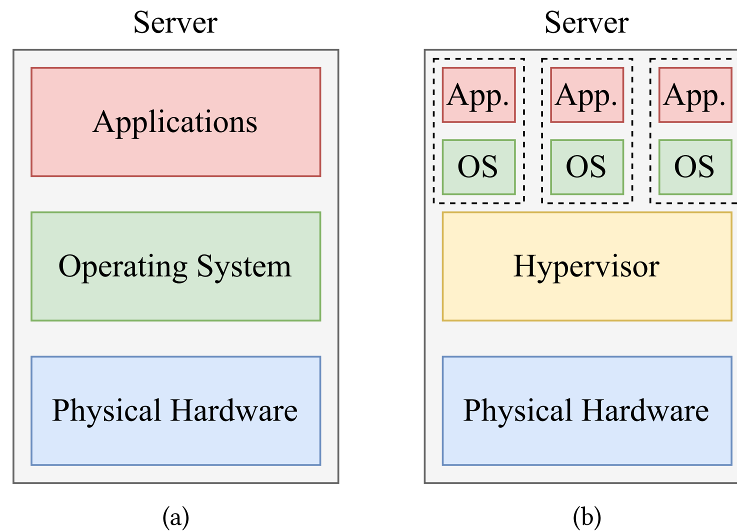


Figure 2.3: (a) Non-virtualized and (b) Virtualized server. Each black dashed rectangleon the right figure represents one VMs.

virtual representation of them. Computing resources such as computer hardware platforms, storage devices, and computer network resources can be virtualized.

Server virtualization (or hardware/platform virtualization) is a process of creating and running a Virtual Machine (VM) that acts like a real computer with an operating system. Server virtualization is enabled by a hypervisor. It can be implemented in software, hardware, or as a hybrid solution. The hypervisor manages the hardware resources (e.g., Central Processing Unit (CPU), Random Access Memory (RAM), input/output (I/O) stack, and more) and allocates them to VMs when needed. An example of a non-virtualized and virtualized server is presented in Fig. 2.3.

There are two types of hypervisors.

1. **Type 1.** Type 1 hypervisor (or a native or bare metal hypervisor) is a hypervisor that runs directly on the server's hardware (or on the host), and it controls the hardware to manage VMs (or guest operating systems). Examples of Type 1 hypervisors: KVM (part of Linux kernel since 2007), Microsoft Hyper-V, and VMware vSphere.

2. **Type 2.** Type 2 hypervisors run as a normal application or program on a generic operating system. They abstract the guest operating systems from the underlying operating system running on the host server. Examples of Type 2 hypervisors: VMware Workstation and Oracle VirtualBox.

To reduce the possible overheads introduced by running a hypervisor, containerization approaches were developed. Containerization is a type of lightweight virtualization. In it, applications are deployed in isolated user spaces, called containers, and they utilize the same shared operating system. Containers contain and encapsulate everything an application might potentially need to run. For instance, it can contain required libraries, configuration files, source code binaries, and all dependencies. To enable containerization on commodity server, typically, one of the following container runtime tools [Esp+20] is used: *containerd*, *CRI-O*, *Docker Engine*, and many more.

There are many benefits of using either, classical server virtualization or containerization. To start with, it is possible to consolidate services (e.g., apps) in a centralized location (e.g., data center). Additionally, it is possible to scale up or down the number of deployed services (e.g., apps) based on the current requirements and load in a highly portable manner. Therefore, this enables network operators and service providers to reduce their operational costs. Moreover, these approaches allow application or service programmers to separate *monolithic* applications into *microservices*. That is, in *microservice* architecture, the full application is a collection of smaller, and independently deployable services. This service architecture provides some benefits such as easier updating of services, Continuous Integration/Continuous Developement (CI/CD), and many more.

In this thesis, server virtualization and containerization are often used when designing systems to utilize their benefits. For instance, in Chapter 3.2, network hypervisors are often deployed within a VM. Additionally, the presented traffic policing Data Plane Development Kit (DPDK) application in Chapter 4 is always run within a container. This is done to increase portability of the evaluated and developed applications. For example, running the presented DPDK application on any other server which supports containerization becomes trivial.

2.2.2 NFV Architecture

NFV proposes a new way of managing and deploying network functions [Bas+14]. The network functions (e.g. firewall, Network Address Translation (NAT), load balancer) are decoupled from the dedicated hardware and realized as software instances, i.e., Virtual Network Function (VNF)s running on commodity servers. The main enabler of NFV is server virtualization and containerization (see Sec. 2.2.1), The benefits of NFV are manifold [Han+15]. Firstly, consolidating services and by using Commercial Off-The-Shelf (COTS) hardware instead of special purpose equipment can lead to reducing network over-provisioning. Secondly, the dynamic and automated orchestration of VNFs (e.g., VNF instantiation or migration) based on the current service requirements is possible through cloud systems (e.g., Kubernetes, OpenShift, AWS). Hence, in this case, the experienced QoS can be improved.

Additionally, in *SotA*, the NFV architecture are even extended further to incorporate programmable P4 devices [He+18].

Unfortunately, the benefits can come with a price: the observed end-to-end delay might increase due to the function softwarization [Li+17]; [Li+13] (because of additional server virtualization layers) or due to placing functions at server locations that promise low resource costs but are far away from users [Var+19]. Besides, running multiple VNFs on the same server with a limited amount of resources can lead to the resource over-utilization and degraded performance of certain VNFs. This drawback is not acceptable for network functions with high QoS requirements. Therefore, understanding how many resources one VNF needs is crucial. To solve this problem, in Chapter 3.2 a novel measurement-based procedure for provisioning of one VNF is presented.

2.2.3 Virtualization of Programmable SDN Networks

Virtualization of SDN networks enables the Network as a Service (NaaS) model. Tenants can request network resources (e.g., a custom virtual topology with bandwidth requirements) from a network

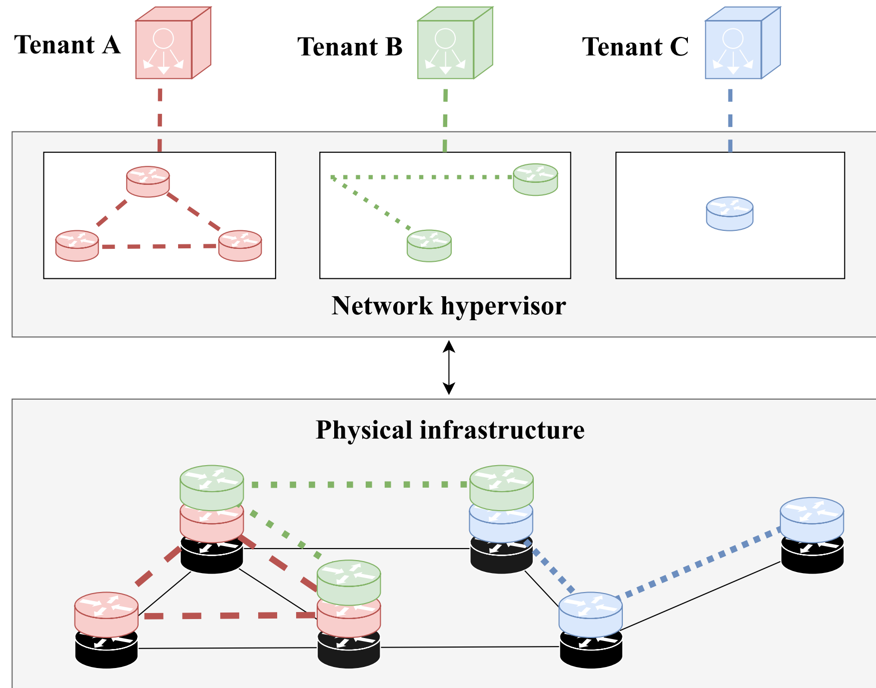


Figure 2.4: Overview of network virtualization. A network hypervisor acts as a proxy between the tenants controllers and the physical infrastructure. A hypervisor can provide different virtualization policies to different tenants, i.e., in this case different levels of topology abstraction.

provider. Through network virtualization, the provider allows different tenants to share a common physical infrastructure. The concept is illustrated in Fig. 2.4. Each tenant is given a Virtual Network (VN), i.e., a set of virtual (interconnected) switches and hosts, to which it can connect its custom SDN controller. Furthermore, in the DP, tenants are differentiated by their *flowspace*, where a *flowspace* is a subset of all possible OF [McK+08] matching fields. If the tenants *flowspaces* are not overlapping, all the DP traffic can be easily mapped to the corresponding tenant. For instance, we can define a *flowspace* of a tenant A (see Fig. 2.4) as the *10.0.0.1/24* subnet with full port access on physical switches highlighted in red color (they are interconnected with dashed lines). Through their controller, tenants can then fully control the DP forwarding behavior in their VN and steer the traffic as they desire within their VN. To provide such a service, the provider uses a Network Hypervisor (NH) which acts as a proxy between the controllers of the tenants and the physical infrastructure. NHs are usually realized as software applications (e.g., a Java program) running on generic commodity computation servers. The most famous NHs are *FlowVisor* [She+09] (written in Java) and *OpenVirtex* [Al-+14] (written in Java).

To enable the existence of such an architecture, network hypervisors typically provide some of the following functionalities [Ble+16a].

1. **Monitoring the Network.** At all times, network hypervisors should maintain the state of the network by either tracking all of the embedded VNs and their requirements in the DP and CP and/or they have to monitor all of the switches in the underlying physical networks. This information can be later used to determine if a new VN can be added or not.

2. **Running Virtual Network Embedding (VNE) Algorithm.** When a tenant requests a new VN, a VNE algorithm within NH is responsible for checking if there is a sufficient amount of resources to embed this request. Additionally, it is also in charge of mapping the requested VN to the physical infrastructure [CRB09]; [Fis+13].

3. **Establishing Isolation Policies.** After a VN request has been accepted by a VNE algorithm, an NH has to reserve the required data and CP resources for the request [Van+19b]; [She+09]. For example, in the DP, an NH could reserve the resources by inserting traffic policing rules (based on the required bandwidth) on the corresponding underlying physical switches. The main goal of establishing isolation policies is to prevent tenants from over-utilizing the physical resources and causing performance degradation to the other tenants (e.g., increased *end-to-end* delay).

4. **Establish Abstraction Policies.** NHs could present the physical networking resources to the tenants' SDN controllers as simplified virtual resources. All types of resources (e.g., link, node, topology) can be simplified (abstracted). By abstracting the physical resources, NHs can simplify to tenants the management of their own VNs. To provide this functionality, NHs have to keep internally additional abstraction mappings.

5. **Message Translation and Inspection.** NHs have to during the runtime translate and inspect all of the CP messages exchanged between the tenants SDN controller and the physical infrastructure. The inspection of messages is performed to ensure that the tenants do not configure or use the resources reserved for a different tenant.

Naturally, not all NHs implement all the previously listed functionalities, and some even provide additional ones. Moreover, NHs often vary in terms of features and they often implement these functionalities in a different manner. For a more detailed overview of network virtualization in SDN, the readers are referred to the following survey [Ble+16a].

In the following section, certain implementation details of the most well known NH, i.e., *FlowVisor*, are presented. This information is later on used in Chapter 3.2 to design a measurement-based methodology for provisioning the hardware resources available to a *FlowVisor* NH.

2.2.3.1 Message Inspection/Processing in FlowVisor

FlowVisor [She+09] implements *partial topology abstraction* and *full port abstraction. Partial topology abstraction* can be defined in the following way: besides the requested end-point physical switches, the intermediate switches on the paths between the end-point switches are also shown to the tenants' controllers, while all other physical switches are hidden. This corresponds directly to the red dashed VN in Fig. 2.4 (i.e., tenant A), where the physical switches are mapped with *one-to-one* configuration to the virtual switches. *Full port abstraction* means that only the physical ports containing the tenants' hosts and interconnecting the VN are shown, while all other existing physical ports are hidden or abstracted away. To achieve such functionalities, each CP message has to be processed by the NH. In OF [McK+08], the forwarding behavior of a switch is modified with *FlowMod Add* message[1],

[1] In OF 1.0 specification [Spe09] *FlowMod Add* is called `OFPT_FLOW_MOD`.

hence, the corresponding message is one of the most complex to inspect. For instance, match and action fields contained in every *FlowMod Add* message have to be checked concerning the agreed virtualization policies. As we focus on provisioning the resources for the flow embedding task, in the following we explain the *FlowVisor's* processing pipeline of *FlowMod Add* in more detail.

Upon a reception of *FlowMod Add* message, *FlowVisor* firstly checks the contained action set. For instance, the corresponding message might be trying to add a rule that forwards the traffic to a port which is not part of the tenant's *flowspace*. Thus, we suspect that the number of virtual ports could have an impact on the processing workload of *FlowVisor*. If the contained action set is not violating the agreed policies, *FlowVisor* then inspects the match. This is done by intersecting the match with the *flowspace* and evaluating if the tenant has permission to match on those fields. If a tenant is using additional field (not included in its *flowspace*), *FlowVisor* rewrites them. Similarly, we suspect that type of matching can have an impact. Thus, if we use port-based matching (common to L2 and L3 forwarding applications), the required resources might scale again with the number of virtual ports. Finally, the message is forwarded to the targeted physical switch. Moreover, the size of the topology could have an impact on the total processing workload, as it can affect the lookup time for determining the correct physical switch destination. Furthermore, the topology size and edge density directly impact the total number of ports in the network, thus, potentially affecting the required workload for inspecting the match and action fields of *FlowMod Add*. Although *FlowVisor* implements most of the look-ups with hashmap (scales with $O(1)$), some look-ups use linked lists (scales with $O(n)$), thus the lookup and workload could be affected by the aforementioned parameters.

2.3 Deterministic Performance in SDN Networks

In this section, to begin with, the general notion of providing deterministic guarantees in the DP of a network is introduced along with an overview of the few systems which provide it (see Sec. 2.3.1). Afterwards, the most promising *SotA* system, i.e., *Chameleon* [Van+20], which provides such guarantees is introduced in detail (see Sec. 2.3.3).

2.3.1 General Notion of Determinism in Networking and Literature Survey

Deterministic networking focuses on providing deterministic (or non-stochastic) bounds on packet loss, packet *end-to-end* latency, and high reliability [COS17]; [VK16]; [LT01] in the DP. There are two main groups of networking solutions which aim to provide such guarantees.

The first group of solutions is mostly used in industrial networking and it includes IETF's Deterministic Networking (DetNet) Working Group, many networking standards (e.g., Time-Sensitive Networking (TSN) developed by IEEE 802.1 TSN Task Group [COS17]), and various proprietary solutions (e.g.„ PROFINET [FFV06], EtherCAT [JB04]) which aim to provide such guarantees. However, even though some of the works are standardized, often proprietary hardware devices are needed to realize such networks. Additionally, these devices often have limited amount of ports (e.g., around 4), and they often support only low link rates (e.g., $100M$, or $1G$) which is common for industrial networks. Therefore, these solutions are not applicable in networks with higher bandwidth requirements such as DC or Wide Area Network (WAN) networks (where link rates are either $10G$ or $100G$ or more).

The second group relies on advanced mathematical frameworks such as Deterministic Network Calculus (DNC) [LT01] to design networking systems [Van+20]; [Jan+15]; [Gro+15]; [Van+19b] which can provide such strict DP guarantees. These systems often can be realized with generic switching hardware such as OF-enabled devices. Thus, they support much higher rates and can be used in various scenarios. For example, in [Van+19b], the authors demonstrated that by utilizing DNC mathematical framework it is possible to even provide deterministic guarantees in small networks with low-cost OF-enabled switches. That is, the deployed deterministic network consisted only of 4-port Zodiac FX switches which cost around 100$. Therefore, DNC-based deterministic solutions maybe even used in networks that resemble industrial scenarios (e.g., network within a robot or an airplane).

Since the goal of this thesis is to improve and develop new concepts which provide QoS guarantees in virtualized programmable networks (e.g., SDN and/or P4) in the following we only focus on the second group of deterministic networking solutions.

2.3.2 DNC Theory

DNC is a mathematical framework for deriving and analyzing worst-case performance bounds in computer networks. It was firstly introduced by R. Cruz in [Cru91a]; [Cru91b] and later extended in [LT01]. In the following, a short overview of the DNC is provided, for more information, we refer the readers to [Cru91a]; [Cru91b]; [LT01].

To provide such worst-case bounds, the following network characteristics have to be modeled and provided as input parameters to DNC.

Flow Modeling. First of all, all the flows in the network have to be modeled. Typically, a flow in a packet-based network is described as a unidirectional sequence of packets sent by a single sender to a single receiver. In DNC, a flow is modeled with an *arrival curve*. The example of an *arrival curve* is shown in Fig. 2.5. The vertical part (overlapping with the data axis) of the *arrival curve* represents the maximal allowed burst size of a flow (i.e., b_a), while the diagonal part represents the rate of a flow (i.e., r_a). In practical systems [Van+20]; [Jan+15], usually, the generated flows are limited with either a traffic policing or shaping entity to ensure that they are not exceeding the pre-defined *arrival curve*.

Service Modeling. Similarly to flows, all the forwarding devices in a network have to be modeled as well. In DNC, they are modeled with a *service curve*. The example of a *service curve* is shown in Fig. 2.5. The horizontal part (overlapping with the time axis) of the *service curve* represents the processing time of a device (i.e., p_s), while the diagonal part represents the offered throughput (i.e., r_s). Many works in the literature measure the forwarding devices [Van+19b]; [Hel+21] with a goal of obtaining or generating the corresponding *service curves*.

After all the flows and forwarding devices are represented with the corresponding *arrival curves* and *service curves*, DNC can quantify worst-case performance bounds by utilizing the *min-plus* and *max-plus* algebra. To be precise, DNC derives the worst-case bounds such as worst-case *end-to-end* delay and the maximal amount of backlog in the system. The maximal amount of backlog refers to the maximal amount of data held inside the system. Therefore, it corresponds to the needed buffer space for the temporary storing of the traversing data.

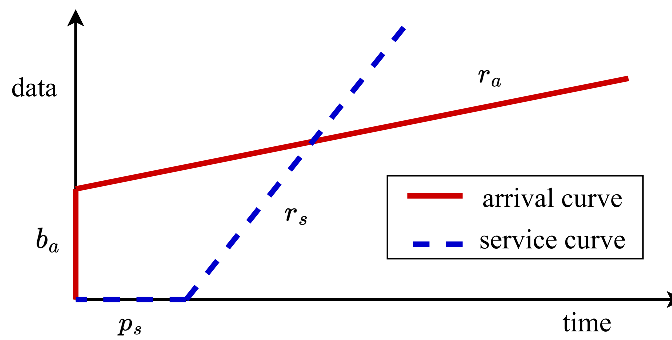


Figure 2.5: Example of an *arrival curve* (red dashed line) and a *service curve* (blue dotted line).

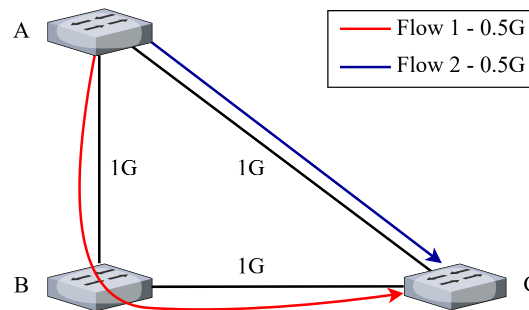


Figure 2.6: Example of a small network with 1G links and two flows, where each flow utilizes 0.5G. Embedding a new flow request of 1G between switch A and B is not possible without reconfiguration.

2.3.3 Chameleon System

In this part, initially, it is discussed why *Chameleon* system performs better (i.e., achieves higher utilization) compared to the other *state-of-the-art* approaches. Afterward, an overview of *Chameleon* system is presented. *Chameleon* system is already published in detail in one conference paper [Van+20] and in one PhD thesis [Bem+20].

2.3.3.1 Why does Chameleon perform better?

There are two major reasons why *Chameleon* system performs better compared to the other *SotA* approaches such as Qjump [Gro+15] and Silo [Jan+15]. They are explained in the following.

1. Flow Reconfigurations. Both Qjump and Silo, do not reconfigure the flows after they have been embedded into a network. In contrast, if there are not enough resources to embed (or accept) a new flow request, *Chameleon* tries to optimize the already embedded flow in the network to create enough space for a new flow. By doing so, network utilization could be increased. Consider the following example in Fig. 2.6. If the link speed is $1G$, and there is a new flow request (e.g., *Flow 3*) with the required bandwidth of $1G$ between switches A and B, both Qjump and Silo will not be able to satisfy this request. In contrast, *Chameleon* will try to reroute one of the already embedded flows (e.g., *Flow 1* or *Flow 2*) in order to create enough space for the new flow request.

2. Queue Level-Topology. Typically, most of the *carrier-grade* switches to support multiple priority queues per one physical port. In most cases, they have either 4 or 8 queues [Van+19a]. Having more queues enables more fine-grain prioritization of flows and supports a higher range of

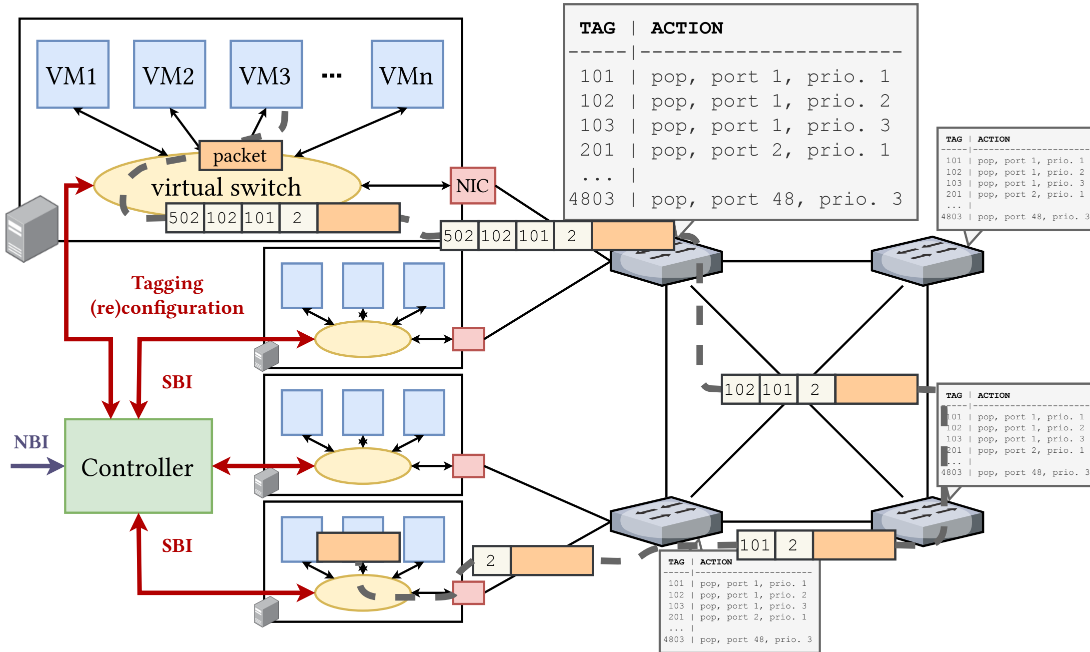


Figure 2.7: Chameleon architecture. Taken from [Van+20].

applications with vastly different delay requirements [GVK17]. Both Qjump and Silo do not utilize all the available resources on switching devices, for instance, Qjump and Silo use only one or two priority queue(s) per port. On the other hand, *Chameleon* utilizes all of the available resources on switching devices, such as the number of queues and available buffer space.

2.3.3.2 Chameleon Architecture

Fig. 2.7 presents an overview of *Chameleon* system architecture. Firstly, *Chameleon* system aims to provide deterministic guarantees in the DP of a cloud network (i.e., in a DC network). Therefore, it is assumed that the end hosts (i.e., servers) can be controlled if desired. They are virtualized through QEMU 2.11.1 with KVM.

Centralized Control. *Chameleon* uses an SDN architecture, where the whole network is controller by a centralized controller. This controller listens on the Northbound Interface (NBI) for flow requests. A flow request is defined by its source and destination server nodes, required rate, required burst, and delay deadline. Upon flow request reception, *Chameleon* firstly tries to find a route for the flow by utilizing LARAC [Jut+01] algorithm. Afterward, DNC [LT01]; [Cru91a]; [Cru91b] mathematical framework is used to ensure the embedding of the new flow does not violate the guarantees of the other already embedded flows. Moreover, if the outcome of DNC framework is negative, *Chameleon* tries to reroute the already embedded flows to free up the resources. To do so, the proposed greedy reconfiguration algorithm is used. After each reconfiguration attempt, the previous procedure (flow routing and DNC bound checking) is repeated.

Source Routing. Since *Chameleon* relies on reconfiguration, it is important that the physical hardware switches can be reconfigured in a deterministic manner. Unfortunately, that is not the case for certain of SDN switches [Van+19a]; [KPK14b]; [Kuź+18]. For instance, it was shown that

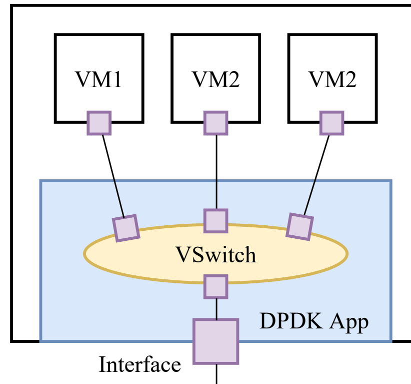


Figure 2.8: Architecture of DPDK application running on end-host.

certain switches sometimes ignore adding rules sent by an SDN controller. Therefore, *Chameleon* system utilizes *source routing*. In *source routing*, the route of a packet in the network is determined by a source host, and it is usually encoded within the packet header. Hosts in *Chameleon* append VLAN tags which carry the information about a route of a packet. To enable source routing in the network, the switches are configured once at the network start-up with the predefined set of VLAN tags. Fig. 2.7 illustrates how one packet can be routed in a network with source routing. In this case, a source host appends 4 tags (i.e., with VLAN IDs 502, 102, 101, and 2). The first tag determines on which port and priority queue should the first switch forward the packet. Based on a VLAN tag with an ID of 502, the switch can deduct that the packet should be forwarded on port 5 and priority queue 2. The second tag determines the forwarding on the second switch, and so on.

DPDK Application. To realize source routing on end hosts (or server) in a DC, a DPDK application based on Intel's VMDq application was developed. The logical representation of the DPDK application is presented in Fig. 2.8. The main objective of the developed application is to connect VMs of users with the physical network in a deterministic manner, hence, it behaves as an enhanced virtual switch. The VMs belonging to users are connected to the developed DPDK application via *virtio* using a *vhost-net/virtio-net* para-virtualization [Ada+15]. Additionally, each VM is assigned one Rx/Tx queue pair. The DPDK application utilizes three separate cores: one for receiving packets (i.e., Rx part), one for sending packets (i.e., Tx part), and one for the control of the DPDK application.

Rx Part. In the receiving part, the DPDK application uses Virtual Machine Device Queues (VMDQ) technology available on Intel's NIC to sort the incoming packets into the physical Rx queue dedicated to the corresponding VMs. The (software) Rx part of the DPDK application (running in an infinite loop) then pulls a batch of packets and forwards them to the corresponding VMs based on the MAC destination address and VLAN tag. Additionally, in the receiving part, there is no need for any other additional functionality.

Tx Part. In the sending part, the DPDK application runs an infinite loop. In it, it pulls packets (in a batch) from queues connecting the VMs in a round-robin manner. After pulling the packets, the DPDK application adds the corresponding VLAN tags and it polices the traffic based on the configured parameters. This is done in order to ensure that the VMs do not exceed the agreed *arrival curves* (see Sec. 2.3.2)

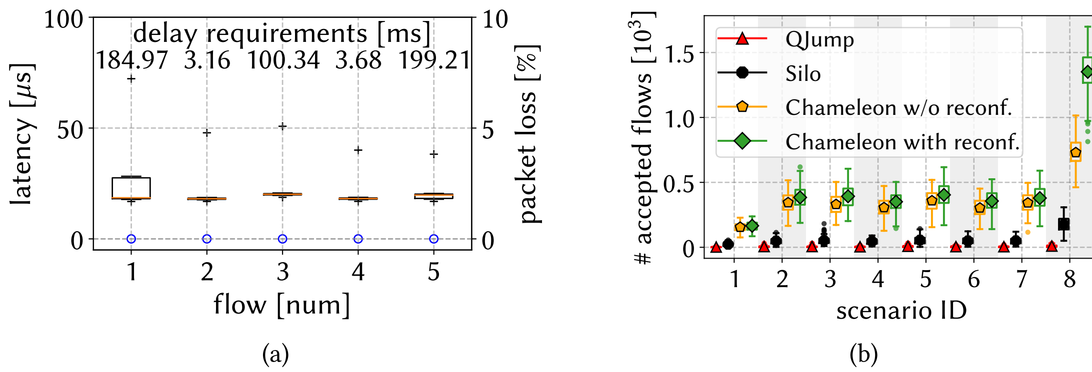


Figure 2.9: (a) Measured packet latency and loss of 5 flows in the deployed network managed by *Chameleon*. (b) Number of accepted flows in 8 different scenarios by *Chameleon*,*Qjump* and *Silo*. Source: [Van+20].

Ensuring Predictability. To ensure that the DPDK application processes packets in a predictable manner, the following configurations were applied on each server before running the application.

- Hyperthreading and Turbo-Boost were disabled. Additionally, all power-saving features were disabled, and the running frequency of each CPU was set to the (maximal) base value. For example, on Intel Xeon E5-2650 v4 CPU, the running frequency was set to 2.2 GHz for each of the 24 cores.

- The three parts of DPDK application (e.g., Rx, Tx, and control part) were running on isolated cores. The isolation was achieved by setting kernel *isolcpus* parameter for certain 3 cores and by pinning the DPDK application to those three cores.

- To isolate the available cache to the DPDK application, Intel's Cache Allocation Technology (CAT) was leveraged on each server.

By deploying *Chameleon* in a small data center network, it was possible to demonstrate that *Chameleon* system indeed can provide deterministic guarantees in the DP and higher utilization compared to the *SotA*. Fig. 2.9a shows packet latencies and loss of 5 different flows along with the required delay during one deployment measurement run. It can be seen that for all of the evaluated flows, none of the packets were lost, and the achieved delay was always lower compared to the requested one. Fig. 2.9b illustrates the number of accepted flows in various scenarios of *Chameleon*, *Qjump*, and *Silo*. Compared to *Qjump* and *Silo*, *Chameleon* was able in every scenario to accept more flows.

2.4 Summary

In this section, an overview of the relevant technologies and network architectures were presented. Since this thesis aims to enhance technologies needed for realizing QoS-aware virtualization of programmable networks, initially, the most promising programmable technologies such as SDN (including OF protocol) and P4 are presented along with their benefits. Throughout this thesis, many

concepts utilize or rely on the features of programmable networks, hence, understanding them may be beneficial. For example, `NAGA` system presented in Chapter 5 relies on SDN network architecture, while Chapter 4 utilizes OF protocol to configure the switches. Afterward, the technologies such as server virtualization, network virtualization, and NFV are introduced. In particular, one of the most promising solutions offering SDN network virtualization, i.e., *FlowVisor*, is described in detail. Understanding SDN network virtualization concepts and implementation details of *FlowVisor* is crucial for understanding the contributions presented in Chapter 3, which deals with provisioning the CP of virtualized SDN networks. Subsequently, an overview of the *SotA* solutions (including the necessary background) for realizing deterministic DP guarantees is presented as well. Moreover, an extensive overview of the best performing system, i.e., *Chameleon*, is presented. The requirements and shortcomings of these concepts motivated development of the contributions presented in Chapter 4. That is, certain drawbacks of the best performing *SotA* system (i.e., *Chameleon*) are highlighted and resolved in Chapter 5.

Chapter 3

QoS-Aware Network Hypervisor Resource Provisioning

The realization of network virtualization in Software-Defined Networking (SDN) networks is achieved by deploying a softwarized Network Hypervisor (NH) [BBK15]; [Al-+14]; [Han+18] between the tenant's SDN controllers and the physical Data Plane (DP) (as explained in Sec. 2). The introduction of such an entity generated many novel challenges [Đer+17]; [Ble+16a], especially related to the Control Plane (CP) (e.g., network management and orchestration). Therefore, this chapter presents solutions for some of the novel problems.

First of all, to enable network virtualization, NHs perform a certain set of virtualization functions. That is, as classified in [Ble+16a] and explained in background section (see Sec. 2), they have to: *1)* isolate DP and CP traffic of all the tenants, *2)* translate all the messages exhanged by the tenant's SDN controllers and the physical DP, and *3)* abstract the networking resources (if required by the tenants). As suggested by the *state-of-the-art* (*SotA*) works [Ble+16a]; [Sie+16]; [Al-+14], these functions can potentially have an impact on the resource utilization of NHs, e.g., Central Processing Unit (CPU) or Random Access Memory (RAM). If this is the case, it is crucial to consider the impact of such functions when designing network management algorithms (e.g., Virtual Network Embedding (VNE) problem), or the output of the corresponding algorithms might be sub-optimal. Hence, this observation motivated the first contribution presented in this chapter (see Section 3.1.2). That is, to evaluate if such functions indeed have a significant impact, initially the results of one measurement campaign are presented. The goal of this campaign was to analyse the impact of different topology abstraction policies on the resources utilization (i.e., CPU) of an NH. The presented measurement study indicates that indeed the impact of topology abstraction on the CPU utilization is significant, i.e., depending on the used abstraction policy the difference can reach values up to 400%. Furthermore, even though the difference is significant, the results are predictable, thus, three novel CPU estimation models are presented in Section 3.1.3.

The observed big variance in CPU utilization (e.g., from 10% – 150%, where 100% corresponds to fully utilized 1 CPU thread) during our measurement study motivated the second contribution of this chapter (see Section 3.1.4). In this part, by utilizing the developed simulation tool and the novel CPU estimation models, the impact of topology abstraction on an offline VNE problem is studied. In contrast, most of the *SotA* VNE solutions [Ble+16c]; [CRB12]; [CRB09] do no consider neither NH

nor its functions in their VNE problem formulation. Therefore, in this part, it is studied what is the performance penalty of a such design choice. The simulation results revealed that different topology abstraction policies can have a serious impact on the performance of the presented VNE algorithm. For instance, considering topology abstraction as part of a Virtual Network Request (VNR) can improve the total number of embedded Virtual Network (VN) for 30% in certain scenarios. Therefore, both NH resources and it's functions should be considered when designing VNE algorithms.

The previously introduced contributions suggest that the NH-specific functions and DP networks can have a significant influence on the NH CPU utilization and the performance of network optimization algorithms (e.g., VNE problem). Thus, in Sec. 3.2 a novel Quality of Service (QoS)-aware measurement-based NH CPU provisioning approach is presented (the main contribution of this chapter). The objective of this approach is to determine how much CPU resources should be allocated to an NH while avoiding performance degradation (e.g., impact on processing latency). To do so, initially, the CPU utilization of one *SotA* NH is measured and studied in various scenarios while considering a grid topology. Afterward, based on the presented measurements, an accurate CPU estimation model is developed. In the *SotA*, there are a few works [Sie+16]; [SOK17] focused on estimating the mean NH CPU utilization in various scenarios. However, they are not applicable for provisioning, as provisioning with a mean CPU value introduces a big performance penalty (i.e., increase in processing time). In contrast, the proposed estimation model is focused on estimating the maximal amount of needed CPU resources while avoiding performance degradation. Finally, by using the proposed model on a deployed *SotA* NH and an emulated DP network, it is demonstrated that the presented model can be used for provisioning the resources of an NH even in scenarios consisting of unseen (DP) network topologies (e.g., Internet2, NobelEU, etc.) while keeping the performance degradation to a minimum. Therefore, by utilizing the presented QoS-aware provisioning procedure, network operators can maximize their overall resource utilization while incurring negligible performance degradation.

Moreover, in Section 3.3, a demonstration of the benefits of VN reconfiguration and Virtual Machine (VM) migration on the performance of a remote control application is presented.

The majority of the contributions presented in this chapter are already published in three scientific papers. To be precise:

- The contributions presented in Section 3.1 are published in the following conference paper: N. Đerić et al. "SDN hypervisors: How much does topology abstraction matter?" In: *2018 14th International Conference on Network and Service Management (CNSM)*. IEEE. 2018, pp. 328–332. Moreover, this section also presents some unpublished results. Namely, Section 3.1.4 presents additional analysis regarding the VNE problem.

- The contributions presented in Section 3.2 are published in the following journal paper: N. Đerić et al. "Enabling SDN Hypervisor Provisioning through Accurate CPU Utilization Prediction." In: *IEEE Transactions on Network and Service Management* (2021).

- The practical demonstration presented in Section 3.3 is published as part of the following conference paper: N. Đerić et al. "Coupling VNF orchestration and SDN virtual network reconfig-

uration." In: *2019 International Conference on Networked Systems (NetSys)*. IEEE. 2019, pp. 1–3.

The structure of this chapter is as follows. Section 3.1 presents the impact of topology abstraction on NH CPU utilization and an offline VNE problem. Section 3.2 presents the QoS-aware provisioning approach. Furthermore, the demonstration is presented in Section 3.3. Section 3.4 concludes the chapter, and Section 3.5 outlines the possible future research directions.

3.1 Function-Aware Virtual Network Embedding Problem

This section presents a realistic VN embedding problem which considers the impact of the topology abstraction function on resource utilization. To develop such an algorithm, it is necessary to understand if different topology abstraction policies have an impact on the resource utilization of an NH (e.g., CPU). Therefore, in the first part of this section, a comprehensive measurement campaign of one *SotA* NH, i.e., OpenVirtex (OVX) is conducted. The goal of the campaign is to investigate if two VNs with the same DP requirements (e.g., topology, flow rate) and different topology abstraction policies utilize the same amount of CP resources (i.e., CPU utilization of an NH) or not. Two polices are considered: *transparent* and *big-switch* abstraction policy (see Sec. 2.2.3). As expected, the two considered topology abstraction policies indeed do produce different CPU loads on NH. However, since the observed measurement results for both policies are predictable, we also present three estimation models which predict the CPU utilization of an NH based on the DP requirements and the corresponding topology abstraction policy. Furthermore, based on the presented measurement results and models, in the second part of this section, a realistic offline VNE problem is presented and evaluated. The presented algorithm takes into consideration the impact of topology abstraction, hence, it achieves better performance compared to the baseline algorithm based on the *SotA* approaches.

This section (i.e., Section 3.1) is organized as follows. In Section 3.1.1, the related work is presented. Section 3.1.2 presents the considered measurement setup, results and the proposed NH CPU estimation model. Section 3.1.4 presents the problem formulation and the proposed optimization approach. Thereafter, the simulation setup and performance evaluation are presented in Section 3.1.6.1. Finally, in Section 3.1.7 presents the additional discussion regarding the major points presented in this section.

3.1.1 Related Work

Since in this section, we present measurement results, CPU estimation models, and an offline VNE problem, we divide the related work into three categories. The following categories are considered: *1)* slice abstraction, *2)* network hypervisor CPU estimation, and *3)* VNE problem.

3.1.1.1 Slice Abstraction

FlowVisor is the first proposed network hypervisor [She+09] and it provides abstraction of physical ports. To be precise, *FlowVisor* shows only the physical ports (on forwarding devices) containing the tenants hosts to the tenants controller. Since *FlowVisor* acts like a transparent proxy, it is unable to

abstract or hide the intermediate switches. *AdVisor* [Sal+11] enhances the topology abstraction by hiding the intermediate nodes of the virtual path by forwarding only the Open Flow (OF) [McK+08] messages coming from the endpoint switches. Topology abstraction is improved even further in *VeRTIGO* [Cor+12], where authors allow tenants to select the desired level of topology abstraction.

OpenVirtex [Al-+14] provides arbitrary topology abstractions, with a limit that one physical switch cannot be represented as two virtual ones. Furthermore, optimization of the data-plane within the abstracted switches is possible by specifying the desired routing algorithms using the provided API. In [Han+18], authors presented Virtualization Layer (VL) developed on ONOS platform [Ber+14], which also fully supports big-switch abstraction. Furthermore, in [Han] authors evaluated the performance of the previously mentioned VL platform in terms of processing time. *CoVisor* [Jin+15] explores how to abstract the physical network in order to improve the performance of tenants' SDN controllers. On the other hand, efficient rule placement algorithm within the big-switch was proposed in [Kan+13]. Further, in [Geb+15]; [Gei+17], authors abstracted multiple OF 1.0 physical switches into one virtual switch in order to provide OF 1.3 functionalities.

3.1.1.2 Network Hypervisor CPU Estimation.

The DP performance is greatly influenced by the CP performance [TG10]. Therefore, in [Ble+16b]; [Ble+15], authors evaluated NH placement problem. The objective of their work was to determine the number of needed NH instances and their locations in the physical network. In [She+09]; [Al-+14] authors performed offline benchmarks of NH hypervisors in order to correlate the number of OF messages and SDN hypervisor CPU utilization. In order to avoid long offline benchmarks, online machine learning algorithm was proposed in [Sie+16]. Furthermore, the algorithm was extended in [SOK17] to support environments with varying resources. However, to the best of our knowledge, the impact of topology abstraction on the CPU utilization was not considered in any *SotA* work.

3.1.1.3 Virtual Network Embedding

Realization of the virtualization in SDN provides a new dimension and challenges to the VNE problem [Gue+14]. For example, in virtual SDN environment, the network hypervisor and the DP resources (e.g., NH CPU, link bandwidth, etc..) have to be considered in VNE problem. However, even in traditional networks VNE problem is NP-hard [Fis+13], i.e., it can be solved optimally, but the solving run-time increases exponentially. The run-time can be reduced by restricting the problem [Fis+13]; [Yu+08] or by employing heuristic approaches [Qin+12]; [BCB10].

However, to the best of our knowledge, the impact of different NH functions on the control plane resources (e.g., NH CPU or RAM) has not been addressed in the literature. Furthermore, the impact of different topologies on the CP resources (i.e., NH CPU) is not yet explored.

3.1.2 Topology Abstraction Measurements

In this part, the considered measurement setup and results of two topology abstraction corner cases are presented. It is shown that the topology abstraction can have substantial effects on NH CPU utilization. Additionally, an NH CPU estimation models is also presented. This model is based on the tenant requirements, such as: flow rate, network topology, and topology abstraction.

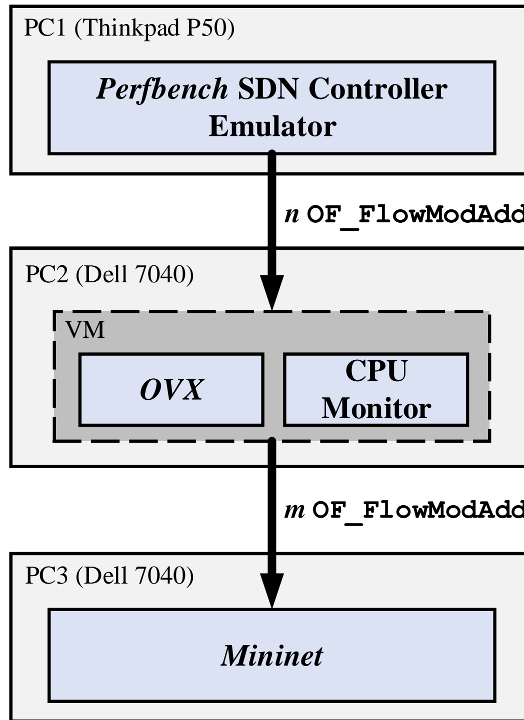


Figure 3.1: Control and Data-plane measurement setup.

3.1.2.1 Measurement Setup

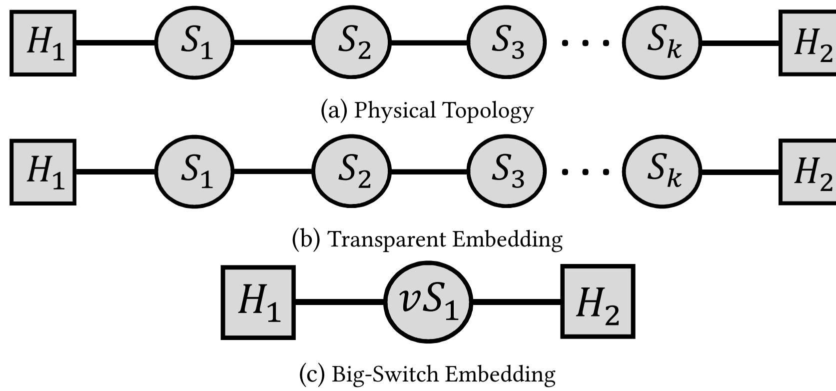
The goal of the shown measurements is to answer the following question:

Does the topology abstraction produces an impact on the NH CPU utilization? If yes, how much?

To find out the answer, the measurement setup as illustrated in Fig. 3.1 is considered. Three different Personal Computers (PC)s are used in order to emulate the SDN controller, the NH, and the data-plane network. The first PC (i.e., Thinkpad P50) emulates the SDN controller by using SDN benchmarking tool, *perfbench* [Bas+17]. *Perfbench* enables us to easily generate `OF_FlowModAdd` messages with a variable rate. The second PC (i.e., Quad-Core Dell Optiflex 7040) hosts the VM which runs the NH. OVX [Al-+14] is used as an NH. It provides fully topology abstraction, and the implementation is available online on github. Topology abstraction is realized by mapping the physical switches and ports to the corresponding virtual switches and ports, respectively. Additionally, the same VM holds VM monitor, developed with *python psutil* library [Ble+16a]. The third PC (i.e., Quad-Core Dell Optiflex 7040) is used to emulate a line DP topology as shown in Fig. 3.2a.

3.1.2.2 Measurement Scenario

In general, to process one `OF_FlowModAdd` message with a topology abstraction function, the NH has to receive the message, decode it, perform routing based on the data within the message (e.g., two requesting end-points), and potentially send multiple `OF_FlowModAdd` to the corresponding switches (as explained in background chapter). There are two ways to perform the routing. Either the NH calculates during the run-time path between two requesting endpoints (*online*) or all the paths are pre-calculated at the beginning and are stored in list or hash table (*offline*). Therefore, in the first



(a) Physical Topology

(b) Transparent Embedding

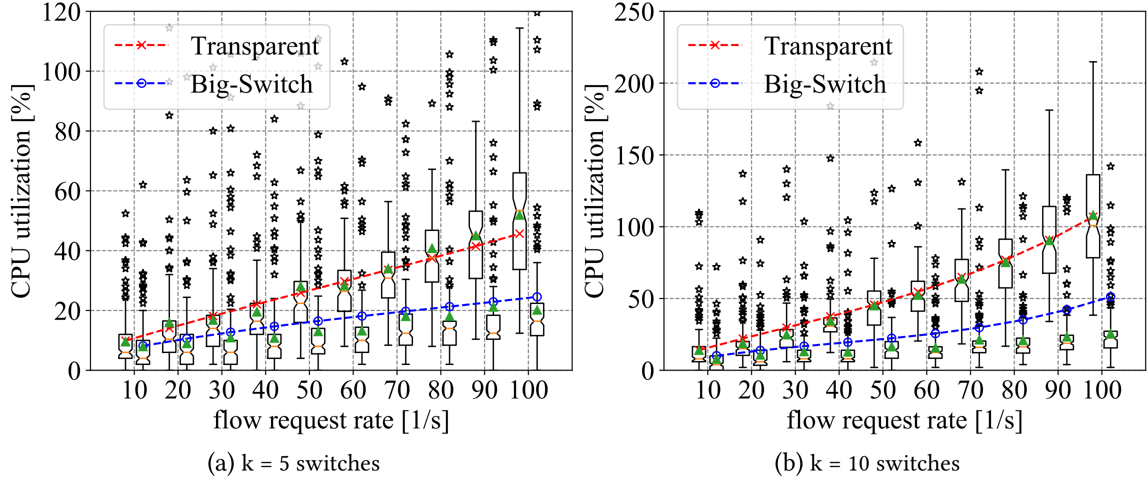
(c) Big-Switch Embedding

Figure 3.2: (a) Physical representation of the measurement topology, and the two considered topology abstraction cases: (b) transparent case with no abstraction and (c) big-switch abstraction.

case, since finding a path with a routing algorithm is more challenging in more complex topologies, the considered topology and routing algorithm can have an influence on NH CPU utilization. In the second case, since the paths are pre-calculated, the influence is minimal (hash table has constant look-up time $O(1)$). Hence, depending on the implementation details of a corresponding NH, the considered routing strategy can have an impact on the NH CPU utilization. In this part, we aim to minimize the impact of routing, and to focus only on the basic task which every topology abstraction function has to implement, i.e., receiving an `OF_FlowModAdd` message, decoding it, and potentially sending multiple `OF_FlowModAdd` to the corresponding switches. Therefore, in the following, all the measurements and modeling are performed on a line topolgy. Additionally, if we assume that the NH pre-calculates paths and stores them in a hash table, it can be assumed that the routing tasks should not have a significant impact on the presented CPU estimation models in Sec. 3.1.3.

The data-plane topology consists of two hosts (i.e., H-1 and H-2) connected in a line topology, with k switches between them. The VN is established between the two hosts and spans across all the corresponding physical switches and links as in Fig. 3.2a. Furthermore, two topology abstraction corner cases are considered: *transparent* abstraction (i.e., without any abstraction) as in Fig. 3.2b, and *big-switch* abstraction in which the VN is completely abstracted as a big-switch as it can be seen in Fig. 3.2c.

In OF, the `OF_FlowModAdd` message is used to add forwarding rules to switches. Thus, in order to establish one traffic flow between the two data-plane hosts, each switch on the path has to receive at least one `OF_FlowModAdd` message. Hence, in total k `OF_FlowModAdd` messages are sent by the NH on the Southbound Interface (SBI) for both corner cases. However, the situation on the Northbound Interface (NBI) differs based on the considered topology abstraction case. In case of *transparent* abstraction (Fig. 3.2b), the SDN controller has to generate k `OF_FlowModAdd` messages towards each switch, while the NH has to only forward the messages to the corresponding physical switches. However, in the case of *big-switch* abstraction (Fig. 3.2c), the whole DP network is abstracted, thus, the SDN controller has to generate only one `OF_FlowModAdd` message in order to establish the same traffic flow. In this case, the NH has to find a physical route between the virtual ports, and translate one northbound `OF_FlowModAdd` into k `OF_FlowModAdd` southbound messages which are sent towards each switch on the corresponding physical path.



(a) k = 5 switches

(b) k = 10 switches

Figure 3.3: CPU utilization with respect to the flow rate between two hosts when there are k number of switches between them. Left column of each box plot represents the case when VN is *transparently* embedded, while the right column box plots represents if the VN is embedded using *big-Switch*.

The CPU utilization of the NH for the two topology abstraction corner cases is compared. To do so, the number of switches in between the two hosts is varied between 2 and 10 (i.e., $k = \{2...10\}$), and the DP flow request rate between the two hosts is increased from 10 to 100 with increments of 10 (i.e., $r = \{10...100\}$). The length of one measurement instance is 90 seconds. During the each measurement instance, *perfbench* generates `OF_FlowModAdd` messages with a rate corresponding to the data-plane flow request rate. The CPU monitor gathers CPU utilization samples of the VM hosting OVX instance every 0.5 seconds. The samples are represented as in percentages of the cores used, hence, 150% corresponds to the one and a half cores utilized. The first 5 seconds and the last 5 seconds of each measurement run are discarded due to the fluctuations which are caused by start-up and close-down of CPU monitor.

3.1.2.3 Measurement Results and Observations

The measurement results for $k = 5$ and $k = 10$ are shown in Fig. 3.3. All the other measurements follow the same trend and are used for modeling. Two box plot samples are shown for each *x-axis* value, the left box-plots represent the *transparent* abstraction corner case, while the right ones represent the *big-switch* case. In both abstraction cases, increasing the data-plane flow rate increases the average NH CPU utilization *linearly*. This is due to the fact that the number of northbound and southbound messages is increased. Furthermore, it can be observed that the CPU utilization for the *transparent* case is much more pronounced. Since the NH southbound message rate is the same for both corner cases, we can conclude that forwarding $k \times r$ messages in the *transparent* abstraction case requires more CPU resources than calculating physical routes and translating r messages in the *big-switch* abstraction case.

Moreover, if we take a look at the *big-switch* abstraction case for $k = 5$ (the blue line in Fig. 3.3a) and $k = 10$ (the blue line in Fig. 3.3b), it can be seen that the CPU utilization difference is not drastic. Since the northbound messages rates in both cases are the same, it can be concluded that the number of southbound messages in this case does not make the biggest impact on the CPU utilization.

3.1.3 Modelling CPU Estimation

Based on the measurements, we suspect that the CPU utilization depends either linearly or polynomially on the required DP flow rate r, the number of switches on the path k, and the requested abstraction level a. Thus, we formulate linear, quadratic and 3rd order polynomial functions to fit the CPU utilization, as in the following:

$$g_{lin}(r,k,a) = c_0^l + c_1^l rka + c_2^l rk \tag{3.1}$$

$$g_{qua}(r,k,a) = c_0^q + c_1^q rka + c_2^q (rka)^2 + c_3^q rk + c_4^q (rk)^2 \tag{3.2}$$

$$g_{pol}(r,k,a) = c_0^p + c_1^p rka + c_2^p (rka)^2 + c_3^p (rka)^3 + c_4^q rk + c_5^q (rk)^2 + c_6^q (rk)^3 \tag{3.3}$$

where c represents coefficients in the equations. The parameter a is the requested abstraction level which represents the ratio of virtual switches on the virtual path and physical switches on the corresponding physical path. For the *big-switch* abstraction case, there is one virtual switch and k physical ones, hence $a = 1/k$. In the *transparent case*, $a = k/k = 1$. Therefore, the multiplications rka and rk actually represent the NBI and SBI `OF_FlowModAdd` message rates, respectively.

Using the *scipy* Python library, we take the average workload CPU utilization values from the measurements and find the best fitting coefficients in Eq. 3.1, Eq. 3.2, and Eq. 3.3. Table I contains all of the corresponding coefficient values. Moreover, it also shows the average relative errors for all CPU estimation functions based on the all data (Error) and the data consisting only of samples where the CPU utilization is higher than 30% (Error30). Fig. 3.4 shows the measured CPU utilization and the corresponding estimates of all the models for the *transparent* case with k = 10 switches between the end hosts. As it can be seen in Fig. 3.4 and Table 3.1, among all functions, the 3rd order polynomial fits the best, but the error is not significantly lower compared to the other models. Therefore, in Fig. 3.3, 3rd order polynomial model is used for fitting and estimating the mean CPU utilization. From Table I and Fig. 3.4, it can be generally seen that the models actually perform worse for the lower CPU utilization. Fig. 3.5 depicts the 3rd order polynomial estimation model. It can be observed that increasing either the number of switches or the required flow rate increases the CPU utilization.

3.1.4 VNE Problem Formulation

In this part we mathematically formulate the substrate physical network and virtual private network request. Furthermore, two offline VNE problems are outlined. The mentioned problems are solved optimally in order to show the importance of the correct provisioning of NH CPU resources.

3.1.4.1 Substrate Network

As the substrate network, an undirected connected graph $G^S = (N^S, E^S, h^S, A^S)$ is considered. Superscripts are used to differentiate substrate S and virtual private network V, unless otherwise specified. N^S denotes the set of all substrate nodes, which is defined as $N^S = \{n_1^S, n_2^S, ..., n_{|N^S|}^S\}$. E^S represents the set of all substrate edges $E^P \subseteq N^S \times N^S$ and $e_{i,j}^S = (n_i^S, n_j^S)$ represents one substrate edge connecting the substrate nodes n_i^S and n_j^S. A substrate path connecting the arbitrary substrate nodes

Table 3.1: Values of Modeling Parameters

Model	Linear	Quadratic	3rd Order Polynomial
c_0	5.8956	7.5945	5.4317
c_1	0.0665	0.0345	0.0418
c_2	0.0215	4.568×10^{-5}	1.8250×10^{-5}
c_3	-	0.0251	2.1590×10^{-8}
c_4	-	-9.0565×10^{-6}	0.0497
c_5	-	-	-7.2677×10^{-5}
c_6	-	-	4.2753×10^{-8}
Error	12.29%	10.87%	10.22%
Error30	7.55%	5.78%	4.49%

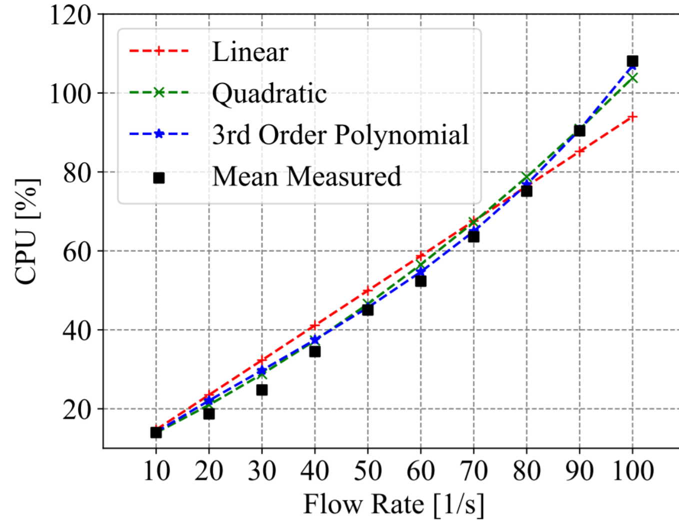
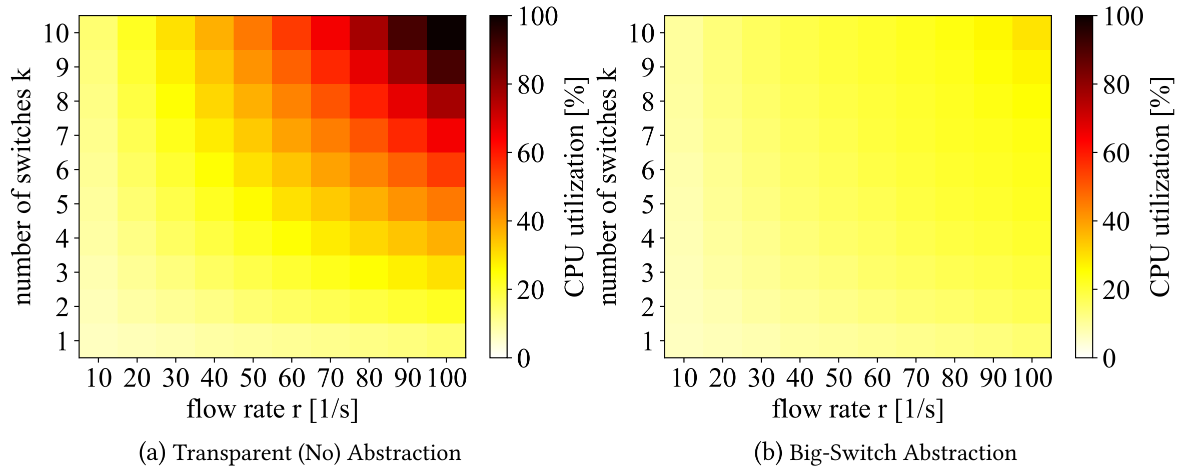


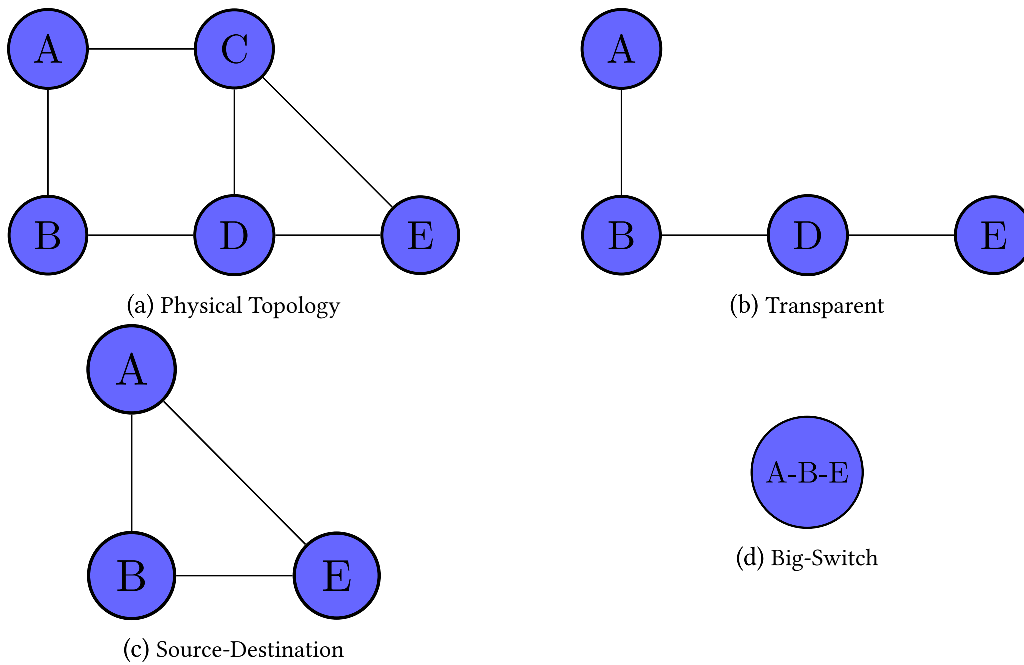
Figure 3.4: Mean measured CPU utilization for the transparent case with k = 10 switches in the data plane and the corresponding estimation models.

$n_x^S, n_y^S \in N^S$ without loop(s) is represented as a set of substrate edges, $p_{x,y}^S = \{e_{x,z}^S, e_{z,q}^S, ..., e_{r,x}^S\}$, where $n_z^S, n_q^S, n_r^S \in N^S$ are intermediate nodes on the path. Furthermore, it is considered that NH is represented as an additional node h^S, which is connected to every substrate node via an out-of-band control-plane channel. Every object (substrate edge or an NH) $o \in E^S \cup h^S$ has a capacity attribute $A^S(o) = a_o^S$. In this part, total amount of CPU resources is considered as the NH attribute $A^S(h^S) = A_{CPU}^S(h^S)$. Actually, the CPU resources are represented by the total number of cores, therefore, $A_{CPU}^S(h^S) \in \mathbb{Z}$. Besides, bandwidth is considered as the substrate edge attribute $A^S(e_{i,j}^S) = A_{bw}^S(e_{i,j}^S)$, thus, $A_{CPU}^S \cup A_{bw}^S = A^S$.



(a) Transparent (No) Abstraction (b) Big-Switch Abstraction

Figure 3.5: Estimation of CPU utilization based on the presented model for both abstraction cases, i.e., transparent (no) abstraction and big-switch abstraction.



(a) Physical Topology (b) Transparent

(c) Source-Destination (d) Big-Switch

Figure 3.6: Physical representation of the topology and three supported VNE abstraction levels.

3.1.4.2 Topology Abstraction Levels

Let us consider an arbitrary substrate topology as in Fig. 3.6a and a VN request that requests to connect the *terminals* located on the substrate nodes *A*, *B*, and *E*. In this part, three different topology abstraction levels are conidered:

- *Transparent* (Fig. 3.6b): topology abstraction assumes that all of the intermediate substrate nodes are shown to the tenant.
- *Source-Destination* (Fig. 3.6c): topology abstraction hides all of the intermediate substrate nodes, but shows the substrate nodes hosting the tenants' *terminals*.

- *Big-Switch* (Fig. 3.6d): topology abstraction hides all of the substrate nodes, hence, the whole VN is shown as one big-switch to the tenant.

3.1.4.3 Virtual Network Request (VNR)

VNE problem is NP-hard, therefore, we consider hard node location constraints (as in [Yu+08]) in the definition of VNR. In the literature, sometimes this kind of a constraint problem is also referred to as Virtual Private Network Embedding (VPNE) problem [Gup+01]. Therefore, VNR is defined as a graph $G^V = (N^V, C^V, l)$. N^V represents a subset of substrate nodes ($N^V \subset N^S$) which contains the tenants *terminals* (we refer to it as *terminal* nodes, defined as n_i^V). New services, such as Machine-to-Machine (M2M) communication, differ greatly from the traditional services as they generate a huge number of short-lived traffic flows with very low bandwidth requirements. Therefore, we define two matrices $C^V_{bw|N^V|\times|N^V|}$ and $C^V_{fr|N^V|\times|N^V|}$ to represent the bandwidth and flow rate requirements between each pair of *terminal* nodes, respectively. Moreover, in every VNR, the required abstraction level a is specified. Three different topology abstraction levels are considered (as explained in Sec. 3.1.4.2), hence, $a \in$ {transparent, src-dst, big-switch}.

3.1.4.4 Estimating NH CPU Requirements of a VN Request

In the following, we present how to estimate the total required NH CPU resources for a given VNR V_x. To achieve this, the following procedure is used. For every traffic demand between the two *terminal* nodes (e.g., n_i^V and n_j^V), it is possible to find a substrate path $p_{i,j}^S$ which contains all of the substrate edges between the two *terminal* nodes. Furthermore, in a similar way, a virtual path can be derived, defined as a set of virtual nodes (i.e., the substrate nodes shown to the tenant based on the selected abstraction requirement) between the same two *terminal* nodes ($p_{i,j}^{V_x}$). We can then define the topology abstraction level for the traffic demand $a_{p_{i,j}^{V_x}}$ as bellow:

$$a^{V_x}_{p_{i,j}} = \frac{|\ p_{i,j}^{V_x}\ |}{|\ p_{i,j}^{S}\ |} \tag{3.4}$$

where $|\ p_{i,j}^{V_x}\ |$ and $|\ p_{i,j}^{P}\ |$ represent the total number of hops on the virtual and substrate path, respectively. In fact, this metric represents the ratio of an NH's northbound and southbound `OF_FlowModAdd` messages rates. Thus, since the flow rate $C_{fr}^{V_x}(n_i^{V_x}, n_j^{V_x})$ and the number of physical nodes $k =|\ p_{i,j}^S\ |$ on the physical path is known, it is possible to estimate the required CPU for this virtual path using one of the proposed CPU estimation functions in Sec. 3.1.3. These functions are based on the measurements performed on a line topology, thus, they do not include the impact of complex topologies and routing algorithms. However, if we assume that the NH pre-calculates all the possible paths in the substrate graph and stores them in a hash table, we can assume that the overhead of routing is not significant as the look up time of hash table scales with $O(1)$. In general, using the propsed line-based estimation functions with complex VNs can introduce some error in CPU estimation. However, modeling the network with them still can provide important insights regarding the impact of topology abstraction on the NH provisioning and VNE problem.

Including the notations presented in this part of the thesis, function presented in Eq. (3.1), can be written as:

$$g_{p_{i,j}^{V_x}} = g(C_{fr}^{V_x}(n_i^{V_x}, n_j^{V_x}), k, a_{p_{i,j}}^{V_x}) \quad (3.5)$$

Finally, it is possible to estimate the total required NH CPU resources g^{V_x} for the whole VNR by summing up the corresponding required CPU values for each traffic demand between each pair of *terminal* nodes as follows:

$$g^{V_x} = \sum_{i}^{N} g_{p_{i,j}^{V_x}} \quad (3.6)$$

where N is the number of traffic demands between pairs of *terminal* nodes in in $C_{bw}^{V_x}$.

3.1.4.5 Data-plane Constraints.

As explained before, every substrate edge $e_{i,j}^S$ is associated with a bandwidth capacity $A_{bw}^S(e_{i,j}^S)$. Additionally, all of the traffic demands between *terminal* nodes are embedded in the physical network using Dijsktra's shortest path algorithm [Dij+76]. As a result, the bandwidth requirements of each terminal node pair in $C_{bw|N^V|\times|N^V|}^{V}$ can be translated to the corresponding (i.e., equal) bandwidth requirements on each substrate edge of the shortest path. Therefore, if we consider that $C_{bw|N^S|\times|N^S|}^{V_x^T}$ is the total bandwidth requirement of VNR V_x between the two *substrate* nodes n_i^S and n_j^S, we can specify the data-plane constraints as:

$$A_{bw}^S(e_{i,j}^S) \geq \sum_{x=1}^{N} z^{V_x} C_{bw}^{V_x^T}(n_i^S, n_j^S) \quad (3.7)$$

where $z^{V_x} \in \{0, 1\}$ is a decision variable which equals to 1 if the VNR V_x is embedded, and 0 if it is not. Thus, this constraint makes sure that the amount of used bandwidth for VNR does not exceed the physical link bandwidth capacity.

3.1.4.6 Control-plane Constraints.

The total NH CPU capacity is defined as $A_{CPU}^S(h^S)$. Also, g^{V_x} is used to represent the NH CPU requirements of VNR V_x. How the g^{V_x} is calculated from the data-plane requirements of a VNR V_x is explained in Sec. 3.1.4.4. Finally, the control-plane constraint can be formulated as:

$$A(h^S) \geq \sum_{x=1}^{N} z^{V_x} g^{V_x}. \quad (3.8)$$

3.1.4.7 Objectives.

The main goal of the first objective function is to maximize the embedding ratio. Therefore, it can formulated as below:

$$max \sum_{x=1}^{N} z^{V_x} \quad (3.9)$$

Table 3.2: Simulation Parameters

Parameters	**Values**
Network Topologies	Abilene, Internet2, Germany50
Physical Node Capacity (CPU)	Uniformly distr. $U(50, 100)$
Physical Edge Capacity (bw)	Uniformly distr. $U(50, 100)$
Number of VN nodes	Uniformly distr. $U(2, 10)$
VN traffic demand prob.	0.5
VN edge capacity demand (bw)	Uniformly distr. $U(0.2, 0.3)$
VN edge flow demand (flow rate)	Uniformly distr. $U(10, 100)$
VN topology abstraction level	Uniformly distr.
VN arrival time	all at start
VN lifetime	inf
VNE problem type	offline
1st VNE objective	Max-Acceptance
2nd VNE objective	Max-DP-Utilization
Solver	Gurobipy 8.0.1

The main goal of the second objective function is to maximizing data-plane utilization. Therefore, it can formulated as below:

$$\max \sum_{\forall i,j \in E^S} \frac{\sum_{x=1}^{N} z^{V_x} C_{bw}^S(e_{i,j}^S)}{A(e_{i,j}^S)}. \tag{3.10}$$

3.1.4.8 Baseline

Since the goal is to investigate the impact of topology abstraction on VNE, we consider a baseline case for comparison purpose. In the baseline case, the estimation of CPU resources does not include the topology abstraction (it is labeled as *without-abstraction* in the following figures). Hence, it is assumed that each VNR uses only the transparent toplogy abstraction. Therefore, the estimated VNE resource requirements in terms of NH CPU are overprovisioned.

3.1.5 Simulation Scenario

In this part we present the considered substrate topology and selected parameters used for the evaluation of the impact of topology abstraction on the performance of VNE problem. We implemented and solved the proposed VNE optimization model using gurobipy [Gur16], and ran the simulations on a Dell OptiPlex 7040 PC. The simulations were repeated 100 times in order to obtain stable and confident results.

3.1.5.1 Physical Network

Three widely used physical network topologies are considered to evaluate the proposed approach: Internet2 (34 nodes, 50 edges) [Hoc+14], Germany50 (50 nodes, 88 edges) [Orl+07], and Abilene (11 nodes, 14 edges) [Kni+11]. Offline VNE problem is considered, in which all VNRs are known at the start of the simulation. Also, their request time is defined an infinite (i.e., the embedded VNs are never removed until the end of the simulation). Moreover, all parameters used in the simulation are shown in Table 3.2.

We use unit-less values which are uniformly distributed between 50 and 100 ($U(50, 100)$) for substrate edge capacity constraints (data-plane bandwidth) [CRB09]; [CRB12]; [Ble+16c]. Furthermore, the number of *terminal* nodes in the VNRs is uniformly distributed between 2 and 10 ($U(2, 10)$) [CRB12]. Also, the flow request rate between these pairs is uniformly distributed between 10 and 100 flows per second ($U(10, 100)$), while the bandwidth requirement is uniformly distributed between 0.5 and 1 ($U(0.5, 1)$).

As presented in Section 3.1.2.3, *big-switch* abstraction requires the least amount of NH CPU resources, therefore, it is *the most attractive solution* for our problem. However, some industrial scenarios have deterministic delay requirements, thus, fine grained control of priority queues on every intermediate node is needed [GVK17]. Since we do not have the exact data on the distribution of VN topology abstraction requirements, we consider that topology abtraction level of each VNR is selected uniformly from the set $L = \{\text{transparent, src-dst, big-switch}\}$.

3.1.6 Simulation Results

In this part, we initially evaluate and compare how the topology abstraction level affects the total amount of needed NH CPU resources if all VNRs are embedded (CPU provisioning problem). Thereafter, we show the impact of the topology abstraction on the achieved objective function values for both defined objective functions, in case of varying NH CPU capacity.

3.1.6.1 Impact of Topology Abstraction on NH CPU Provisioning

To evaluate the impact of topology abstraction on the CPU provisioning problem, only in this subsection, the DP and CP constraints are ignored (i.e., they are set to unlimited). This makes it possible to embed all of the 100 VNRs. Further, it is assumed that in one simulation cycle, all VNRs have the same topology abstraction requirement, either *transparent*, *src-dst*, or *big-switch*. The required NH CPU resources depending on the abstraction level are shown in Fig. 3.7. For each topology, it can be observed that the *big-switch* topology abstraction case requires the least amount of NH CPU resources. The biggest difference is observed for Internet2 topology where the *big-switch* abstraction case requires 50% less resources then the *transparent* one. Furthermore, it can be seen that even though Internet2 is a smaller network than Germany50 in terms of the number of nodes, it requires the most resources. This is because Internet2 topology has a lower density of edges (i.e., the ratio of the number of nodes and edges). Hence, the paths are typically longer and require a higher number of control-plane messages in order to be established.

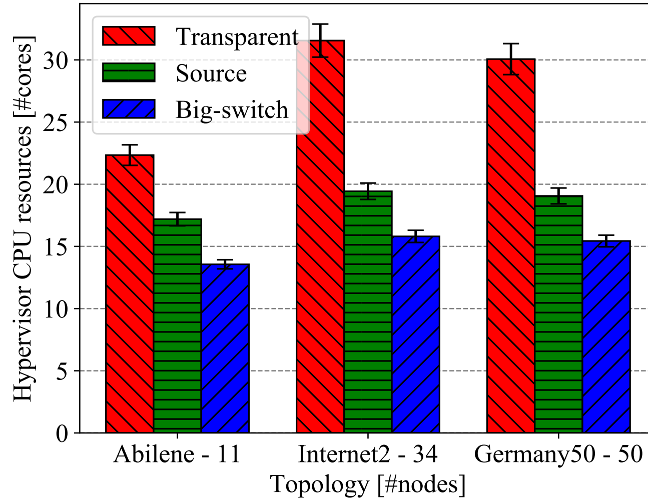


Figure 3.7: Required NH CPU resources based on topology abstraction.

3.1.6.2 Comparing the Performance of the Objective Functions

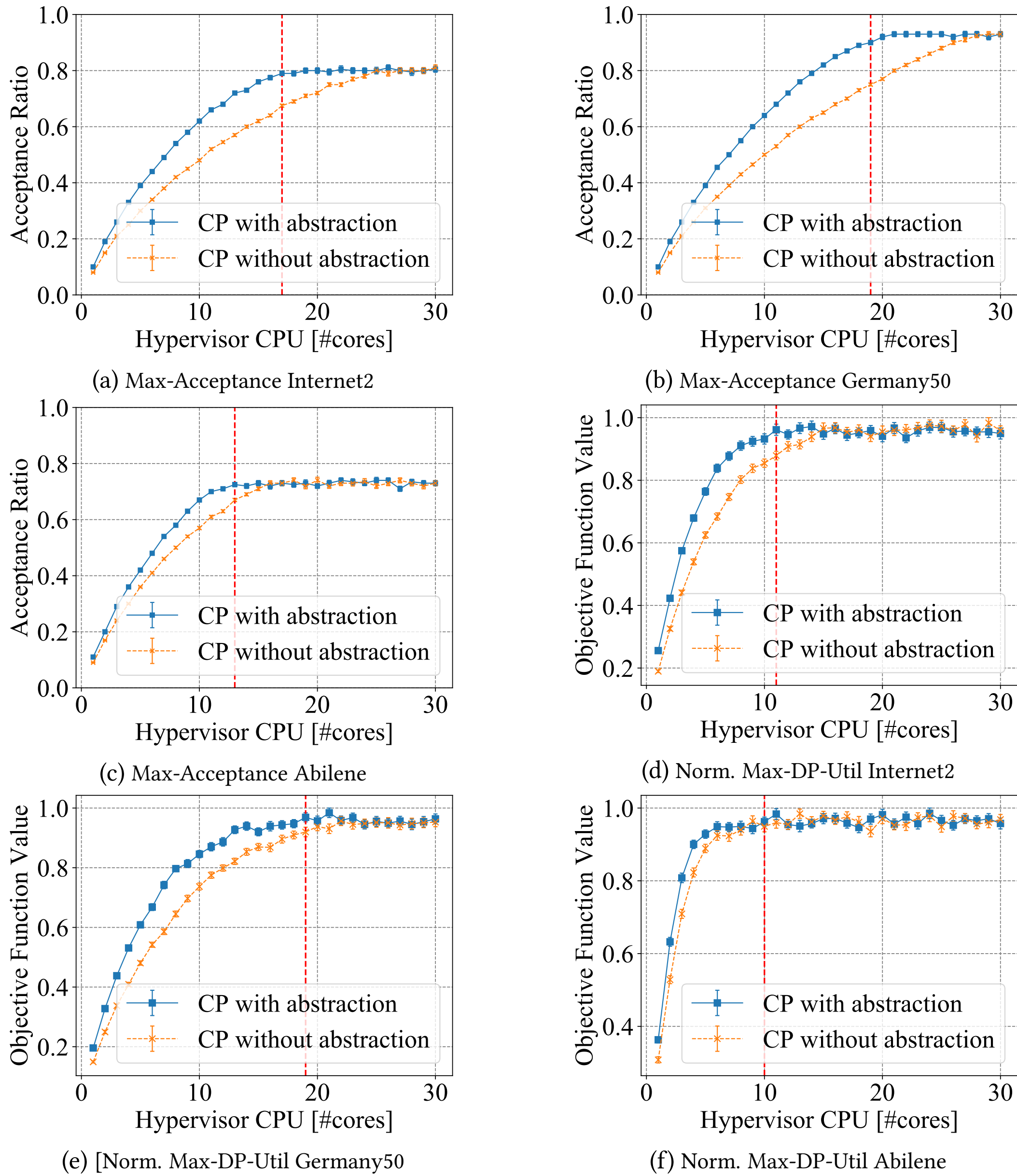
The total NH CPU capacity $A^S_{CPU}(h^S)$ is varied from 1 to 30 cores. Fig. 3.8 depicts the comparison of achieved values of VNE objective functions for the two cases: *i) with abstraction:* when the VNE problem relies on the proposed topology abstraction aware NH CPU estimation model *ii) without Abstraction:* when the topology abstraction is not included in the VNE problem.

The red vertical lines highlight the lowest NH CPU value with the achieved objective function value (e.g., either acceptance ratio or unit-less data-plane utilization) of at least 95% of the maximal one (in case when topology abstraction is used for the estimation). Therefore, if the topology abstraction is considered in the VNE problem, increasing the total CPU resources further from this line does not increase the values of objective functions significantly. In this region (on the right side of the red lines), the data-plane reseources (e.g., bandwidth) becomes the bottleneck and limits the embedding. For instance, as it can be seen in Fig. 3.8a, the acceptance ratio is almost the same for the cases when the total number of NH CPU cores is 25 or 30.

The highest difference between the achieved objective function values for the two considered cases (i.e., with abstraction and without abstraction) is observed in the close proximity of the red lines. For example, the highest absolute error of 20% is made when maximizing the acceptance rate with 20 available NH CPU cores (Fig. 3.8d). In this case, the NH CPU resources are almost optimally provisioned, thus, every decision based on the incorrect estimation of the CPU resources incurs a big error. However, when the NH CPU resources are over-provisioned (e.g., 30 available cores), there is no difference in the achieved values of the objective functions. The highest relative error is achieved in cases when there is only one available CPU core. For instance, the relative error of around 100% is made when the objective is to maximize the data-plane utilization with 1 available NH CPU core (Fig. 3.8d).

3.1.6.3 Distribution of Topology Abstractions in the Embedded VNs

Fig. 3.9 shows the distribution of the topology abstraction levels of the embedded VNRs when the total NH CPU capacity is varied from 1 to 30 cores. In case when the control-plane resources are

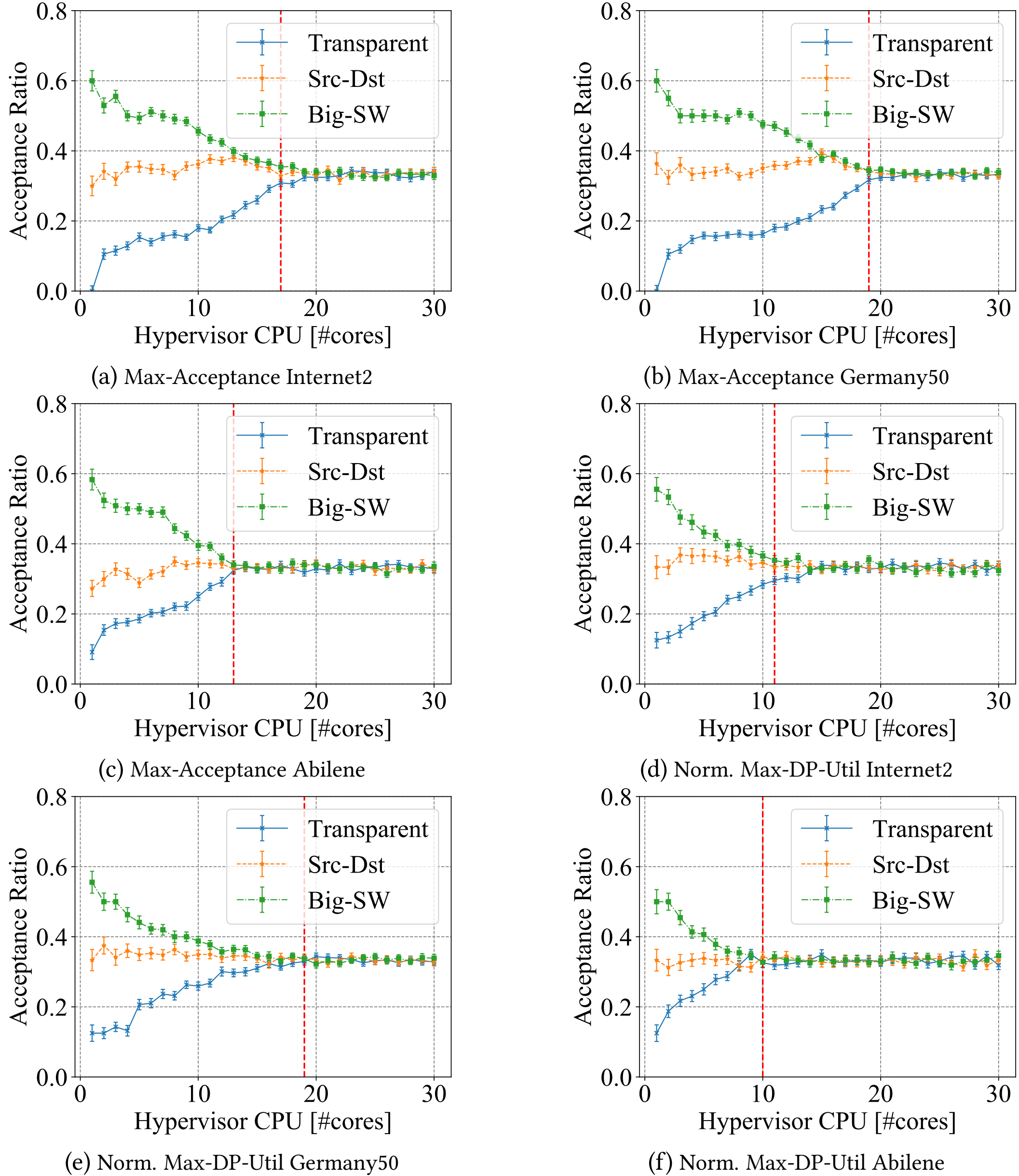


(a) Max-Acceptance Internet2
(b) Max-Acceptance Germany50
(c) Max-Acceptance Abilene
(d) Norm. Max-DP-Util Internet2
(e) [Norm. Max-DP-Util Germany50
(f) Norm. Max-DP-Util Abilene

Figure 3.8: Difference in achieved objective function values when topology abstraction is considered and when it is not. Normalized with the maximal achieved value for maximize data-plane utilization objective.

scarce (e.g., total NH CPU is between 1 and 10), it can be observed that VNs with a *big-switch* abstraction are preferred strongly. Further, Fig. 3.8a shows that when there is only one CPU available, only around 10% of VNs are embedded, and 60% of them use the big-switch topology abstraction (see Fig. 3.9a).

Although *big-switch* abstraction requires the least amount of resources in the control-plane, we can observe that the other types of VNs are also embedded. This is due to the fact that also the locations of the *terminal* nodes and flow rate requirements between them affect the required resources. For example, VNs with very short physical paths and low flow rate requirements would require a low amount of control-plane resources regardless of the abstraction level.



(a) Max-Acceptance Internet2
(b) Max-Acceptance Germany50
(c) Max-Acceptance Abilene
(d) Norm. Max-DP-Util Internet2
(e) Norm. Max-DP-Util Germany50
(f) Norm. Max-DP-Util Abilene

Figure 3.9: Distribution of topology abstraction levels of embedded VM when the NH CPU resources are varied.

In case when the NH CPU resources are over-provisioned with respect to the data-plane (on the right side of the vertical red line), the distribution of the topology abstractions becomes uniform as the data-plane becomes the limiting factor in embedding the VNR.

3.1.7 Insights and Discussion

Initially, the NH (i.e., OpenVirtex) CPU utilization is studied, measured and modeled for the two topology abstraction corner cases, *i) transparent* (no topology abstraction), and *ii) big-switch*. For the measurement setup, a line topology with a variable number of switches is considered. The presented measurements indicate that the VNs with the *big-switch* abstraction actually requires less

control-plane resources (i.e., NH CPU) compared to VNs with the *transparent* abstraction. The difference comes mostly from the difference in the OF message rate on the NH's NBI. In the *transparent* abstraction case, tenant's SDN controller has to establish flow rules on every switch, while in the *big-switch* case, this is done by the NH. Therefore, the difference is larger in bigger networks with (on average) longer paths. For instance, *transparent* abstraction of the path with 10 switches requires around 4× more CPU resources compared to the *big-switch* abstraction case. Additionally, the presented measurement results are predictable, hence, three novel NH CPU topology abstraction aware estimation models are presented in Sec. 3.1.3.

In the later part of the section, the impact of topology abstraction on the NH CPU provisioning problem and VNE problem are studied. Firstly, the proposed estimation models are used to provision the control-plane resource (i.e., NH CPU) based on the topology abstraction for three different networking topologies (Internet2, Germany50, Abilene). The simulation results indicate that considering the topology astraction while provisioning the resources can reduce the total amount of needed CPU cores for around 50% compared to the baseline (topology abstraction oblivious case).

Secondly, an offline VNE optimization problem is presented with two objective functions in order to evaluate how different topology abstraction levels influence the embedding of VNs (i.e., the selection of VNR to embed). The achieved objective function values are compared in different scenarios (e.g., different toplogies, amount of available NH CPU resources) with a baseline algorithm which does not consider topology abstraction. The simulation results for three different networking topologies (Internet2, Germany50, Abilene) indicate that including the topology abstraction in VNE problem improves the results significantly in the cases when the NH CPU cores are the limiting resource in a network. The highest observed absolute error is around 20% in case when the NH has only 10 available cores for the whole Internet2 network. Moreover, it is also shown that the VNs with the *big-switch* abstraction are heavily preferred in cases when the NHs have low amount of available CPU resources. However, if the NH is over-provisioned, the topology abstraction does not influence the embedding.

3.2 QoS-Aware Network Hypervisor Resource Provisioning

The previous section is motivated by the fact that the impact of various NH functions (e.g., topology abstrction) on the CPU utilization of a NH is still not well studied in the literature. Therefore, the previous section has two main goals: *1)* to demonstrate that different topology abstraction policies can have a significant impact on the CPU utilization of a NH, and *2)* to assess how much can such a NH function influence the performance of NH provisioning and VNE problem. Additionally, the previous section also unravelled another interesting effect which motivated the work presented in this section.

To be precise, it became evident that different DP scenarios can have a significant impact on the CPU utilization of an NH. For instance, managing larger networks (with higher number of nodes or switches) requires significantly more resources compared to the smaller ones. Therefore, the NH CPU utilization can vary greatly depending on the scenario. This effect motivated us to to investigate the problem of provisioning the CPU resources of an NH. For instance, overprovisioning the CPU resources of a NH based on the worst-case is simply waste of resources. On the other hand, underprovisioning the resources is risky as it can lead to performance degradation (e.g., increase in the processing time). In this section, a novel approach which provisions NH CPU resources efficiently, while avoiding performance degradation is presented. Three steps are taken to achieve the aforementioned goal: *(i)* a profound measurement campaign is conducted to determine what is the minimum amount of CPU resources that needs to be allocated to a NH in order to have no performance degradation; *(ii)* the key properties of VNs that affect the CPU utilization are studied; *(iii)* a precise CPU prediction model is developed. Further, the presented evaluations indicate that provisioning the CPU resources of an NH based on the proposed QoS-Aware prediction model does not degrade the NH forwarding performance.

The rest of this section (i.e., Section 3.2) is organized as follows. Section 3.2.1 motivates the alreday introduced problem and it shows the benefits of solving it. Section 3.2.2 presents the related work. Section 3.2.3 outlines the benchmarking procedure, the considered measurement setup and its results. The NH CPU prediction model is developed in Section 3.2.4 and thoroughly evaluated in Section 3.2.5. Finally, Section 3.2.6 presents a short discussion regarding the main points of this section.

3.2.1 Motivation: Predictable Virtual Network Performance

This section presents a short measurement study which motivates the problem of provisioning the CPU resources of an NH. Additionally, it also highlights the potential benefits which could be obtained by solving such a problem.

To begin with, different strategies from *SotA* can be used to allocate hardware resources to hypervisors. One existing strategy simply over-provisions hypervisors [She+09]; [Ble+19] — a clearly too expensive strategy in terms of resource consumption. Another strategy is to derive hypervisor performance models [Sie+16]; [SOK17] and to provision resources accordingly. However, as it will be shown, *SotA* performance models fall short in terms of accuracy and precision.

In order to illustrate this, we evaluate the CP processing latency of one NH (i.e., *FlowVisor*) for varying amount of allocated CPU resources (see Fig. 3.10). In this scenario, allocating around 90 %

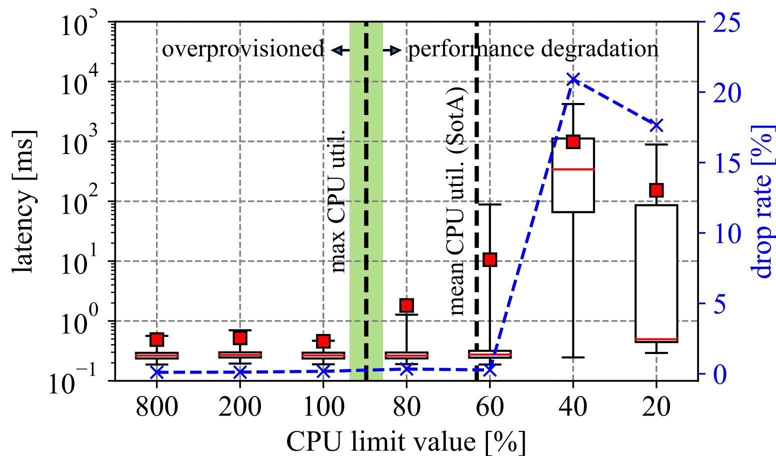


Figure 3.10: Impact of the allocation of CPU resources of an NH on the control plane message latency and loss. Dashed black lines show the maximal and average observed CPU utilization during an unconstrained run, where all of the cores where allocated to network hypervisor (CPU limit is 800 %). The server running an NH has in total 4 physical cores, thus 8 hyperthreads, making the maximal CPU utilization of 800 %.

capacity of a single CPU core represents an ideal resource allocation decision. In Fig. 3.10, this is shown as the highlighted area around the left dashed line (i.e., green area). Allocating fewer resources, i.e., *under-provisioning* the CPU resources, increases the CP message latency by up to three orders of magnitude and the loss of CP messages by 20 %[1] — a clearly unacceptable performance degradation. On the other hand, *over-provisioning* the CPU resources does not yield performance benefits: the latency stays below 1 *ms* for 100 % – 800 %[2] of CPU capacity. However, as the figure illustrates, over-provisioning leads to a waste of resources and is actually not needed when having a precise performance model. Furthermore, Fig. 3.10 also shows why state-of-the-art CPU prediction approaches [Sie+16]; [SOK17] are not suitable for provisioning. They simply predict the mean CPU utilization (in this scenario 60 %), which results in under-provisioned NH. An ideal NH provisioning system allocates the least amount of CPU resources so that no performance degradation occurs; in this case, the ideal allocation lies around 90 % of one CPU core. Furthermore, state-of-the-art prediction models are also too simplistic: they base their CPU prediction solely on the CP message rate [Sie+16]; [SOK17]. Thus, these models ignore the potential impact of (virtual) network parameters and dynamic configuration changes (e.g., the number of VNs or a changing VN topology in case of varying demands or, e.g., failures). In this chapter we tackle the challenge to derive generalizable performance models that allow for precisely determining such CPU provisioning in various scenarios.

3.2.2 Related Work

The presented approach for provisioning the CP resources (i.e., CPU) of an NH is based on a carefully designed measurement based CPU prediction model. Since an NH can be considered as a Virtual Network Function (VNF), firstly, *state-of-the-art* VNF and NH resource prediction models are described

[1] In case the resources are limited to 20%, an NH throttles the Transmission Control Protocol (TCP) connections in order to reduce the total amount of received messages. Hence, in this case processing time and latency are better compared to a scenario when the limit is 40%.

[2] Note that 800 % means a dedicated reservation, i.e., pinning of 8 threads.

along with their shortcomings. Furthermore, as the presented model is based on comprehensive NH measurements, subsequently, existing works dealing with the benchmarking of NHs are covered.

3.2.2.1 Resource Prediction Models

In order to reduce excessive power consumption in cloud computing systems through dynamic resource scaling, many authors focused on designing accurate VNF resource prediction models [Suk+16]; [JKE17]; [Mij+17]; [Mij+16]; [MAC18]; [Tan+19]. Although an NH can be considered as a VNF, these approaches cannot be directly applied. For instance, they often consider different input parameters (e.g., Internet Protocol (IP) source address, TCP destination port etc.) [JKE17]; [MAC18] which do not affect NH CPU utilization. Or they are based on already observed CPU samples [Mij+16]; [Mij+17]; [Tan+19]. However, as the VN configurations can change over time (e.g., number of virtual switches), the prediction performance of these models would also suffer.

On the other hand, there are also a few NH specific CPU prediction models [Sie+16]; [SOK17]; [Đer+18]. However, there are two problems with these approaches. Firstly, they predict the average NH CPU utilization only. Using the average CPU utilization only does not count for potential variability in the overall CPU utilization. This can result in a significant performance degradation of the perceived tenant performance (as shown in Sec. 3.2.1). Secondly, the prediction is only based on the CP message rate. Thus, the algorithm cannot react to changes in the VN configuration parameters, e.g., the number of virtual switches or hosts. Furthermore, as these parameters have a significant impact on the utilization of NHs [Ble+19], the prediction performance would suffer in dynamic scenarios. On the contrary, the proposed model predicts the 90th percentile of the CPU utilization, which does not incur a forwarding performance degradation on average. Moreover, the model also accounts for performance-critical parameters such as the number of virtual switches or ports.

3.2.2.2 Hypervisors Benchmarks

Network hypervisor benchmarks have so far either focused on exploring the processing time of various CP messages [Han]; [She+09]; [Al-+14]; [DKR13]; [Nur+19]; [Jin+19]; [ZA20]; [Ble+19]; [BBK15] or on measuring the CPU utilization of NHs in various different settings [She+09]; [Ble+19]; [Sie+16]; [SOK17]; [Đer+18]. In contrast to our benchmarks, evaluating how to predict NH CPU requirements based on different parameters, e.g., number of VNs, has been ignored so far.

Furthermore, some of the aforementioned studies suggest that the CPU utilization of NH and the processing time is only correlated with a subset of VN parameters, e.g., the number of tenants [Sie+16]; [SOK17] or the number of virtual and physical switches [Ble+19]. However, these studies consider only very basic combinations of virtual and physical network parameters and settings, e.g., a single-switch topology [Sie+16]; [SOK17], only a line topology [Nur+19]; [Đer+18] or two-port switches [Ble+19]. Therefore, the impact of arbitrary topologies (in terms of the number of switches and the interconnecting links) is not considered. In this section, comprehensive NH performance benchmarks (including latency and CPU) are performed on various different and realistic physical and virtual network topologies. In particular, a multitude of impactful factors are investigated. Furthermore, the benchmarks are also tailored with the goal of detecting and learning the impact of various physical and VN parameters in a fast and efficient manner.

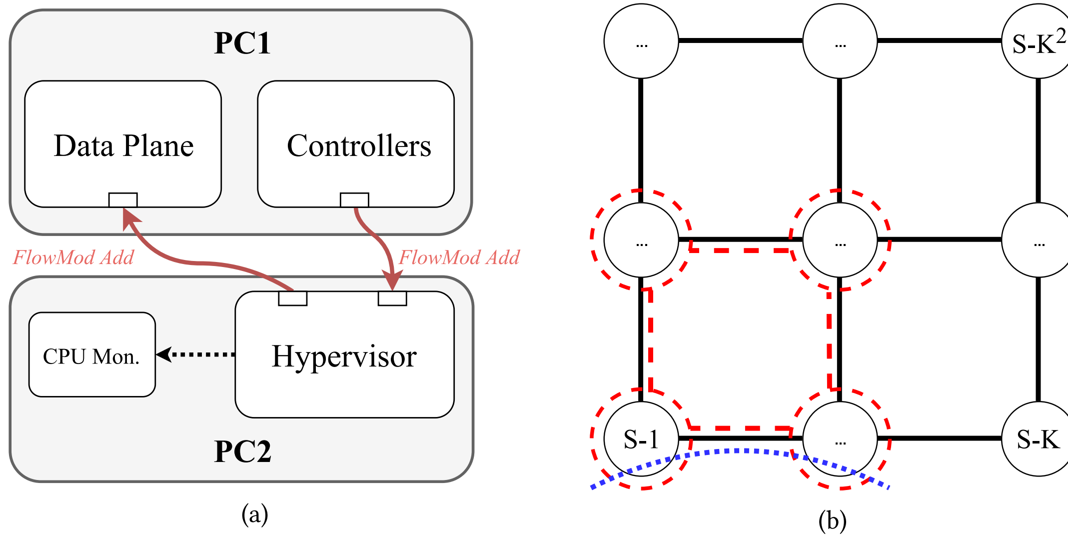


Figure 3.11: (a) Measurement setup consisting of 2 interconnected PCs, and (b) illustrations of the physical data plane grid topology of dimension $k = 3$ (black solid lines), with example grid VN of dimension $k_v = 2$ (red dashed lines), and a possible flow request spanning over two virtual and physical switches (blue dotted line).

3.2.3 Network Hypervisor Benchmarking

In this section we firstly introduce the used measurement setup (Sec. 3.2.3.1) and present the considered VN parameters (Sec. 3.2.3.2). The parameters are selected based on the insights presented in the background section (Sec. 2.2.3). Further, Sec. 3.2.3.3 we presents the used benchmarking procedure, while the results of performed measurement campaign are presented in Sec. 3.2.3.4.

3.2.3.1 Measurement Setup

In order to better understand how different VN configurations (e.g., VNs with different amount of virtual switches) influence NH resource utilization, initially we conducted the measurements on a smaller setup. Having such a setup provides a more controlled environment, thus, the impact of various parameters is easier to investigate. The measurement setup is depicted in Fig. 3.11a. It consists of two PCs equipped with Intel quad-core i7-7700 CPUs, 16GB of RAM, and running Ubuntu 14.04 LTS.

The first PC (*PC1*) *(i)* runs *mininet* [LHM10] to emulate a physical DP network, and *(ii)* runs the tenants SDN controllers as multiple *Ryu* [Com17] instances. The controllers generate CP messages with the goal of embedding a certain number of flows per second (described in Sec. 3.2.3.3).

The second PC (*PC2*) runs FlowVisor (FV) as the network hypervisor. The logging and state keeping features, which are not necessary for the normal operation of the NH, are disabled. The CPU utilization of *PC2* is sampled every 0.1 second (lower values are not recommended) by a CPU monitor implemented with the Python *psutil* [Rod] library. CPU utilization is measured in percents of a single-threaded core: since the PC has four cores with hyper-threading, CPU utilization ranges from 0% to 800%. The messages sent by the controllers are received, translated, and forwarded by FV towards the corresponding DP switches emulated by *mininet*.

Table 3.3: Evaluation parameters.

Parameter	Notation	Values
number of tenants	t	2, 3, 4, 5
physical topology size	k	4, 9, 16, 25
per-tenant flow request rate	r_i	90, 180, 280, 400
per-tenant flow length	l_i	2, 3, 4, 5, 6, 7
per-tenant virtual topology size	v_i	4, 9, 16, 25
Per-tenant number of virtual ports	p_i	1, 2, 3, 4, 5

The amount of available resources allocated to an NH is controlled with *cpulimit* tool. The *cpulimit* tool is process-based, meaning that if a specified process exceeds the allowed CPU resource consumption, it uses SIGSTOP and SIGCONT POSIX signals in order to throttle the process accordingly.

The physical DP and tenants controllers run on the same PC. To ensure that this does not affect the measurements, we make sure that the PC is always underutilized during the measurements.

3.2.3.2 Evaluated Parameters and Scenario

We focus on evaluating the impact of flow embedding tasks on the required NH CPU resources; our CP traffic between tenant controllers and their virtual networks consists of `OF_FlowModAdd` messages only (the traffic exchanged during the initial OF handshake procedure and the necessary periodic traffic is excluded). This is a standard choice as flow embedding is *(i)* a paramount functionality of the remote control of networks, and *(ii)* many applications, e.g., industrial applications with strict QoS requirements messages [Van+19b], can be implemented solely with `OF_FlowModAdd` messages. In order to establish one flow between two hosts, tenant's controller generates one `OF_FlowModAdd` message towards each switch on the chosen shortest virtual path. It is considered that tenants add rules matching on the physical input port and unique destination IP addresses. While the action is to forward (output) the matching packets to a certain port. Input and output ports are determined by the shortest path calculation. The NH receives the corresponding messages and processes them as described in Sec. 2.2.3.

Based on the insights presented in the background section (see Sec. 2.2.3), the following evaluation parameters are considered (see Tab. 3.3):

- *Number of tenants t:* For each tenant, the NH has to maintain additional TCP/OF connections from the virtual switches, i.e., from the hypervisor, to the tenant controllers. Additionally, it also has to store the isolation and abstraction policies, and potential state variables.
- *Physical topology size k:* The NH appears as a controller to all physical switches. Hence, an NH must keep one TCP/OF connection towards each physical switch. Thus, the number of switches k in the physical topology can potentially affect the required CPU resources.

- *Per-tenant flow request rate* r_i: Each tenant adds flows to its VN with a pre-defined rate r_i (with uniform arrival distribution). Increasing the rate linearly increases the number of `OF_FlowModAdd` messages on both interfaces of the hypervisor (i.e., on SBI and NBI interface). Since the message have to be received from the tenant controllers and forwarded towards the physical switches. As a consequence, the rates might conceivably increase CPU utilization of an NH [Sie+16]; [SOK17]; [Đer+18].

- *Per-tenant path-flow length* l_i: The longer the path of a flow is, the more messages have to be processed and translated by the NH (one message per physical switch). The path-flow length is defined as the number of physical switches between the end hosts of the flow. FV does not provide any topological abstraction: even if the tenants only request to manage two physical switches, they might have to also control all the switches connecting the two requested physical switches.

- *Per-tenant virtual topology size* v_i *(*$v_i \leq k$*):* FV guarantees that the tenants can only modify their own VNs. Hence, every `OF_FlowModAdd` message received by FV has to be inspected in order to ensure that the tenants are only modifying the switches in their VNs. Thus, with more virtual switches, the larger the virtual switch list is, which could affect the workload.

- *Per-tenant number of virtual ports* p_i*:* Before forwarding `OF_FlowModAdd` messages to the physical switches, the hypervisor translates virtual port numbers to physical port numbers and makes sure that a tenant is using only physical ports attached to its VN. Having a larger number of virtual ports can hence increase the lookup time, in turn affecting the workload of the NH.

In our scenario, we assume that each tenant has one host attached to each switch which is mapped to one virtual port. Furthermore, each tenant also requests the additional virtual ports in order to interconnect its virtual grid network. *Note:* In order to vary the number of virtual ports p_i, we use a different method, we allow tenants to also attach the virtual ports (hosts) dedicated to other tenants, thus increasing the amount of virtual ports.

Chosen Parameter Values (Tab. I). Current *state-of-the-art* SDN-enabled carrier grade switches are being shipped with small flow table size and they cannot handle high CP traffic rates [KPK14a]; [KPK15]. For instance, the 10G forwarding device PICA P-3290 can only handle up to around 1000 rule/flow updates per second [KPK15]. Therefore, in this section we consider similar parameter values as the goal is to demonstrate that the proposed solution is capable of supporting carrier grade SDN-enabled hardware. For example, if we consider 5 tenants, where each tenant has a flow request rate of 400 per second, in the worst case, one physical switch could experience up to 2000 update messages per second. This value is around 2x higher compared to the supported rate of PICA P-3290. Furthermore, the maximal considered sizes the physical (grid) topology and VNs is consistent with common topologies such as Internet2 [Hoc+13] and Nobel EU [Orl+07].

3.2.3.3 Measurement Procedure

Fig. 3.12 depicts the general measurement procedure. A measurement scenario is defined by the set of evaluation parameters values defined in the previous subsection (see Sec. 3.2.3.2). For one

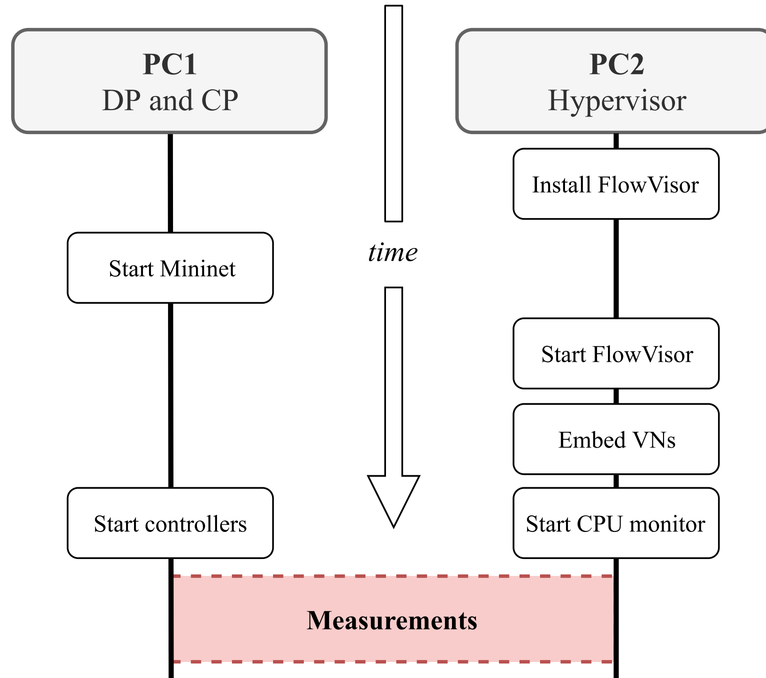


Figure 3.12: Benchmarking procedure. PC1 emulates the data plane and the tenants SDN controllers, while PC2 is running the virtualization layer and CPU monitoring script.

measurement configuration, the measurement runtime is 60 seconds and it is repeated 10 times for statistical significance (unless stated otherwise). Furthermore, we discard the observed samples during the first 10 seconds of the run, in order to avoid any potential transient phase due to initial switch connections or transport connection stabilization. Exploring all possible combinations of our six evaluation dimensions is time-wise infeasible. For instance, if we vary every parameter five times, the total measurement time would amount to a few months. Therefore, we perform the measurements only for certain scenarios with manually chosen parameter values, selected with the goal of inferring parameter scaling dependencies.

Running the measurements one after each other during the considered measurement campaign could lead to software aging effects [Hua+95]; [Gar+98], i.e., the CPU utilization of FV could increase over successive runs of the same scenario. Hence, before each run, FV is completely reinstalled. That is, all configuration files, logs, and the database are deleted, and FV is rebuilt. Finally, all remaining processes from previous runs are killed and the memory cache is cleared.

Firstly, the DP topology is started based on the measurement scenario. FV is then booted and the tenants VNs are embedded with the corresponding requirements based on the chosen configuration of the measurement scenario. The flowspace of tenants corresponds to different /16 subnets and all the requested virtual switches with their virtual ports. We consider grid topologies for both the physical and VNs because they are easy to scale up or down with non-random parameters. Occasionally, embedding large VN can overload the NH, resulting in a software crash. Such runs are repeated.

After all VNs are embedded, the CPU monitor is initialized and the tenants controllers are started. Before starting the measurement, it is ensured that all controllers finished the initial OF handshake

procedures with all of the requested virtual switches (i.e., with FV). Furthermore, it is also ensured that all tenants SDN controllers finished generating their corresponding path request list. That is, each tenant initially generates randomly 100 paths, where each path is generated with Dijkstra's shortest path algorithm [Dij59] between two randomly selected end hosts. The path is defined as a set of switches on the path, with the corresponding ports. In certain measurement cases, all paths are supposed to have the same length. Thus, before adding a path to the corresponding path request list, we simply repeat the path generation procedure until we obtain a path with desired length. During the runtime, a flow request is then defined by generating *(i)* a unique destination IP address within the tenant's flowspace and by *(ii)* randomly selecting a path from the path request list. This is done in order to avoid heavy path computations during the measurement runtime, as it could influence the precision.

After the initialization procedure, the controllers (each tenant has one) start adding flows with rate r_i[3]. The generation of flows is uniformly spaced, and the flows are generated based on the aforementioned path request list and unique IP address. In this section, we assume that all tenants have the same rate, virtual topology size and number of virtual ports.

If the controller sends multiple OF messages as one TCP segment, this reduces the total workload of the NH, in contrast to sending one OF message per TCP segment [Ble+19]. However, as the flow generation is uniformly spaced in the time, merging of multiple TCP segments almost never occurs on the controllers side.

3.2.3.4 Measurement Results

This part reports the results of presented measurement campaign. Firstly, the stability and repeatability of the measurements is studied, i.e., it is evaluated if `OF_FlowModAdd` generation rates are stable and if repeating one measurement scenario produces the same results (as the already observed run). Afterwards, the impact of the considered VN parameters on the observed CPU utilization is evaluated.

Measurements Stability. Fig. 3.13 shows the measured CPU utilization time series for the complete duration of one run, alongside with the total flow request rate $\sum_{i=1}^{n} r_i$ generated by the controllers. All flows have the same length, thus the amount of `OF_FlowModAdd` messages per second received by FV is constant and directly correlated with the flow request rate. However, even though the flow generation is uniformly spaced and stable (the maximum variance is in range of a few percents), it can be observed that CPU utilization exhibits high variability, with multiple extreme peaks. For instance, the minimum observed CPU utilization is close to 0 %, the maximal is around 198 %, while the mean is 45 %. During run-time, in order to avoid blocking the TCP socket/connection, FV places the received messages in a queue, which is then periodically cleared. Hence, the workload oscillates with time, suggesting that predicting the exact CPU utilization in one specific time instance of one run is hardly possible.

[3] The rate of flow addition is configured in *Ryu* through simple *sleep* commands. This is quite imprecise but, as shown in Fig. 3.13, is stable enough. Hence, we define the r_i as the mean rate *actually* generated by *Ryu* and obtained through post-processing of the traces.

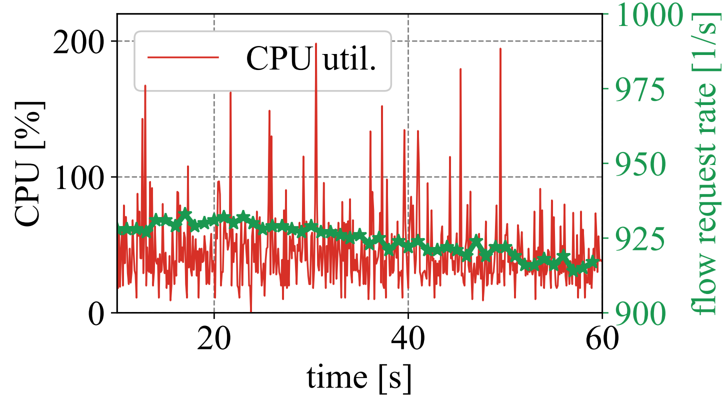


Figure 3.13: Observed time series of CPU utilization (red line with left y-axis) during one measurement run with the following parameters: $t = 5$, $r_i \sim 184$, $k = 4 \times 4$, $v_i = 4 \times 4$, $l_i = 4$ and $p_i = 1$. Furthermore, the green scattered line with the right axis represents time series of the total observed flow request rate per second during the aforementioned run.

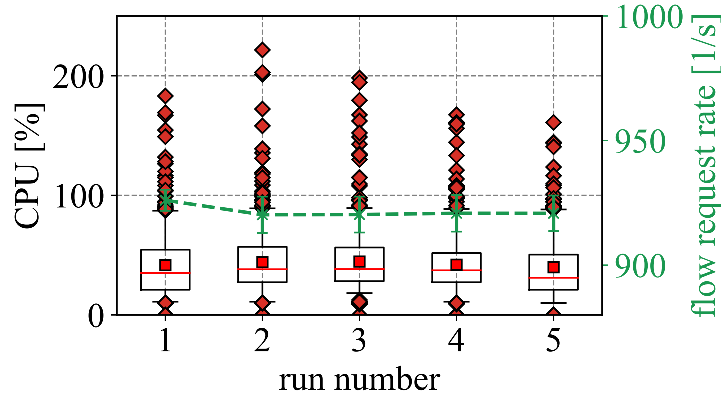
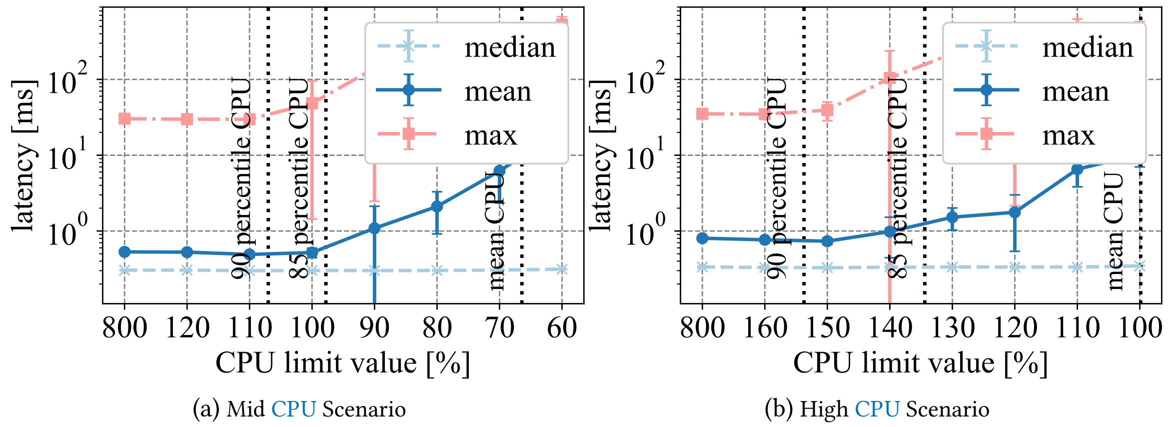


Figure 3.14: Distributions of measured CPU utilization for five measurement runs with identical evaluation parameters, i.e., $r_i \sim 300$, $k = 5 \times 5$, $v_i = 4 \times 4$, $l_i = 4$ and one host per virtual switch. The boxplots whiskers correspond to the 5% and 95% percentiles. The green line shows the mean total `OF_FlowModAdd` rate and its standard deviation.

Although the observed CPU utilization within one run varies, repeating the same measurement run multiple times with the same parameters produces almost identical CPU utilization distributions. For instance, in Fig. 3.14, the mean observed CPU utilization of all five measurement runs falls within the range of 39%–45%. This indicates that modeling and predicting the statistical properties (e.g., median) of an NH CPU utilization profile is indeed feasible.

Allocating a Sufficient Amount of Resources. Precisely allocating hypervisor resources is only needed in case a CPU limitation truly affects the network performance, e.g., NH forwarding latency. Accordingly, for precise performance modeling, it is important to find a point, i.e., statistical value, which is used for allocating hypervisor resources. Ideally, such point would minimize the amount of allocated CPU resources while avoiding performance degradation. In order to determine it, we evaluate two randomly generated scenarios, first scenario has higher CPU requirements while the second one has lower. Fig. 3.15 shows the impact of limiting the available CPU resources of the NH on the CP processing time for the two aforementioned scenarios. Furthermore, statistical properties (mean,



(a) Mid CPU Scenario

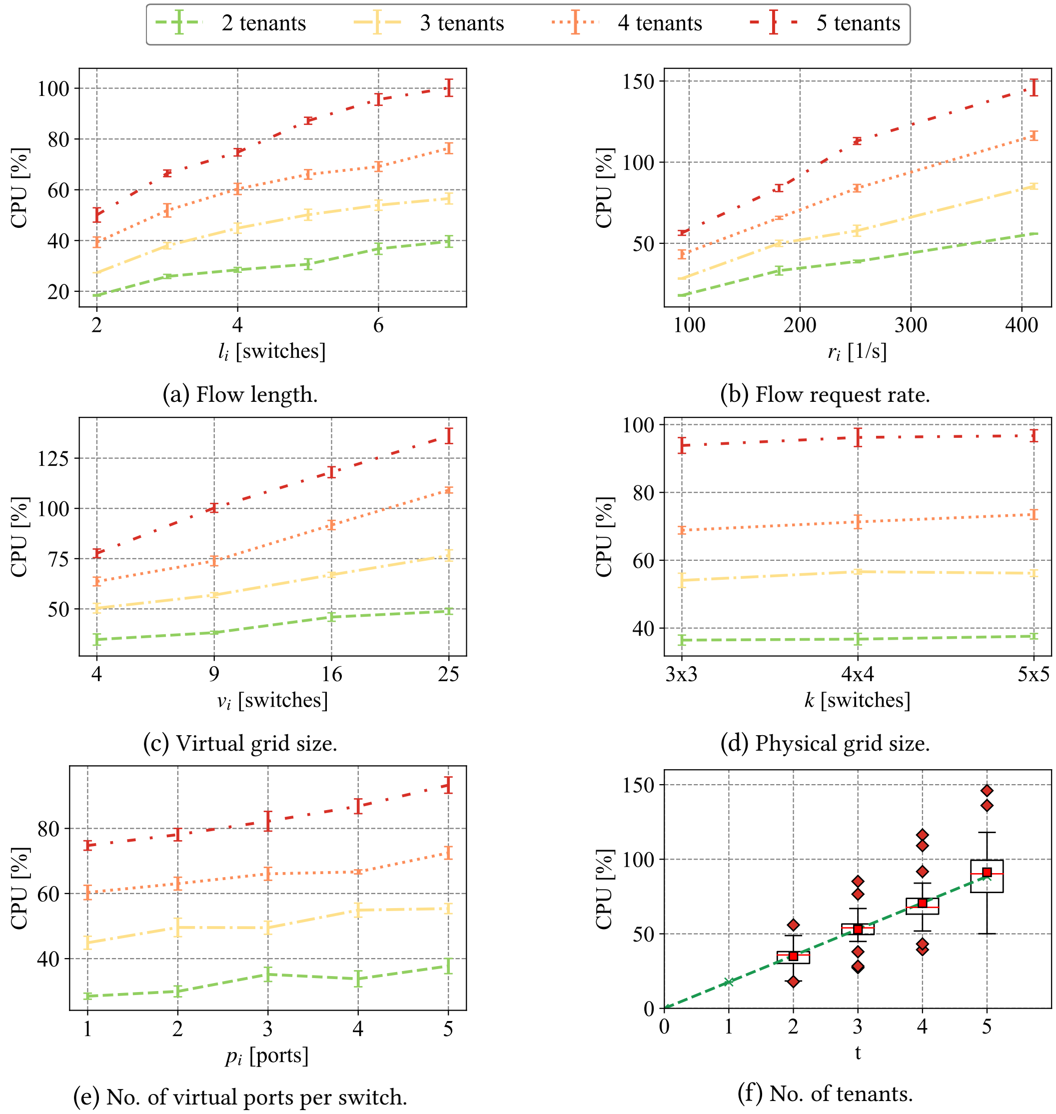
(b) High CPU Scenario

Figure 3.15: Impact of limiting the CPU resources allocated to NH on the observed CP processing time latency (median, mean and maximal) for two different measurement cases. Every run with different CPU allocation limit is repeated 5 times, and the mean values of these 5 runs are shown with their corresponding confidence intervals.

85th percentile and 90th percentile) of the measured CPU utilization profile of an unconstrained run (i.e., run when all of the resources are allocated to the NH, that is CPU limit is set to 800%) are shown as dashed vertical lines. Firstly, provisioning the CPU resources based on the observed mean (*state-of-the-art* approach) or 85th percentile incurs a significant increase of both mean and max latency. On the other hand, provisioning with 90th percentile does not produce the same impact, i.e., the latencies stay the same as in unconstrained or baseline scenario (i.e., CPU limit is 800%). Therefore, we advocate to use the **90th percentile** as a profound match: the 90th percentile offers a good trade off between performance predictability and resource allocation overhead. *Note:* Different use cases might have different performance requirements, thus using any other percentile value could also pose as a valid solution – the chosen value only defines a trade off between performance and resource consumption.

From now on, the aforementioned evaluation parameters (see Sec. 3.2.3.2) are evaluated while considering only the 90th percentile CPU utilization (with CPU utilization we refer to 90th percentile CPU utilization). In the following, we strive to learn the scaling dependencies in order to generate a prediction model capable of supporting arbitrary topologies and randomly generated VN requests. Fig. 3.16 shows the observed CPU utilization values of an NH for all the considered parameters. Each value is a mean of 10 runs (for each run we observe 90th percentile CPU utilization) and the corresponding confidence intervals.

Flow Length. Fig. 3.16a shows the impact of the flow length on the NH 90th percentile CPU utilization. Since we disabled the state keeping feature, the messages are processed independently. Thus, the total number of `OF_FlowModAdd` messages increases linearly with the flow length; in turn the CPU utilization increases linearly with the `OF_FlowModAdd` message rate. For example, for 5 tenants adding 184 flows per second, the measured CPU utilization increases with the path length from around 50 % to around 100 % linearly.



(a) Flow length.

(b) Flow request rate.

(c) Virtual grid size.

(d) Physical grid size.

(e) No. of virtual ports per switch.

(f) No. of tenants.

Figure 3.16: (a)–(f) Impact of different evaluation parameters on the NH 90th percentile CPU utilization on a physical grid topology for 2 to 5 tenants. Plots show mean observed CPU utilization values of 10 runs along with the 95% confidence intervals assuming uniform distributions. The following parameters are presented: (a) path length with $r_i = 184$, $k = 4 \times 4$, $v_i = 4 \times 4$ and $p_i = 1$, (b) flow request rate with $k = 5 \times 5$, $v_i = 4 \times 4$, $l_i = 5$ and $p_i = 1$, (c) virtual topology size with $r = 430$, $k = 5 \times 5$, $l_i = 3$ and $p_i = 1$, (d) physical topology size with $r = 430$, $v_i = 3 \times 3$, $l_i = 3$ and $p_i = 1$, (e) number of virtual ports dedicated for connecting the hosts per switch with $r = 184$, $k = 4 \times 4$, $v_i = 4 \times 4$, $l = 4$. (f) shows the data of (a)–(f) as box plots for the different number of tenants (5% and 95% percentile whiskers are used).

Flow Rate. Similarly, the `OF_FlowModAdd` message rate and the CPU utilization increases linearly with the flow request rate (see Fig. 3.16b). For instance, on average, for 5 tenants with path lengths of 6, embedding 5×90 flows per second requires only around 56 % CPU resources, while embedding 5×434 flows per second requires around 146 %. As both of these parameters impact the `OF_FlowModAdd` message rate per tenant in the same manner, the impact of the flow rate and flow length is almost the same. Since all other parameters are the same (e.g., VN size), the slopes of both curves are indeed approximately equal: 0.0109 %/(*FlowMod*/s) for flow length and 0.0113 %/(*FlowMod*/s) for the flow

rate. Moreover, the flow request rate and the corresponding flow lengths can be used to determine the total number of translated `OF_FlowModAdd` messages by the hypervisor.

Physical and VN Size. As suspected, increasing the VN size does indeed affect the CPU utilization (see Fig. 3.16c). For instance, embedding 5×434 flows per second (5 tenants) with a length of 3 in a 2×2 virtual grid network generated CPU utilization of around 78 %, while repeating the same measurement in 5×5 grid lead to 136 % CPU utilization. Interestingly, the CPU utilization does not scale linearly with the total number of virtual switches but with its square root. Although this might sound counter-intuitive (as typically lookup scalings are $O(n)$ for a list or $O(\log(n)), O(n \log(n))$ for tree-like data structures), increasing the virtual grid dimension also increases the average node degree (i.e., number of virtual ports), which also has an impact of CPU utilization. Contrary to our expectations from Sec. 3.2.3.2, the size of the underlying physical topology does not produce a significant impact on the CPU utilization (see Fig. 3.16d). For instance, with 3 tenants, the observed CPU utilization increased only from $\sim$ 54 % (for 3×3 physical grid) to around $\sim$ 56 % (for 5×5 physical grid).

Number of Virtual Ports The previous measurements are based on the assumption that each tenant has one host connected to each virtual switch (requires one virtual port). Thus, coupled with topology knowledge, the total number of virtual ports per each tenant's switch is fully defined. However, other physical topologies and VN configurations can have a different number of virtual ports per switch. To incorporate the topology impact in our measurements, we investigate the impact of the number of virtual hosts/ports per switch. To this end, we increase the total number of hosts per tenant on each switch from 1 to 5, in turn increasing the number of virtual ports. We observe a non-negligible linear impact (see Fig. 3.16e). For instance, the observed mean CPU utilization for 5 tenants is increased from around 74 % (for 1 host per virtual switch) to around 93 % (for 5 hosts per virtual switch). This occurs since the NH inspects every `OF_FlowModAdd` in order to investigate if the tenants are indeed using only their corresponding *flowspaces*.

Number of Tenants. Fig. 3.16f presents all of the data from Figs. 3.16a–3.16e merged and sorted based on the total number of tenants. Furthermore, the plot shows the linear regression of the 90th percentile values for each number of tenants. We observe that the intercept of the regression line is around 0 %. Although the number of tenants in some cases could have a non-linear impact (see $p_i = 2$ on Fig. 3.16e), we also highlight that the deviation is not drastic. Overall (see Fig. 3.16f), for small number of tenants, the impact can be simplified and assumed additive. That means that the increase in CPU utilization observed for each additional tenant is always the same. In particular, that means that, for predicting the CPU utilization for a multi-tenant scenario, we can simply compute the impact of each tenant independently and add the resulting CPU values.

Summary. While CPU utilization is by nature a highly variable metric, it can be seen that the presented measurement procedure always obtains stable results. Moreover, it is shown that the impact of considered evaluation parameters on the 90th percentile CPU utilization can be approximated by simple functions (models). This suggests that a precise modeling of the 90th percentile of CPU utilization is indeed possible. Further, the size of the physical topology seems to have no major effect

on the resource consumption, hence, suggesting that the model can be physical-topology-agnostic for smaller sized networks. In the following part (see Sec. 3.2.4), the presented measurements and the observed scaling dependency are used in order to generate a CPU performance model capable of supporting arbitrary physical networks and a wide variety of VN requests.

3.2.4 Hypervisor CPU Prediction Model

In this part, based on the results of considered measurement campaign, NH CPU utilization prediction model is developed and presented. Again, the 90th percentile CPU utilization is used as it provides a good trade off between performance and resource consumption (see Sec. 3.2.3.4). Firstly, the comprehensive grid measurements are fitted with a linear model. Then, the model is extended with a port scaling factor in order to generalize its applicability to arbitrary physical topologies and randomly generated VNs.

3.2.4.1 Model for Grid Topologies

The results of Sec. 3.2.3.4 show that the CPU utilization of a FV scales with *(i)* the `OF_FlowModAdd` message rate (as witnessed by the impact of flow length and flow request rate), *(ii)* the total number of virtual switches, and *(iii)* the number of virtual ports per switch. Furthermore, for a small number of tenants, *(iv)* the impact of this parameter can be assumed to be additive (see Sec. 3.2.3.4): the contribution of each additional tenant to the CPU utilization is equal to its contribution as a single tenant. As the number of virtual ports is used to represent the topology impact (e.g., edge density), it is only considered when extending the model for arbitrary topologies (Sec. 3.2.4.2). The initial CPU prediction model is defined as:

$$f_1(r, v, n) = \sum_{i=1}^{n} \left(c_0 + c_1\, r_i^{FM} \sqrt{v_i}\right), \tag{3.11}$$

where n is the number of tenants, r_i^{FM} is the total CP `OF_FlowModAdd` rate, which is fully defined with the flow request rate and the corresponding path lengths, (i.e., if we assume that the total number of different paths of a tenant i is X_i, then $r_i^{FM} = \sum_{j=1}^{X_i} r_j^i \cdot l_j$). The variable v_i gives the total number of virtual switches of a tenant i, and c_0 and c_1 are fitting coefficients. We multiply the parameters (r_i^{FM} and $\sqrt{v_i}$) since the required resources for processing each `OF_FlowModAdd` message depend on the configuration of a VN (see Sec. 2.2.3 and Sec. 3.2.3.4). To illustrate, if no messages are sent ($r_i = 0$ or $l_i = 0$), the size of a VN (i.e., parameter $\sqrt{v_i}$) should not have a significant impact on the CPU utilization.

Using a regression model minimizing the mean square error and the measurement data presented in Figs. 3.16a–3.16c, the coefficients $c_0 = 6.46$ and $c_1 = 2.84 \times 10^{-3}$ are obtained. The cumulative distribution of the absolute fitting error and the corresponding median are shown in Fig. 3.17. The median of the absolute error is around 3.41 % and the maximum error is 12.4 % (*for scenario*: num. tenants = 5, $r = 430$, $k = 5 \times 5$, $v_i = 3 \times 3$, $l_i = 3$ and $p_i = 1$), witnessing the fitting accuracy.

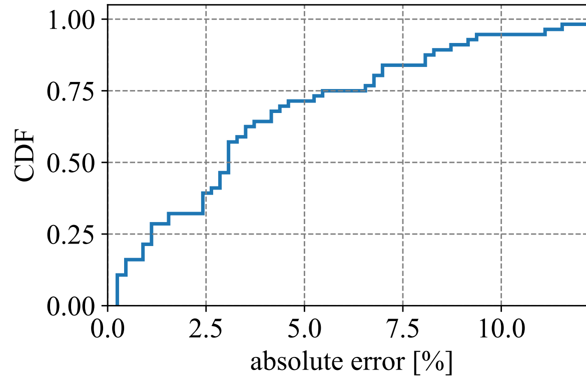


Figure 3.17: Cumulative distribution function (CDF) of the absolute fitting error of the grid measurement data with a linear based model (Eg. 3.11).

3.2.4.2 Model Extension for Arbitrary Topologies

The initial model f_1 (Eq. 3.11) assumes a physical grid topology as well as virtual grid networks. However, in practice, other topologies are possible. For instance, tree-like topologies used in data center networks, ring-like topologies used in industrial networks and wide-area network topologies such as those from the Topology Zoo [Kni11] have less dense structures. The average node degree of a ring topology is always 2, while the average node degree of a grid with 9 nodes is 2.67 [4].

We have seen that the number of virtual ports affects the NH CPU utilization (see Fig. 3.16e). As the average number of virtual ports of a VN directly corresponds to the average node degree of its topology, this notion is used to extend our model to arbitrary topologies. To do so, the grid-based CPU prediction model is linearly scaled with a per-tenant port scaling factor ϕ_i. A linear scaling factor ϕ_i is used because of the linear dependency observed in Fig. 3.16e.

The extended model can then be formulated as:

$$f_2(r, v, n) = \sum_{i=1}^{n} \phi_i \big(c_0 + c_1\, r_i \sqrt{v_i}\big). \tag{3.12}$$

The intuition for defining the per-tenant port scaling factor ϕ_i is similar to the cross-multiplication rule in elementary arithmetic. We first divide the per-tenant grid-based prediction (based on Eq. 3.11) with a parameter representing the average node degree of a grid topology (i.e., with p_i^e). This division "removes" the grid aspect in the prediction (details are in the next paragraph). Afterwards, again on a per-tenant basis, we multiply the newly obtained predictions with the average node degree of each tenant's virtual topology (i.e., with p_i). Mathematically, using a tuning parameter α, the per-tenant port scaling factor ϕ_i can be represented as:

$$\phi_i = \left(\frac{p_i}{p_i^e}\right)^{\alpha}, \tag{3.13}$$

where p_i is the average node degree of tenant's i VN:

$$p_i = \frac{\sum_{j=1}^{v_i} a_j^i}{v_i}, \tag{3.14}$$

where a_j^i is the number of virtual ports of virtual switch j.

[4] Excluding additional ports for connecting hosts to the virtual switches.

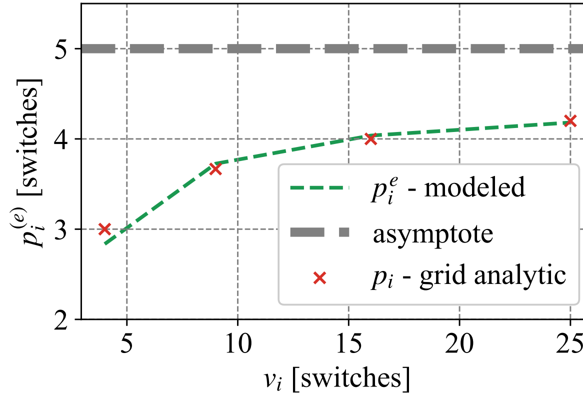


Figure 3.18: Computation of p_i^e based on v_i. Values are interpolated (green dashed line) from the average number of virtual ports per switch for grid topologies (red crosses) and converge to 5 (thick dashed gray asymptote) as internal nodes in a grid topology connect to 4 other nodes and one host.

Calculating p_i^e. The parameter p_i^e removes the grid aspect from the prediction. It tries to compute the average node degree of a grid topology with similar properties as the requested VN topology by a tenant. If a tenant requested a VN where the total number of nodes is a *square number*, i.e., the root of the total number of virtual switches is an integer, i.e., $y_i = \sqrt{v_i}, y_i \in \mathbb{Z}^+$ (e.g., $\sqrt{4} = 2$), an equivalent grid VN is simply $y_i \times y_i$ grid. Thus, it is easy to calculate the average node degree of the corresponding grid VN equivalent. For instance, if a tenant requested a VN with $v_i = 4$ nodes, the grid-like equivalent is a 2×2 grid, which has 12 virtual ports, thus $p_i^e = 12/4 = 3$.

If the total number of nodes of a requested VN is not a square number, we determine p_i^e based on the linear fitting of the average node degree of various VN sizes which were used in the measurements (i.e., 2×2, 3×3, 4×4, 5×5). The fitting is shown in Fig. 3.18. The red crosses correspond to the grid topologies used in the measurements. In this case, p_i^e simply corresponds to the average number of virtual ports per virtual switch for a grid topology of dimension $\sqrt{v_i}$. For intermediate values, since there is no direct grid-like VN equivalent, we simply fit the total number of virtual ports in the network piece-wise linearly and divide it by the number of nodes. This is shown by the green dashed line. Note that the values used for linear fitting are based on our measurement scenarios, which means that each virtual node also has an additional virtual port for connecting one virtual host to it. That is why the p_i^e values converge to 5, as shown by the thick gray dashed line.

Calculating α. As observed in Sec. 3.2.3.4, increasing the number of virtual ports also increases the observed CPU utilization linearly. However, the increase is not directly proportional, i.e., doubling the average number of virtual ports does not double the observed CPU utilization. Therefore, we introduce an exponential tuning parameter α, and we use it to improve the fitting of our scaling parameter ϕ_i to our measurements. To calculate α, the measurement data from Fig. 3.16e is used and α is derived while minimizing the mean square error. We obtain $\alpha = 0.281$. Hence, replacing parameters in Eq. 3.12 with their real values, we obtain the final model.

Table 3.4: Physical topologies considered and their number of nodes and edges and their density. As a comparison, 5×5 and 6×6 grids have densities of 1.6 and 1.67 respectively.

Topology	# Nodes	# Edges	Density
Ring30	30	30	1
Internet2	34	42	1.23
NobelEU	28	41	1.46
Watts-Strogatz	30	60	2
Erdos-Reny15	30	63	2.1
Erdos-Reny30	30	136	4.53

Table 3.5: Distribution of parameters for the final evaluation. $U(x, y)$ denotes a uniform distribution between x and y.

Scenarios	VN size [#nodes]	Flow rate [1/s]
1 – 20	$U(2, 25)$ – *Full*	$U(200, 600)$ – *Full*
21 – 40	$U(13, 25)$ – *Big*	$U(400, 600)$ – *High*
41 – 60	$U(2, 13)$ – *Small*	$U(400, 600)$ – *High*
61 – 80	$U(2, 13)$ – *Small*	$U(200, 400)$ – *Low*
81 – 100	$U(13, 25)$ – *Big*	$U(200, 400)$ – *Low*

3.2.5 Model Evaluation

Having established the model, we now evaluate its accuracy using topologies and VNs that were not used during the measurements. This evaluations should quantify how the model can adapt to arbitrary physical topologies and randomly generated VNs. First, we introduce the evaluation scenario and explain how to generate a wide variety of different VN requests (Sec. 3.2.5.1). Second, the baseline models (used for comparison) are introduced in Sec. 3.2.5.2. Finally, in Sec. 3.2.5.3, the precision of the CPU prediction model is evaluated. Afterwards, the effect of provisioning the resources with the proposed model on the NH processing latency is studied.

3.2.5.1 Scenario

Topologies. Six different physical topologies (Tab. 3.4) are considered in the evaluations: two existing wide-area network topologies (Internet2 [Hoc+13] and Nobel EU [Orl+07]), a typical industrial network topology (a 30 node ring) and three randomly generated network topologies: two Erdos-Reny [ER59] models with 30 nodes and an edge probability of 15 % and 30 %, and a Watts-Strogatz [WS98] model with 30 nodes with an average degree of 4. The selected topologies have a wide range of different edge densities (defined as the number of edges divided by the number of nodes), supporting the choice of these topologies as a representative set.

VN Requests. The total number of tenants during one run is static, and it ranges from 2 to 5. For each topology, and for each number of tenants, 100 different measurement scenarios are defined. One measurement scenario is generated based on the *(i)* per-tenant VN size distribution and, *(ii)* a per-tenant flow request rate distribution. Depending on the scenario number, different parameter values are used, as listed in Tab. 3.5 (as in Sec. 3.2.3). This is done since using the full ranges as in the measurement section will lead to VN requests with different requirements. However, in multi-tenant cases, the average of all requirements of all tenants converges to expected mean values. Therefore, in order to evaluate more extreme cases, we consider cases for which all tenants may request only *big* or *small* networks, and only *high* or *low* flow request rates (as in Tab. 3.5). As a consequence, four different combinations are built, making a total of 5 different cases, with 20 scenarios each.

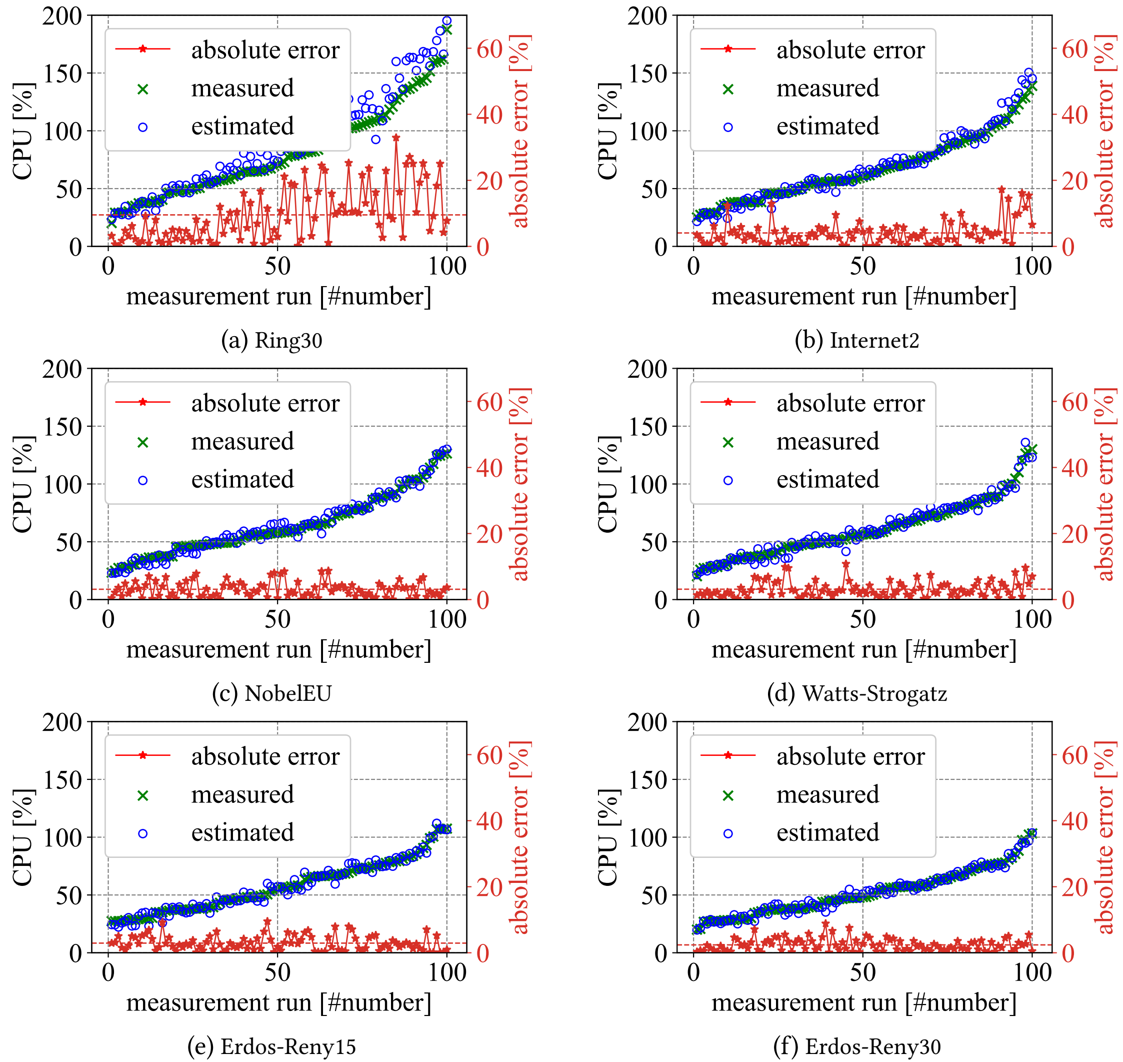


Figure 3.19: (a)–(f) Estimated and measured 90th percentile CPU utilization for 100 randomly generated scenarios with 2-5 tenants using the considered topologies (i.e., Internet2, NobelEU, Ring, Erdos-Reny and Watts-Strogatz). The scenarios are sorted based on the measured CPU utilization in the ascending order. The absolute error is shown as red starred line, while the mean observed error is shown as horizontal red dashed line.

VN Connectivity. Based on a given VN size, edges in the VN are generated as follows. One node is randomly selected. Out of all its neighbors, a new node is randomly selected and connected. Out of all the neighbors of existing nodes, a node is randomly selected and connected. The procedure stops as soon as enough nodes are connected. Finally, the per-tenant flow request rate is generated based on the values in Tab. 3.5. The same flow generation procedure is used as presented in the measurement section.

3.2.5.2 Model Variations & Baseline

State-of-the-art VNF CPU resource prediction models generate their estimate while considering different input parameters [JKE17]; [MAC18]. However, apart from the message rate, these parameters (e.g., IP source address, TCP destination port) often do not have an impact on the CPU utilization of an NH. Thus, if we directly apply these approaches for provisioning the CPU resources of an NH, they would generate their estimate solely based on the CP message rate. Similarly, *state-of-the-art* NH CPU performance models [Sie+16]; [SOK17]; [Đer+18] only consider the total CP message rate as their input parameter for estimating the CPU utilization of an NH. Therefore, as the baseline, a linear model based on the total CP message rate [Sie+16] is used. This model does not take into account the properties of the VNs. The baseline model is defined as follows:

$$f_3(r, n) = c_o^B + c_1^B \sum_{i=1}^{n} r_i. \tag{3.15}$$

The coefficients c_0^B and c_0^B are obtained by a least mean square error fitting using the same measurement data set presented in Fig. 3.16; the coefficients are $c_0^B = 11.96$ and $c_0^B = 13.86 \times 10^{-3}$.

Furthermore, in order to evaluate separately the effect of the number of virtual switches and ports, we also consider our original f_1 model, which does not include the effect of virtual ports.

3.2.5.3 Evaluation Results

First, the accuracy of the proposed model is evaluated. It is shown that, even for arbitrary topologies and randomly generated VNs, the prediction error remains low (8–9 % on average). Finally, the provisioning performance of the proposed prediction model, and the impact on the processing latency is studied. Furthermore, we also compare the achieved accuracy and provisioning performance to the two baseline models.

Prediction Accuracy. Fig. 3.19 shows, for each physical topology, the measured and predicted CPU utilization for 100 randomly generated scenarios. It can be observed that the proposed model accurately predicts CPU utilization, even for randomly generated virtual topologies and unknown physical topologies. For instance, the mean absolute prediction errors per topology falls between 2.4–9.6 %, while the highest observed error is over 30 %. The maximal absolute error is observed for the ring topology, as it exhibits the most different characteristics compared to the grid (e.g., significantly lower edge density). Due to this difference, the maximal measured CPU values for the ring topology exceeded the maximal grid-based values, which were used for model fitting (more detailed explanation is in the next paragraph). For example, for the ring topology, we observed CPU

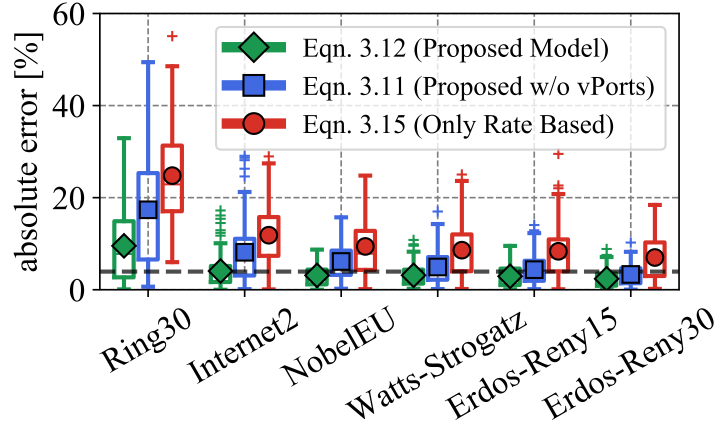


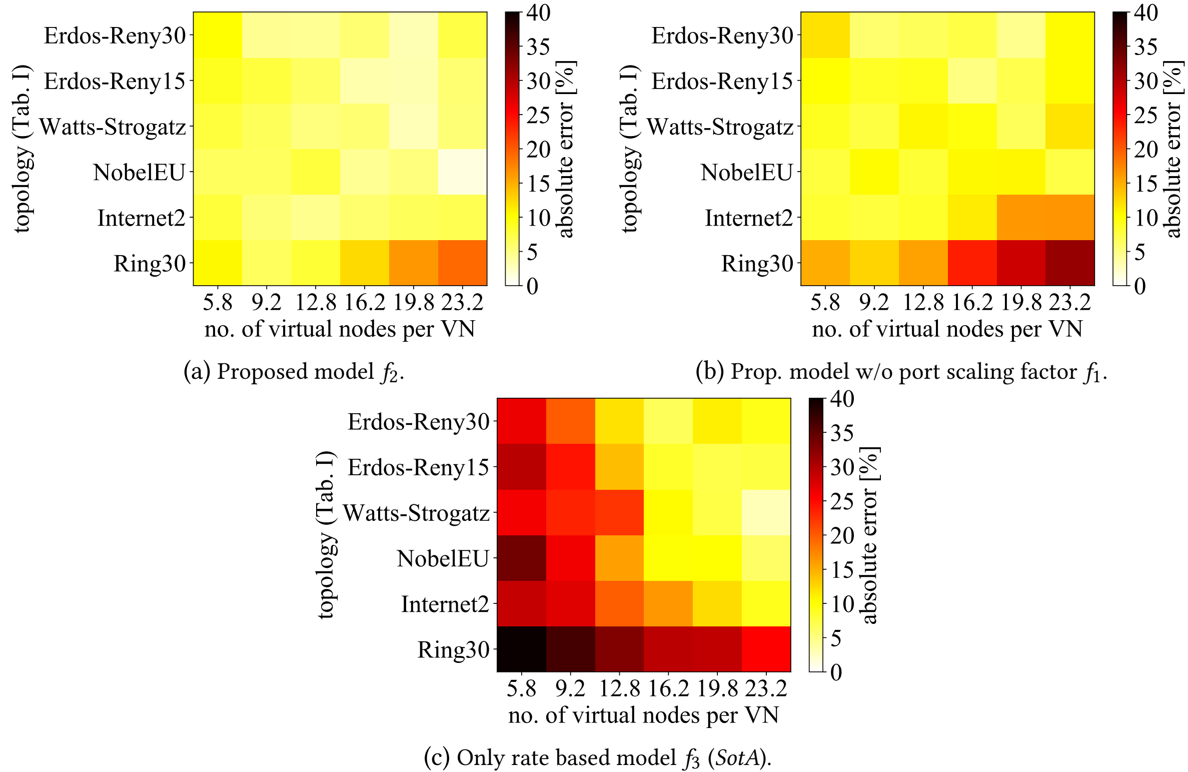
Figure 3.20: Box plots of the prediction accuracy of the three models f_2 (Eq. 3.12, proposed model), f_1 (Eq. 3.11, proposed model without port scaling factor) and f_3 (Eq. 3.15, only rate based model - *SotA*). The dashed gray line indicates the average fitting error of the grid measurements, thus, applying the model on different topologies with randomly generated requests on average introduces a slight penalty, i.e., the average error increases by a few percents.

values reaching up to 200%, while in the case of a grid topology the maximal values were around 150%. Thus, the predicted values were outside of the modeled range, thus the performance suffered the most in this case.

Topology Analysis. We observe that different topologies require different amount of CPU resources. For instance, *Ring30* requires the highest amount of CPU resources (up to $\sim$ 180 %) while *Erdos-Reny30* requires the least (up to $\sim$ 110 %). This can be explained in the following way: randomly generated VNs on physical topologies with a lower edge density (e.g., ring) typically have longer paths compared to the ones generated on more dense topologies. For instance, a VN with 4 virtual switches on a ring topology is always a line with a maximal path length of 4. On the other hand, a grid VN with the same amount of nodes (i.e., 2×2) on a physical grid topology has a maximum path length of 3. Therefore, for less dense topologies, a larger number of `OF_FlowModAdd` messages are needed, in turn leading to higher CPU utilization. Thus, the observed and predicted mean CPU utilization values in Figs. 3.19b–3.19d are highly correlated with the density of the corresponding topologies (see Tab. 3.4).

3.2.5.4 Mean Error and Baseline Comparison

Fig. 3.20 shows the prediction error of the proposed model f_2 (Eq. 3.12) and of the two baseline models f_1 (Eq. 3.11, proposed model without port scaling factor) and f_3 (Eq. 3.15, only rate based), all models predict 90th percentile CPU utilization. The highest absolute errors are observed for the ring topologies as it differs the most from the measurement grid topology. For instance, the mean prediction error achieved for the ring topology is slightly higher compared to other topologies, i.e., around 9.5 % compared to 2.4 % – 4.1 % for other topologies. Furthermore, the baseline prediction models (i.e., Eq. 3.11, and Eq. 3.15) produce considerably higher prediction errors, as the mean prediction per topology varies between 3 % and 25 %. This confirms our original motivation: certain NH functions



(a) Proposed model f_2.

(b) Prop. model w/o port scaling factor f_1.

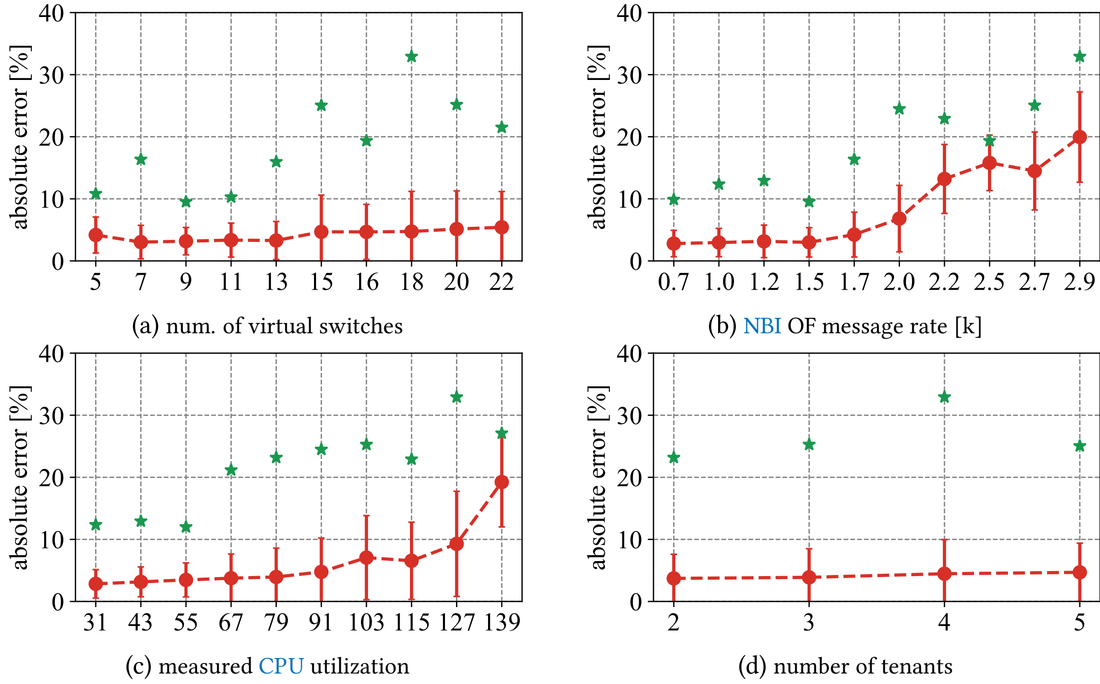
(c) Only rate based model f_3 (*SotA*).

Figure 3.21: Impact of the number of virtual switches per tenant and topology type on the mean observed relative error for (a) proposed model and (b) the baseline. The date is separated in 6 uniformly created bins, and the mean values of each bin are shown.

depend on configurations of the tenant VNs — processing the same message in different network settings requires a different amount of resources.

Sources of Error. Fig. 3.21 shows heatmaps of the measured absolute error for the proposed model and the baseline models for all topologies and for different VN sizes. The proposed model performs quite constantly, having an overall mean absolute prediction error of around 4 %. As the baseline models do not include all affecting parameters (i.e., total number of virtual switch and ports), they fail to perform in extreme cases. For instance, the rate baseline model (Eq. 3.15, only rate based) reaches a mean absolute error of 40 % for small networks, which can lead to significant unpredictability of the performance perceived by tenants.

As the baseline (Eq. 3.15, only rate based) was fitted based on the grid measurement data where VN sizes vary from 4 to 25 virtual switches, we would expect that it performs the best for middle sized VNs (e.g., with around 12-13 nodes). However, as it can be seen in Fig. 3.21c, this is not the case, as the baseline performed the best for virtual topologies with a higher number of nodes. This stems from the fact that (on average) our tree-like VN generation procedure produces VN requests with a lower amount of virtual ports compared to the grid VNs used in the measurement section. This makes CPU prediction of the baseline higher for all VN sizes, hence, the precision becomes worse for middle-sized VNs, and the best for larger VNs (see Fig. 3.21c). The proposed model includes a per-tenant port scaling factor, hence, it can in general mitigate this effect among all VN sizes.



(a) num. of virtual switches

(b) NBI OF message rate [k]

(c) measured CPU utilization

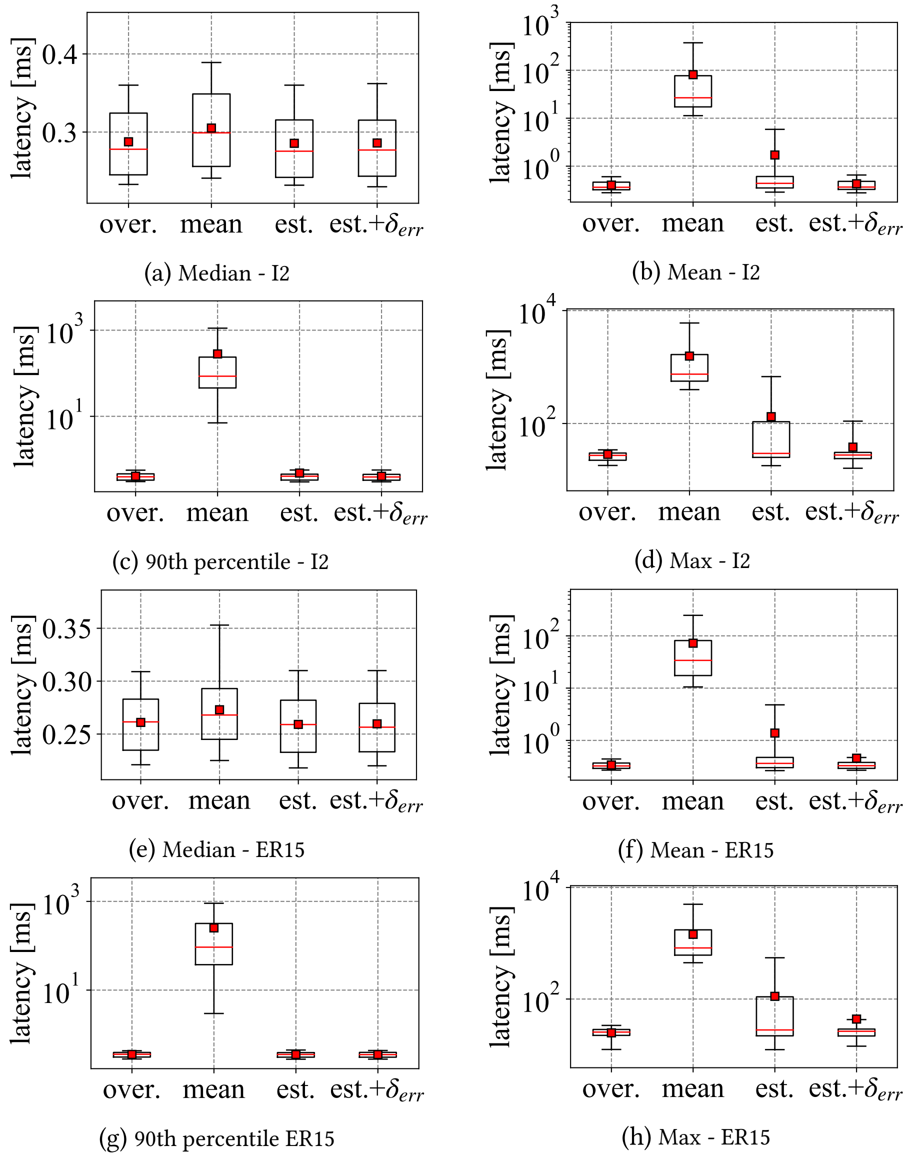
(d) number of tenants

Figure 3.22: Absolute error dependency on (a) the average number of virtual switches per VN, (b) the total `OF_FlowModAdd` rate, (c) the measured CPU utilization, and (d) the total number of tenants. The observed data on figures (a)–(c) is pre-processed by binning it into 10 equally sized bins. For each bin, mean and standard deviation are shown as error bars, while green stars indicate the corresponding maximum.

Overall, the rate-based solutions can perform decently in static and fixed environments, i.e., with fixed number of static tenants (and fixed VN configurations) generating constant CP load. However, novel communication networks are envisioned to be dynamic and flexible, i.e., a tenant should be able to request arbitrary VNs (with a desired configuration) at any time for a certain duration. In contrast to the proposed solution, *state-of-the-art* approaches do not include all the crucial parameters (e.g., the error can reach over 60% depending on VN sizes), therefore, these solutions do not seem suitable for flexible and dynamic networks.

Effect of Evaluation Parameters The proposed CPU prediction model depends on several input parameters: number of tenants, total number of virtual switches, the flow request rate, and the number of ports per virtual switch for each tenant. In order to evaluate whether the performance deviates with some of the considered parameters, the data is uniformly binned, and the mean absolute prediction error along with standard deviation and maximum value is shown in Fig. 3.22. Overall, it can observe that the mean and maximal absolute error is higher when the measured CPU utilization is higher. There are two reasons behind this. Firstly, the absolute errors are shown, thus an absolute error of 10% at 100% is more highlighted on the figure compared to an absolute error of 5% at 50% (while relative errors are the same). Secondly, in case of a higher CPU utilization, the ring topology exhibits significantly higher absolute error compared to the other topologies (e.g., see Fig. 3.19a and Figs. 3.19b-3.19f). Thus, this significant error increase caused by extrapolating skews a bit data on Fig. 3.22. Further, since all these parameters are correlated (e.g., higher rates produces higher CPU utilization), the same trend can be observed for all parameters except the number of tenants.

3.2.5.5 Impact on Latency



(a) Median - I2
(b) Mean - I2
(c) 90th percentile - I2
(d) Max - I2
(e) Median - ER15
(f) Mean - ER15
(g) 90th percentile ER15
(h) Max - ER15

Figure 3.23: (a)–(f) Impact of 4 different NH CPU provisioning strategies on the observed message processing time of an NH. For each run, four different statistical properties are considered: median, mean, 90th percentile, and maximal latency. As we have 400 different scenarios, each box plot represents 400 observed values of a certain statistical property.

To evaluate the impact of CPU provisioning on the processing latency of the NH, 100 scenarios are generated per total number of tenants for Internet2 and Erdos-Reny15 topologies in the same manner as explained in Sec. 3.2.5.1. Each measurement scenario is then run 4 times with 4 different provisioning strategies while tracking the processing latency profiles. The following 4 provisioning strategies are considered.

1. *Over-provisioning.* All physical resources are dedicated to the NH. This strategy is very resource inefficient but is expected to provide the best overall message processing and forwarding performance.

2. *Mean CPU Estimate.* The resources are provisioned based on the observed mean CPU utilization during the over-provisioned run. This corresponds to the *state-of-the-art* CPU performance models [Sie+16]; [SOK17].

3. *Proposed Model.* The resources are provisioned with the proposed improved model, i.e., with f_2 (Eq. 3.12).

4. *Proposed Model with Additional Margin.* As the proposed model is not perfect (average fitting error is around 4%), it occasionally underestimates the CPU utilization, in turn, potentially increasing the processing time of an NH. In order to compensate for underestimation and non-perfect *cpulimit* precision, we also consider provisioning strategy based on the proposed model with an additional margin. For an additional margin we use the maximal fitting error, which is 12.4 % (see Fig. 3.17). Thus, the resources are provisioned with f_2 + 12.4 %. This approach ensures that the resources are always provisioned with a slightly higher value, thus, the impact on latency should always be negligible.

For each measurement run, the mean, median, maximal, and 90th percentile latencies are recorded. Fig. 3.23 shows the latency profiles achieved by the strategies for the two topologies and the four aforementioned statistical properties. As already shown in the introduction, provisioning with the mean CPU estimate (2nd set of box plots) is not sufficient as it incurs a big overall latency increase. Provisioning with the proposed model (3rd set of box plots) does not increase the median latency, however, the mean is increased as the maximal latencies are increased significantly. For instance, the maximal latency increases around 3 times, reaching values of around 100 ms. This is happening when the model underestimates the required CPU resources for a given measurement scenario. Thus, in certain time instances, the NH exceeds the allocated CPU resources, in turn triggering *cpulimit* to throttle the corresponding process.

Provisioning the CPU resources with an additional margin (4th set of box plots) achieved almost the same processing performance as the over-provisioned case. For instance, the mean maximal latencies increased only from 24-29ms to 38-44ms. The mean and median latencies stayed the same. We can conclude that provisioning with an additional margin provides huge resource savings (in the most naive case 710 % less CPU capacity) while having an acceptable and still predictable impact on the processing performance for virtual network tenants.

3.2.6 Insights and Discussion

Correctly provisioning the resources available to an NH is crucial for ensuring stable and predictable network performance for tenants. Yet, the *state-of-the-art* does not offer adequate solutions as they neglect bursty NH workloads or the impact of dynamic VN changes. Thus, in this section, with the goal of provisioning NH resources for flow embedding scenario, an accurate CPU prediction model is presented. It is based on the comprehensive measurements and it can be generalized to different substrate topologies and virtual network requests. The proposed model exhibits high prediction accuracy as the mean average prediction error is overall around 4 %. Provisioning the resources with the corresponding model produces only a slight increase of tail latencies. For instance, on average, the maximal processing time increased from around 25ms to around 44ms. With the presented model,

it becomes possible to minimize the resource consumption or improve the overall utilization through accurate prediction of the required CPU resources of an NH.

3.3 Demonstration of Application-Aware NH Reconfiguration

In the previous sections, the impact on various CP policies (e.g., topology abstraction) or DP configuration parameters (e.g., topology size) on the CP performance is presented (e.g., NH processing time). However, in the following, we take a look from the other side. That is, in this section (i.e., in Section 3.3), the impact of different VN embedding and VM placement strategies (i.e., CP decisions) on the DP performance (i.e., *end-to-end* latency) of a remote control application is demonstrated. Optimizing the DP performance is especially important in telesurgery scenarios [Mun21] (i.e., remote surgery). To demonstrate it, one *state-of-the-art* VL [BBK15] is extended to support VN reconfiguration during the runtime based on the tenants requirements [Đer+19]. Furthermore, the corresponding VL is then deployed in the physical testbed in order to virtualize the emulated DP network. Additionally, one humanoid robot and a remote controller are connected to different end-points in the emulated network. In this demo, it is shown that by optimizing the network (e.g., VN embedding and/or VM placement strategy) it is possible to improve the experienced quality of remote control.

This section is organized as follows. Sec. 3.3.1 presents the demonstration setup and the considered scenario. Sec. 3.3.2 presents the demonstration story line. Sec. 3.3.3 provides a discussion regarding the presented demonstration.

3.3.1 Demo Setup and Scenario

The architecture of the demonstration is presented in Figure 3.24, where the CP and DP separation is illustrated. Starting from top to bottom, two virtually isolated software instances of the Ryu SDN controller software [Com17] are used to emulate SDN controllers belonging to two tenants.

It is considered that HyperFLEX [BBK15] management layer virtualizes the deployed SDN-based DP network. HyperFLEX provides CP and DP isolation, QoS monitoring, and automated management of VNs. In this demo, it is extended to further support the dynamic reconfiguration of VNs. That is, if a tenant requests an additional virtual path or reconfiguration of the already existing one, HyperFLEX firstly checks the available link (e.g., bandwidth) and node resources (e.g., CPU) along the whole physical path. If there are enough resources, it establishes new virtualization (embedding) policies accordingly. If there are no available paths that satisfy the required QoS demands, HyperFLEX checks whether reconfigurations of the other VNs could free the demanded resources. Furthermore, all the isolation and abstraction policies are updated in order to meet the new VN requirements and in order to prevent performance interference among tenants. The VN reconfiguration process is completely abstracted (i.e., hidden) from the tenants.

As illustrated in the lower part of Fig. 3.24, European-based DP network is used. It spans four countries (i.e., Finland, France, Sweden, and Germany). Most of the major cities are interconnected and the network is emulated using mininet [De +14].

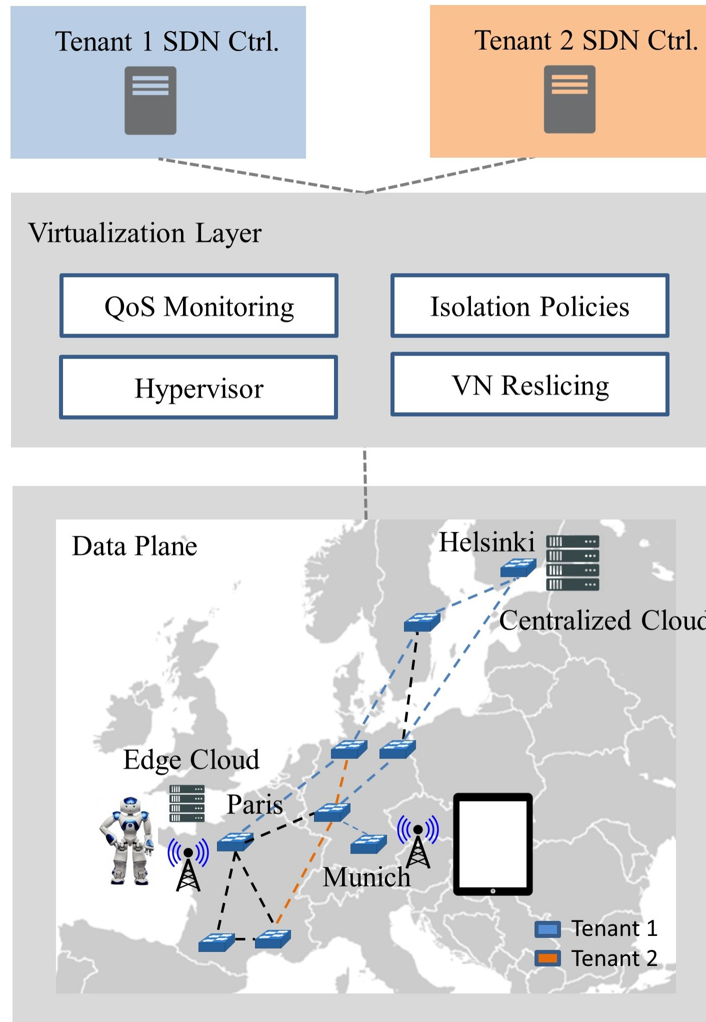


Figure 3.24: The demonstration architecture and scenario show *i)* a European-based data plane topology with a centralized server located in the proximity of Helsinki, Finland, while the edge servers are positioned in the proximity of big cities and *ii)* the virtualization provided by the virtualization layer, i.e., HyperFLEX, and *iii)* two tenants that share the physical architecture, highlighted in blue and orange colours.

We consider that two tenants are sharing the physical network. Tenant 1 is assumed to be a small private data center operator which owns one centralized server for the consolidation of services located in Helsinki, Finland (the remote location), and multiple edge servers located in the proximity of the most major cities: e.g., Paris, Munich. Tenant 1 initially requests a VN spanning multiple links and nodes as depicted in Fig. 3.24. It uses this VN to control his humanoid NAO robot [Zac+18] located in Paris from a remote controller located in Munich (with the controller application running on Apple iPad). Two Linksys WRT1900AC WiFi access points are installed to provide the access to the virtual network, connecting the robot and the remote controller to the backbone network. In order to control the robot, two streams need to be established. The NAO robot generates the first traffic stream in order to provide a live video feed of the exact robot position to the remote controller. Based on the received information, the remote controller issues control instructions to the robot using the second traffic stream. However, before reaching the destinations, both traffic streams have to pass through the firewall VNF, which is initially located in Helsinki. For this demonstration, an internally

developed virtual firewall is used as a VNF in order to prevent malicious traffic entering the network. Tenant 2 is generating dummy cross traffic.

3.3.2 Demo Presentation

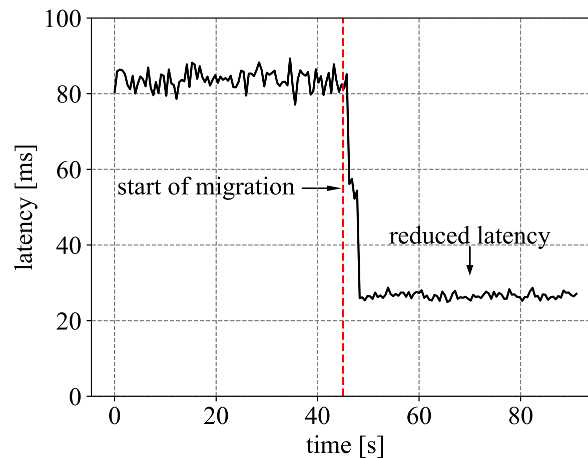


Figure 3.25: Networking delay between the robot and the remote controller based on the given demonstration topology. During the first 46 seconds (left side of the red dashed line), the firewall VNF is located on the centralized server. The right side of the red line shows the network latency after the migration.

This demonstration focuses on presenting two main features: VNF Orchestration and Dynamic VN Reconfiguration.

VNF orchestration. Controlling the robot remotely based on the real-time video streaming is a highly sensitive application. Moreover, the additional delay coming from consolidation of services in a centralized server can potentially harm the performance of the application. In case the QoS requirements of the remote control application are not met, a firewall migration is triggered by the tenant. The firewall is then migrated to the edge server in Paris. The migration of the firewall VNF should reduce the end-to-end delay and improve the experienced QoS. Fig. 3.25 shows the reduction of the network delay between the NAO robot and the remote controller before and after the VNF migration over time.

Dynamic VN Reconfiguration. After the VNF migration is triggered, HyperFLEX reconfigures the VN in order to meet the new requirements. The virtual links towards the centralized location in Helsinki are now redundant and can be removed, while the route connecting the edge servers in Paris and Munich is established. Additionally, isolation (e.g., bandwidth) policies spanning the used physical links between Munich and Paris are updated. The reconfiguration also ensures that the new VN requirements generated by the firewall migration do not affect other VNs.

3.3.3 Insights and Discussion

The benefits of softwarization and consolidation of services are numerous. However, there are some parameters that should be considered: (1) locations far away from users can increase the networking delay, and (2) the processing time of software realizations of the networking functions is inherently

higher compared to the hardware ones. Thus, VNF placement and management algorithms have to take service QoS requirements into the account. In this demonstration, we showed the possible performance gains by migrating a softwarized VNF (i.e., a Linux-based firewall) from a centralized server to an edge server (closer to the user). In order to demonstrate the performance improvements, we supposed an application with a high QoS requirements: the remote control of a NAO robot over a secure network. We showed how the reduced latency is able to improve application quality greatly. Furthermore, we presented and discussed how the VNF orchestration should be realized in a virtualized SDN environment. In the considered use case of this demonstration, the locations of the robot and the controller were static, since they were always connected to the same access points. As a future work, we plan to evaluate dynamic use cases like commercial aviation. Here, the positions of airplanes and the corresponding communication access points are well known in advance. Thus, including more granular and mobility-aware VNF migrations within the network could improve the performance and reduce the total operational cost. Moreover, a vehicular use case represents an even more challenging problem, as the movement of vehicles might not be predictable. Hence, more dynamic reconfigurations of VNs might become necessary for an efficient resource utilization.

3.4 Conclusion

In the first part of this chapter, the impact of the topology abstraction function on the NH CPU utilization is studied and measured in various scenarios (e.g., different topology sizes). The observed results reveal that the impact of such function is not negligible. That is, the maximal observed difference in CPU utilization was almost 400% for two vastly different topology abstraction policies (i.e., *transparent* and *big-switch* abstraction). This motivated us to study and present the impact of topology abstraction on the performance of offline VNE problems. Two important conclusions can be drawn from this study. Firstly, NH functions cannot be neglected neither when *i)* provisioning the CPU resources of an NH or when *ii)* designing the VN admission control algorithms. Secondly, the CPU utilization of an NH can vary significantly based on the considered DP scenario (e.g., topology size). Therefore, it became evident that determining how much resources (e.g., CPU) should be allocated (i.e., provisioning problem) to an NH while ensuring that there is no performance degradation is not a trivial task. For instance, under-provisioning the resources may lead to performance degradation, while over-provisioning is wasteful.

These observations motivated the development of the main contribution presented in this chapter, i.e., the QoS-aware network hypervisor provisioning procedure. To do so, initially, a comprehensive measurement study in various scenarios is performed to *1)* determine how much resources should be allocated to an NH (in a specific scenario) while ensuring that there is no performance degradation, and *2)* to develop an accurate CPU estimation model. The model can be generalized to different substrate topologies and virtual network requests. Additionally, it exhibits high prediction accuracy, as the mean prediction error is around 4%. Provisioning the resources of an NH with the presented model produces only a slight increase of tail latencies. For example, the maximal processing latency is increased from 25ms to 44ms, while the average values remained the same. Therefore, by using the presented provisioning procedure, network operators can maximize their overall resource utilization while keeping the performance degradation to a minimum.

From a more general point of view, the major contribution of this section sheds light on potential other applications where similar predictions are necessary. For example, softwarization of networking functions (e.g., firewall, Network Address Translation (NAT)) and new networking architectures (e.g., SDN) increase the total number of deployed VNFs running in software on commodity hardware. Applying the same procedure to learn the performance profiles of various VNFs could potentially produce enormous savings while provisioning the resources in a centralized cloud where these functions run.

3.5 Future Work

The presented topology abstraction aware CPU estimation models and the formulated VNE optimization problem are based on the measurement results of OVX NH. However, different NHs could implement topology abstraction differently, which could produce different measurement results and in turn different observations. Therefore, developing automated benchmarking programs and solutions are needed to investigate all the different NH implementations. Additionally, the considered proposed estimation models are either linear, quadratic, or polynomial. One potential research direction is to investigate if the other models (e.g., Machine Learning (ML)- or Artificial Intelligence (AI)-based models) could achieve better results.

As presented in the main contribution of this chapter, it is possible to provision the resources of a VNF (i.e., in this case NH) based on the comprehensive measurements. However, manually learning what parameters have an effect, and which are the most affecting ones is tedious work, and it is time-consuming. Since the amount of required resources for a normal operation of VNF may be affected by many different parameters (e.g., number of clients within a firewall-protected network, number of flows a NAT has to process, etc..), this approach does not scale well to a large scale environment. Therefore, new solutions are needed that are capable of learning the most affecting parameters in an automated and online manner. To achieve so, novel AI-based systems and algorithms could pose as an attractive solution.

Chapter 4

Traffic Policing

In order to provide *end-to-end* deterministic Data Plane (DP) guarantees, most of the *state-of-the-art (SotA)* systems [VK16]; [Van+19b]; [Van+20] utilize mathematical frameworks (e.g, Deterministic Network Calculus (DNC) [LT01]) which rely on precise and accurate network state models. For instance, the network state model used in *Chameleon* [Van+20] keeps track of all the embedded flows (and their characteristics such as rate and burst) in the network along with the network utilization (which contains the statistics such as link utilization and switch buffer usage). In the interest of ensuring that the decisions derived by mathematical frameworks are valid, it is crucial that the network state models precisely matches the real traffic in the physical network. If that is not the case, unexpected and unaccounted for traffic bursts entering the network could use the available bandwidth, build up queues, and potentially cause unexpected packet losses and hence degraded performance. To avoid that, network operators usually rely on traffic policing or shaping at the edge of the network, typically performed directly on the edge switches or end-hosts [Van+20]; [Van+19b]; [Gro+15]; [Sae+17]. Furthermore, traffic limiting is even more important in virtualized networks, because of the two following reasons. Firstly, the users (sources of traffic) are not controlled, thus, they can try to generate more traffic than expected or agreed. Secondly, the unexpected degraded performance can lead to the violation of Service-Level Agreement (SLA) of the other network users, which can cause revenue loss to network operators.

Hence, this chapter aims to investigate and answer the following questions. *i)* How can traffic policing be realized or utilized on both, edge switches and end-hosts? *ii)* How can we measure and model the accuracy of the considered traffic policing solution? *iii)* And, what is the offered accuracy and precision of the considered solutions, and can we use them in *deterministic networks*?

Even though traffic policing is a well-known feature of network switches, the extent to which *SotA* programmable switches perform this task accurately and predictably has not been investigated in the literature. Motivated by this observation, this chapter presents traffic policing measurement procedure followed by an extensive measurement study of the policing performance of five Open Flow (OF) switches from three different vendors. We investigate the processing time overhead induced by configuring policing on the switches and quantify the policing accuracy and predictability in terms of the burst and rate parameters typically used for modeling traffic patterns.

Our observations are rather negative: none of the investigated switches perform policing accurately. While policing seemingly does not impact the processing time of switches, we find that the

policed traffic deviates by up to 100% in terms of allowed burst and 60% in terms of rate for some configurations. Even more concerning, we observe that switches do not support the configuration of arbitrary burst values. Some of the considered switches require a minimum burst size (e.g., 13 kbits for *Pica8* switches), while the others do not support the configuration of a burst and only perform rate-based policing.

With the assumption that these inaccuracies can be perfectly modeled, we investigate how these would impact the performance of *SotA* solutions for providing predictable latency in programmable networks. Using the open-sourced code of the *Chameleon* system [Van21]; [Van+20] for cloud networks, we quantify how much these inaccuracies impact the number of flows the system can accommodate while still providing its predictability guarantees. Astonishingly, we observe that *Chameleon* reduces the number of tenants it accepts by around 50% compared to a situation where switches are deemed perfect. This means that the sole limitations of the policing feature of switches force operators to see their revenue halved. The potential solutions and alternatives to circumvent the aforementioned limitations are also discussed and presented.

To evaluate end-host based traffic policing, we initially present an overview of one Data Plane Development Kit (DPDK) application which includes traffic policing functionality. Afterwards, we discuss and present the implementation of this traffic policing function in details. The performance observations regarding the considered software traffic policing function are rather positive. Even though the function was running on a general computing platform, the observed performance was mostly constant, and predictable in the considered scenarios. Thus, proving that in certain use cases software traffic policing approaches can pose as a valid alternative approach.

The contributions of this chapter are partly presented in the two following peer-reviewed scientific papers.

- N. Đerić et al. "Towards Understanding the Performance of Traffic Policing in Programmable Hardware Switches." In: *IEEE International Conference on Network Softwarization (NetSoft)*. 2021
- A. Van Bemten et al. "Chameleon: Predictable Latency and High Utilization with Queue-Aware and Adaptive Source Routing." In: *Proceedings of the ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*. 2020, pp. 451–465

All the content from the first peer-reviewed paper is presented in this chapter (i.e., in Sections 4.3,4.4, and 4.6) and it is one of the major contributions of this thesis. The second peer-reviewed paper introduces *Chameleon* system which is able to provide deterministic guarantees in cloud networks. As already explained in Sec. 2.3.3, to realize this system, we have developed one *end-host* DPDK application which includes traffic policing function. The main concepts of *Chameleon* system (e.g., architecture, admission control) and an overview of DPDK application are already presented in one dissertation [Bem+20]. However, the comprehensive implementation details (including the implemented algorithm and discussion) of this traffic policing function and the performance comparison of it with one *carrier-grade* switch were not presented and discussed before. Therefore, these two parts represent the main novel contributions of Sec. 4.5, and they are presented in Sec. 4.5.1.1 and 4.5.3.

The rest of this chapter is organized as follows. Section 4.1 presents the background on network predictability and justifies the need for accurate traffic policing. The related work is summarized in Section 4.2. Section 4.3 introduces the measurement methodology and procedure which enables modeling the traffic policing accuracy. After that, in Section 4.4, the accuracy of traffic policing in *carrier-grade* switches is discussed. Section 4.5 presents the details of one end-host based traffic policing implementation and the corresponding measurement results. Section 4.6 presents a study showing the impact of inaccurate traffic policing on maximal network utilization. Finally, Section 4.7 concludes this chapter, while Section 4.8 lists potential future works.

4.1 Background

In this section, initially, the importance of limiting the rate and burst of traffic flows in predictable networks (see Sec. 4.1.1) is discussed. Thereafter, the theoretical background behind traffic policing is presented (see Sec. 4.1.2).

4.1.1 The Need for Traffic Policing

Providing predictable latency in programmable networks requires the computation of performance bounds, in particular delay and throughput, in the network. DNC [LT01]; [VK16] is the main framework used by *SotA* solutions for predictable latency [Jan+15]; [Van+19b]; [Van+20] to compute performance bounds.

DNC is a system theory for communication networks based on the min-plus algebra. Based on traffic and node models, DNC allows to derive *(i)* the maximum per-packet delay traffic can experience at a node, *(ii)* the maximum backlog (e.g., amount of data) traffic will generate at a node, and *(iii)* the updated traffic model at the output of the node. Altogether, these bounds enable operators to provide guarantees to their tenants.

Inevitably, a key requirement for the bounds computed by DNC to be valid is for the traffic and node models to be correct worst-case models. In DNC terminology, the traffic model is referred to as the *arrival curve* and the node model is referred to as the *service curve*. Many works have focused on determining the service curve of network nodes, e.g., the recent *Loko* system [Van+19b]. Given a service curve, it is crucial to ensure that the traffic entering the node is not exceeding its arrival curve. Indeed, if the traffic violates the arrival curve used to compute delay and performance bounds, all the guarantees provided to the applications would vanish. For example, slightly exceeding the expected traffic envelope can increase buffer occupancy at some nodes, thereby potentially reaching the buffer capacity of a node, and hence generating packet loss, retransmission, and degraded performance.

Tenants requesting predictable performance from the network are expected to provide the arrival curve of the traffic they wish to send into the network. Because tenants cannot be trusted, operators must ensure that the traffic sent by the different tenants does not exceed the agreed arrival curve. This can be achieved by adding a DNC processing element before the traffic enters the network and that ensures that its output traffic respects the agreed arrival curve (see *(iii)* above). This can be done by either delaying or dropping packets that would lead to the arrival curve being violated. This is called *shaping* and *policing*, respectively. In this chapter, we focus on traffic policing.

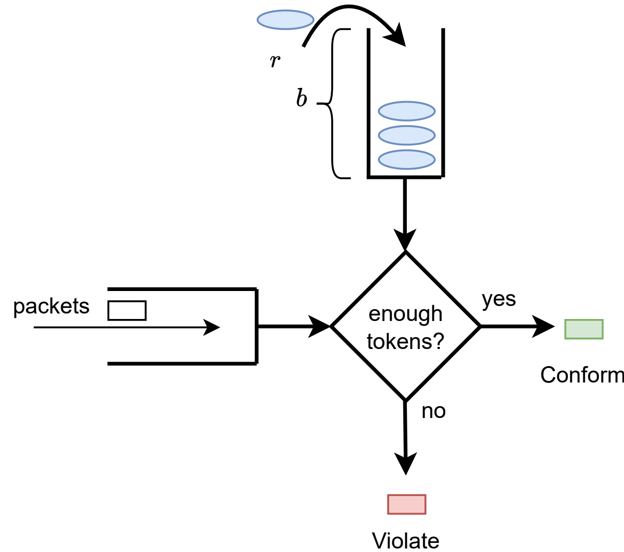


Figure 4.1: Diagram representing a leaky token bucket algorithm. The maximal token bucket size is b, while the rate of token generation rate is r.

4.1.2 Leaky Token Bucket Algorithm

The main type of arrival curve used in the literature is a token bucket arrival curve. In the token bucket arrival curve, traffic is modeled with two parameters: allowed burst and allowed rate. This corresponds to the famous leaky token bucket algorithm (see illustration on Fig. 4.1) [Lee94]. Initially, the token bucket contains b tokens (e.g., bits or packets) which determines the maximum burst size of a flow. The bucket is continuously filled with r tokens per second, which defines the rate of a flow. The token bucket can never be filled with more than b tokens. When an N-bit packet arrives, the number of available tokens in the bucket is checked. If this value is higher than N, the packet is marked as green (conformant) and forwarded, and N tokens are removed from the bucket. If there are not enough tokens available (i.e., the value is lower than N), the packet is marked as red or non-conformant/violated. The procedure of determining if there are enough tokens in a bucket and marking a given packet accordingly is called traffic metering. Packets marked as red are either dropped (this is called traffic policing), sent at a later time (called traffic shaping), or are remarked and forwarded. Packet (priority) remarking usually refers to lowering the drop precedence of the Differentiated Services Code Point (DSCP) field in the Internet Protocol (IP) header. In case when the flow should be metered according to the packet rate, the procedure remains the same, while the number of tokens needed for sending one packet is 1.

In DNC, the token bucket algorithm constrains the burst b and rate r of a flow with the token bucket arrival curve [LT01]; [VK16], defined as $\gamma_{r,b}$:

$$\gamma_{r,b} = \begin{cases} b + r \times t & , \forall t > 0 \\ 0 & , otherwise \end{cases}, \tag{4.1}$$

where t is time. Graphical representation of how one token bucket algorithm constrains the data or traffic is represented in Fig. 4.2.

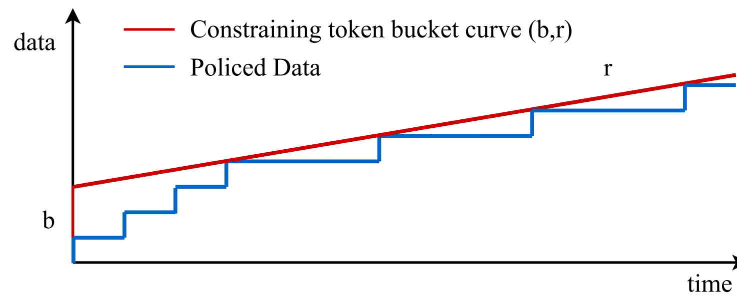


Figure 4.2: Example of discrete data being policed (or constrained) by a policer based on the token bucket algorithm. The red line represents a token bucket arrival curve.

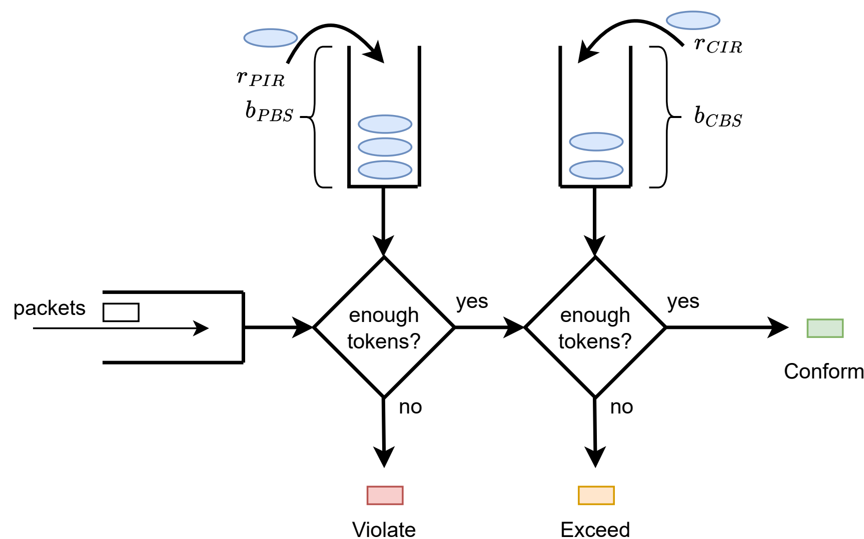


Figure 4.3: Diagram representing two-rate-two-bucket token bucket algorithm (as defined in RFC 2698). r_{PIR} is peak information rate, b_{PBS} is peak burst size, r_{CIR} committed information rate, b_{CBS} commited burst size.

One of the main limitations of a standard leaky token bucket algorithm is that it only meters the packets into two colours. This means that a network operator can only choose to either drop all of the excess traffic (non-conformant) or to remark it and forward it into the network. Naturally, dropping all of the excess traffic is bad for the user (flow owner). While remarking and forwarding all of the packets can be very risky for a network operator as it introduces excess traffic. Thus, there are a few extensions of a standard leaky token bucket algorithm that aim to resolve this issue. For instance, leaky *two-rate-two-bucket* token algorithm as defined in RFC 2698 [HG99] can color the packets into three colors. To achieve this, the algorithm uses two sequential stages where each stage contains one leaky token bucket (see Fig. 4.3). Each token bucket is independent and it utilizes the standard leaky token bucket algorithm. The parameters of the first bucket are peak information rate (i.e., r_{PIR}) and peak burst size (i.e., b_{PBS}). While the parameters of the second bucket are committed information rate (r_{CIR}) and committed burst size (i.e., b_{CBS}). Typically, peak values (i.e, rate and burst) are higher compared to the committed ones. One common setting is to: *(1)* drop all the packets (extreme violations) exceeding the parameters of the first bucket, *(2)* forward the packets without remarking if they are conformant according to the both buckets, and *(3)* forward the packet with

remarking if they are conformant according to the first bucket and non-conformant according to the second one (i.e., exceeded).

Available *carrier-grade* hardware devices (switches) can offer vastly different traffic metering support. For instance, some hardware switches may support only configuring the rate of a single leaky token bucket (see Sec. 4.4.3). However, most of the newer devices support configurable *two-rate-two-bucket* token algorithm or similar variation of it (see Chapter 5).

Since the main type of arrival curve used in the literature and mathematical frameworks providing deterministic guarantees (e.g., DNC) is a standard token bucket arrival curve, this chapter focuses only on it. That is, *two-rate-two-bucket* token algorithm and similar variations are excluded from further analysis.

The main challenge here is to investigate if the traffic metering and policing functionality of modern *carrier-grade* switches and end-hosts (general purpose servers) matches the theoretical properties of the leaky token bucket algorithm. In particular, this chapter focuses on the following research questions:

1. How can the performance of a traffic policer be properly measured?
2. How accurate is the traffic policing feature of *carrier-grade* switches and end-hosts?
3. What is the impact of potential traffic policing inaccuracies on the performance of systems providing predictable latency?

4.2 Related Work

The responsible unit to perform the in-network policing is the network switches. Generally, the measurement and investigation of different performance metrics of the OF switches have been receiving a lot of attention in the literature [Van+19a]; [KPK14a]; [KPK15]; [Kuź+18]; [He+15a]; [BR13]; [DBK15]; [Laz+14]; [HYS13]; [Rot+12]; [He+15b]; [Bau+18]; [PMK13]; [Bia+10]; [Emm+14]; [Van+19b]; [HYS13]; [Jar+11]; [Nao+08]; [GYG13]; [Lin+17].

On the one hand, some works have studied the mismatch between the Control Plane (CP) and DP states of a device [Van+19a]; [KPK14a]; [KPK15]; [Kuź+18]; [He+15a]; [BR13]; [Laz+14]; [HYS13]; [Rot+12]; [He+15b]. As an interesting finding, authors in [Van+19a] have shown that inserting a new OF rule into the flow table of a *carrier-grade* OF hardware switch can take a significant amount of time (can be over a second). Worse than that, sometimes the CP state claims that a forwarding rule is inserted in the switch, while it has never been inserted [Van+19a]; [KPK15]; [Kuź+18]. As we know, to police a traffic flow, the switch has to use a forwarding rule to match the traffic and forwards it to the corresponding meter. Therefore, these mismatching issues can affect the policing function as well and need to be accounted.

On the other hand, some works have focused on measuring the DP performance of *carrier-grade* OF switches [Bau+18]; [PMK13]; [Bia+10]; [Emm+14]; [Van+19b]; [HYS13]; [Jar+11]; [Nao+08]; [GYG13]; [Lin+17]. For instance, [Van+19a]; [Van+19b]; [Bau+18] have measured the available hardware flow table size of various Software-Defined Networking (SDN)-enabled switches and compared their performance. Durner. et al. [DBK15] have measured and evaluated different Quality of Service

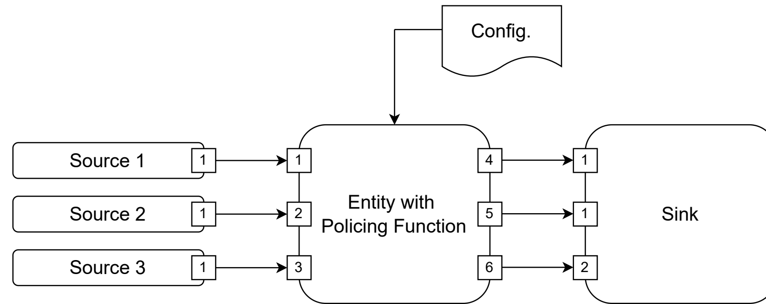


Figure 4.4: Generic testbed.

(QoS) metrics in OF switches such as the priority queuing. However, to the best of our knowledge, we are the first work that investigates the performance of policing feature of the OF switches. Although the packet processing time of the switches has been studied in other works [Van+19a]; [Bau+18]; [HYS13]; [Lin+17], none of them have considered the impact of traffic policing. In addition to the processing time, we study the accuracy of the rate and burst policing on a set of *carrier-grade* OF switches.

Recent works in the literature have focused on improving the host-based traffic policing approaches [Sae+17]; [Jan+15]. Although these solutions can achieve high accuracy [Sae+17], they often cannot support high rates, especially if the packet size is small.

4.3 Procedure for Measuring Traffic Policing Accuracy

This section discusses how the accuracy of rate and burst size policing can be obtained from the device measurements. Two approaches are presented. The first one (presented in Sec. 4.3.2) is a standard approach based on the deriving *average* rate and burst, and this approach is the mostly used one in the litereature which focuses on evaluating the accuracy of policing or shaping [Sae+17]; [Jan+15]. The second approach (presented in Sec. 4.3.3) is novel and is one of the contributions presented in this chapter. It is based on the concepts developed in the DNC framework (see Sec. 4.3.4) and it aims to resolve some issues of the previously mentioned standard approach. Since both approaches aim to explain how policing accuracy can be derived from device measurements, we firstly present (in Sec. 4.3.1) one generic measurement setup which can be used to obtain suitable traces. Afterwards, the two aforementioned approaches are presented along with their advantages and disadvantages.

4.3.1 Generic Measurement Setup

Most of the works in the literature [DBK15]; [Đer+21]; [Van+20] which evaluate the traffic policing accuracy of a hardware device or a software program use the logical measurement setup, or a slight variation of it, illustrated in Fig. 4.4. Additionally, the measurement results presented later in this chapter are also based on the illustrated setup.

To begin with, there is usually one or more traffic sources which are represented in the left part of the figure. The traffic sources are either dedicated physical machines/servers (with traffic generation software) or Virtual Machine (VM)s running locally on the same location as the entity with policing function. Each traffic source is connected over an interface (either physical or virtual) to the entity

which contains traffic policing functionality. The traffic sources generate traffic, usually with a high rate and burst size, and send it to the policing entity. In this thesis, we often refer to the traffic being sent to the policing entity as pre-policed traffic. The entity with policing function receives the generated traffic (i.e., pre-policed packets), and polices them according to the external configuration parameters. The traffic forwarded from the entity to the sink is often called policed traffic. The policing entity can be either a hardware switch fitted with traffic meters, or a software application running on a server. The policed traffic is then forwarded to the sink, which time-stamps all the packets and saves them as traffic traces for further processing. The sink is most commonly realized as a dedicated hardware component, usually fitted with a high precision measurement card. Based on the timestamped packet and saved traffic traces, it is possible to derive some crucial properties (e.g., average rate) of post-policed traffic. These properties can then be later compared with the configuration parameters to derive what was the achieved policing accuracy.

4.3.2 Standard Measurement Procedure

Many works [DBK15] in the literature simply focus on measuring and evaluating the average policing rate and the initial burst size. To do so, they simply configure the traffic sources to generate traffic at a line rate, or a rate much higher compared to the configured policing rate. Therefore, the average rate of the post-policed traffic arriving (after the initial burst) at the sink is expected to be similar to the configured rate of the policer. For instance, if one source generates the traffic with the rate of $1Gbps$, and the policer is configured to police the traffic with the rate of $500Mbps$, the average traffic rate at the sink (after the initial burst) is expected to be close to the configured policing rate, i.e., $500Mbps$. The accuracy of rate policing can then be expressed by comparing the configured value and the measured ones. For instance, in this case, the relative rate policing error e_r in a time interval (t_{i-1}, t_i) can be derived with the following equation:

$$e_r^i = \frac{r_m^i - r_c}{r_c} \tag{4.2}$$

where r_m^i is the measured traffic rate in the time interval and r_c is the configured policing rate. Note: selecting a short time interval may lead to a high error which does not depict the real situation. The measured traffic rate in time interval (t_{i-1}, t_i) can be calculated with the following equation:

$$r_m^i = \frac{\sum_{\forall p \in (t_{i-1}, t_i)} s(p)}{t_i - t_{i-1}} \tag{4.3}$$

where p represents the time when a packet was observed on the sink, and $s(p)$ is the size of a packet.

Regarding the burst size, the most simplistic way to derive it is to simply observe when the first packet was dropped at the start of a measurement scenario. This time represents the time when the token bucket of a meter was empty or it didn't have a sufficient amount of tokens for forwarding a packet. If the pre-policed packets were generated at a much higher rate compared to the policing rate, the burst size or token bucket size is approximately equal to the sum of the packet sizes of all the forwarded packets before the first drop.

However, this procedure has a couple of drawbacks. Firstly, since the traffic is generated at a constant rate, different states of a meter (or a token bucket) are not extensively tested. That is, if the

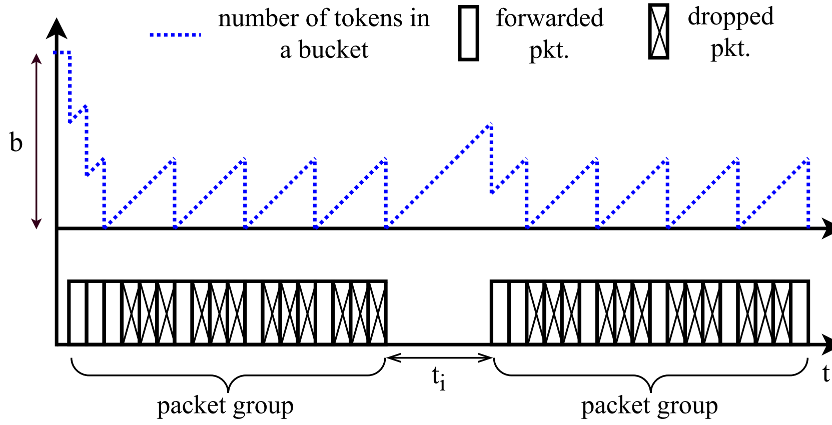


Figure 4.5: An illustration of one time-wise sequence of pre-policed and policed packets with the corresponding token bucket state. The scenario depicts two packet groups, where each one contains at least 3 packets that are not part of the initial burst.

generation rate is higher compared to the configured policing rate, the token bucket will be quickly almost fully emptied. Therefore, the state of a token bucket (i.e., the range of the number of tokens in a bucket) will be the same during the whole measurement procedure. Secondly, the DNC based systems [Van+20] which provide deterministic guarantees require an upper bound of the rate, and not an average rate. To be more precise, according to Eq. 4.3, the average rate is calculated over a certain time period. Therefore, the packet distribution in the corresponding time period cannot be derived from the average rate (e.g., are the packets uniformly spaced or grouped in bursts) and this is insufficient for the aforementioned systems. Therefore, the average rate is not sufficient. These observations motivated us to create a novel and more suitable measurement procedure which utilizes certain properties of DNC mathematical framework. This procedure is presented in the following section.

4.3.3 DNC-Based Measurement Procedure

In order to test all the different states of a meter, we proposed to use the following traffic generation procedure. Fig. 4.5 illustrates an exemplary time series of the generated pre-policed traffic during one measurement run. *Source* generates multiple randomly-spaced groups of packets at line rate. The idea behind sending a group of packets at a line rate is two-fold. Firstly, we aim to empty the token bucket of the corresponding meter to observe the accuracy of burst policing. Secondly, we strive to observe the accuracy of the token generation rate after the bucket is emptied. To achieve these goals, considering the token bucket rate and burst, the length of the generated packet group has to be big enough to observe a burst and a few packets (10 packets in our case) policed at line rate.

Further, to explore how the policing behaves with different initial states of the token bucket (i.e., number of tokens), we vary the time between packet groups t_i with the following uniform distribution:

$$t_i(r, b) = U(0, 1.5 \times \frac{b}{r}), \tag{4.4}$$

where $\frac{b}{r}$ is the time needed to completely fill the token bucket of a meter. We multiply it by 1.5 to make sure that the token bucket can be filled fully again during one measurement run.[1]

4.3.4 Deriving Rate and Burst

In this subsection, we explain how the rate and burst size of a policed flow are derived from a measurement trace. Deriving a valid (constraining) and accurate token bucket curves (defined with rate r and burst b) is crucial for ensuring the correct operation of DNC-based systems (as explained in Sec. 4.1.1). To do so, we rely on token bucket properties and the methodologies provided in the DNC framework.

Initially, we start with applying min-plus self-deconvolution [LT01]; [VK16] on the measured policed traffic flow to produce its minimum arrival curve [LT01]; [VK16]. In particular, the minimum arrival curve represents a valid flow model that can be used as an input parameter (flow description) for providing guarantees with DNC. For a detailed explanation of the DNC framework, we refer the readers to [LT01].

There are many ways of deriving a valid (constraining) token bucket arrival curve from the minimum arrival curve. Any curve which is above the minimum one represents a valid solution. In this chapter, firstly the rate of a flow is derived from a minimal arrival curve. Afterward, the burst size is calculated based on the determined rate.

Rate. To derive the rate, the timestamp of the first (t_a) packet which is not part of the initial part ($t < t_a$) of the minimum arrival curve (see Fig. 4.6)[2] is taken. In our scenario, we can find the first packet based on the packet inter-arrival times. We define rate r as the maximal slope of the minimum arrival curve y between t_a and any other time instance $t \geq t_a$ (e.g., t_b in Fig. 4.6):

$$r = \max_{\forall t, t > t_a} \left(\frac{y(t) - y(t_a)}{t - t_a} \right). \tag{4.5}$$

Burst. To calculate the burst size, we simply find the minimal b which satisfies the following equation:

$$b + r \times t \geq y(t). \tag{4.6}$$

Using the previously explained procedure, it is ensured that the derived token bucket arrival curve $\gamma_{r,b}$ is always above the self-deconvoluted minimum arrival curve. Hence, it is constraining the flow with rate r and burst b.

4.4 Switch Measurements

The main goal of this section is to understand the traffic policing capabilities and accuracy of *carrier-grade* OF switches. Therefore, in the beginning, it is discussed how traffic policing is realized in OF (see Sec. 4.4.1). Afterwards, the measurement setup is presented (see Sec. 4.4.2). While the measurement results are presented in Sec. 4.4.3.

[1] If multiple meters are used during the same run, the corresponding generated packet groups (one group belongs to one meter) are interleaved. In these cases, t_i (Eq. 4.4) separates interleaved packet groups.

[2] At higher policing rates, switches forward the packets in microbursts (at a line rate). In such cases, we take the timestamp of the last packet of the first microburst. This effect is presented in Sec. 4.4.3.6.

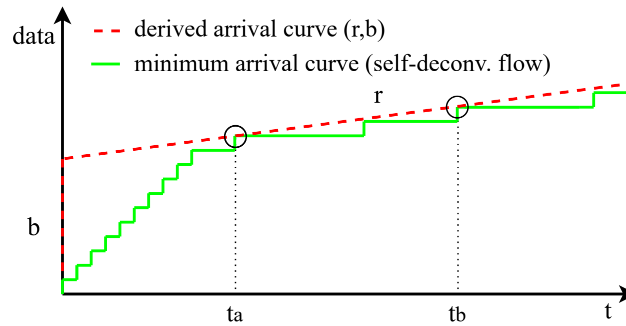


Figure 4.6: An illustration of one time-wise sequence of policed packets (measured traffic trace) with the corresponding self-deconvoluted function and the constraining token bucket arrival curve.

4.4.1 OpenFlow Meters

In OF, traffic policing can be realized with the *metering* feature [McK+08]. Metering is introduced in OF 1.3 [McK+08] and it is supported by most of the *carrier-grade* switches. This feature is typically realized as a hardware meter. In OF, meters and flow rules are disjoint. Hence, to police a traffic flow, a meter has to be configured and assigned to the corresponding flow rule as part of the instructions of the flow rule.

Various metering configurations are supported [McK+08], such as: *i)* rate policing (either kbps or packets per second (pps)), *ii)* burst size policing (either kbps or pps), *iii)* different band types (excess traffic is either dropped or remarked and forwarded), *iv)* collecting metering statistics.

4.4.2 Setup

In this part, the measurement setup, parameters, and procedure are introduced. The setup is based on the generic measurement setup presented in Sec. 4.3.1. Table 4.1 shows the considered OF switches for benchmarking. To measure the performance of the traffic policing feature of these switches, the following measurement setup presented in Fig. 4.7 is constructed. It consists of two servers, one networking tap, and a Device Under Test (DUT), i.e., one of the switches from Table 4.1. On the first server (i.e., Host 1 in Fig. 4.7), a Ryu SDN controller [Ryu15] is deployed, running a custom application that configures the DUT with a certain number of flow rules, and police the traffic based on the parameters listed in Table 4.2. We note that the parameter values are chosen based on the related *SotA* approaches [Jan+15]; [Van+20]. Configuring a DUT (i.e., modifying flow rules and meters) is done through OF 1.3 (green dashed line in Fig. 4.7). Each inserted flow rule matches the incoming traffic with a certain unique IP address on port 1. The matched traffic then passes through exactly one unique meter, and it is forwarded further on port 2 (packets can be dropped depending on the policing outcome). Additionally, Host 1 runs Moongen traffic generator [Emm+15] to send the DP pre-policed traffic (blue line in Fig. 4.7). This generated traffic is mirrored by the networking tap device, and it is forwarded to both, the DUT and the second server which contains Endace DAG 7.5G4 measurement card [Lim16]. Finally, based on the configured rules, the DUT forwards the traffic to the Host 2 server (equipped with a measurement card). Thus, on the second server, it is possible to obtain traffic traces of both, pre-policed and post-policed traffic. The cases when the total pre-policed rate is lower or equal to the total configured policing rate are not considered.

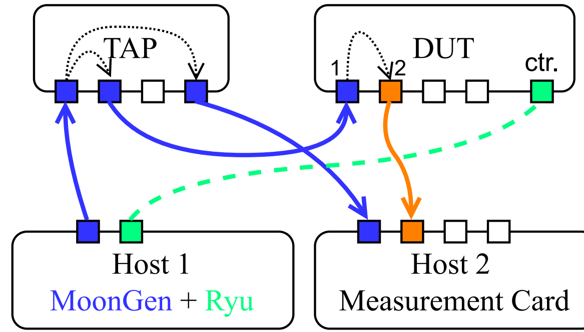


Figure 4.7: The measurement setup.

Table 4.1: Specifications of the evaluated switches: names, ASIC, CPU, and ports.

Switch	ASIC	CPU	Ports
HP E3800	HPE ProVision	Freescale P2020	48×1G-RJ45 + 4×10G-SFP+
DELL S3048-ON	Broadcom StrataXGS	*undisclosed*	48×1G-RJ45 + 4×10G-SFP+
DELL S4048-ON	*undisclosed*	*undisclosed*	48×10G-SFP+ + 6×40G-QSFP+
Pica8 P3290	Broadcom Firebolt 3	Freescale MPC8541CDS	48×1G-RJ45 + 4×10G-SFP+
Pica8 P3297	Broadcom Triumph 2	Freescale P2020	48×1G-RJ45 + 4×10G-SFP+

Table 4.2: Considered Parameters

Parameter	Abbreviation	Configured on	Values
Number of flows	n	Ryu, MoonGen	1, 10
Policing rate [kbps]	r	Ryu, MoonGen	10, 10^2, 10^3, 10^4, 10^5
Burst size [kbits]	b	Ryu, MoonGen	13, 15, 30, 50, 75, 100, 200, 300, 500, 1000
Packet size [byte]	s	MoonGen	100, 500, 1000, 1500

4.4.3 Results

4.4.3.1 Policing Flags Support

Table 4.3 lists the supported policing flags and the total number of available meters within each switch. Furthermore, for easier comparison, the table lists their corresponding flow table size (values are taken from [Van+19a]). To begin with, we observe that all the switches do support a traffic policing feature, except the ones manufactured by Dell (i.e., 1G S3048-ON and 10G S4048-ON). Therefore, we only consider *Pica8* and HP switches for the measurements. Moreover, considered HP device only supports rate policing, while *Pica8* switches support also burst policing in addition to rate. As a result, this limitation significantly constrains the usage of HP switches in predictable networks. This is discussed in more detail in Sec. 4.6.

Table 4.3: Supported Policing Flags, and Number of Meters

Switch-Type	Policing Flags				Band Type		Number of	
	rate	pps	burst	stats	drop	dscp_remark	meters	flows
HP E3800	✓	✓	×	✓	✓	✓	2047	ca. 4085
Pica8 P3290	✓	✓	✓	✓	✓	✓	4096	2046
Pica8 P3297	✓	✓	✓	✓	✓	✓	8192	4094
DELL S3048-ON	×	×	×	×	×	×	-/0	1000
DELL S4048-ON	×	×	×	×	×	×	-/0	1000

As it can be seen in Table 4.3, *Pica8* switches support significantly more meters compared to HP. Since *SotA* approaches rely on a fine-grained flow control [GVK17], the total number of flows on each switch in the network can easily grow up to several thousands [Van+20] (e.g., over 2000 flow rules). Thus, having around 2000 meters may be insufficient for some use-cases. Furthermore, *Pica8* switches are equipped with more meters than the flow table size. This can facilitate the deployment of hierarchical traffic policing approaches [Doc21].

Finally, regarding the band types (see Sec. 4.4.1), it is observed that HP and *Pica8* switches support both drop and packet remarking.

4.4.3.2 Processing Time

To measure the impact of traffic policing on the packet processing time of each switch, two sets of measurements are performed, with and without policing. To disable policing, we do not configure the flow rules to forward the matched traffic to each corresponding meter. The processing time of each packet is the difference of a packet's timestamp before and after (policing and) forwarding it. In order to identify each packet, each packet has a unique Media Access Control (MAC) address.

Outcome. The minimal, maximal, and average packet processing time of switches are presented in Fig. 4.8 (the data is consolidated and based on all the measurement runs). Overall, there is no statistically significant impact of policing on the processing time. For instance, the minimal (and maximal) packet processing time of *HP 3800* is the same in both cases, i.e., $t_{min} \approx 3\ \mu s$ (and $t_{max} \approx 4\ \mu s$). To be more clear, we also illustrate five specific scenarios in an isolated manner in Fig. 4.9. These scenarios have the same packet size of 1000 bytes, but different configured policing rate and burst size (see Fig. 4.9). Accordingly, Fig. 4.9 shows that first, the processing time of the *Pica8* switch is higher than HP. More importantly, it can be seen that the packet processing time with and without policing is almost the same, even for different policing rates. Thus, it can be concluded that the policing is implemented in hardware (processing time usually varies a lot in software implementations [Van+19a]). Furthermore, it can be assumed that all the results presented in comprehensive *SotA* hardware measurement studies [Van+19a]; [Kuź+18]; [Bau+18] can be valid in cases with policing enabled.

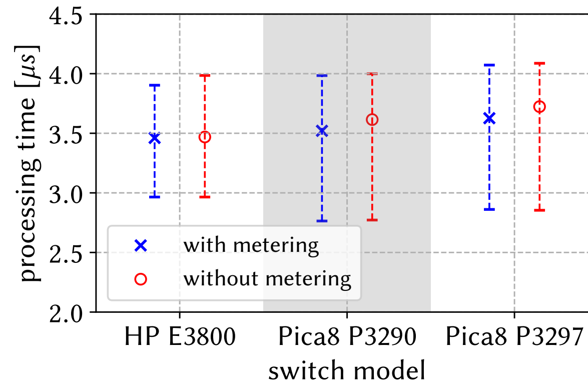


Figure 4.8: Impact of metering on the processing time. All the measurement data is aggregated.

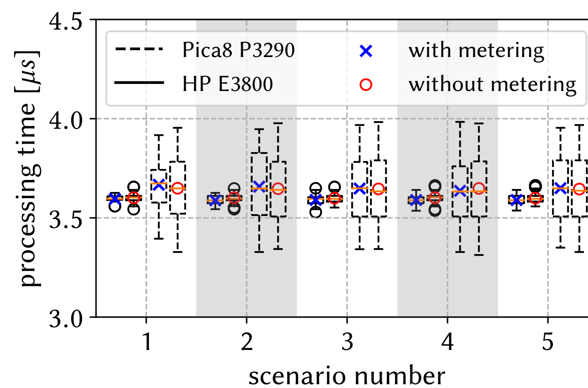


Figure 4.9: Impact of metering on the processing time for five different measurement scenarios.

4.4.3.3 Verifying the Derived Token Bucket Curves

Fig. 4.10 depicts the initial part of the policed traffic trace, its min-plus self-deconvolution (i.e., minimum arrival curve), and the derived token bucket arrival curve for three different measurement runs. Since the switches police the traffic according to the token bucket algorithm, the initial parts of the self-deconvoluted function and policed traffic trace are very similar (see Fig. 4.10). Furthermore, the derived rate and burst of each token bucket arrival curve indeed fit well to the corresponding minimum arrival curve. We note that relying on the configuration values for generating a valid arrival curve has never been sufficient in our experiments. This means that the evaluated switches have always forwarded more traffic than expected.

4.4.3.4 Rate Deviation

Fig. 4.11 illustrates the relative deviation of the derived policing rate from the configured one for *Pica8 P3290*, *Pica8 P3297*, and *HP E3800*. Overall, the derived rate often exceeds the configured one (average deviation is around ∼+1%), hence, the switches forward excess traffic into the network. This can lead to delay violations and even packet loss. Furthermore, in the case of lower rates, both *Pica8* switches significantly deviate from the expected flow rate. For instance, if we configure a meter to police the traffic with the rate of 10 kbps, the forwarded policed rate is around 60% higher than expected, i.e., ∼ 16 kbps. This unexpected inaccuracy can have a significant impact on deploying *SotA* solutions, which is further investigated in the discussion section. Nevertheless, these deviations appear to be

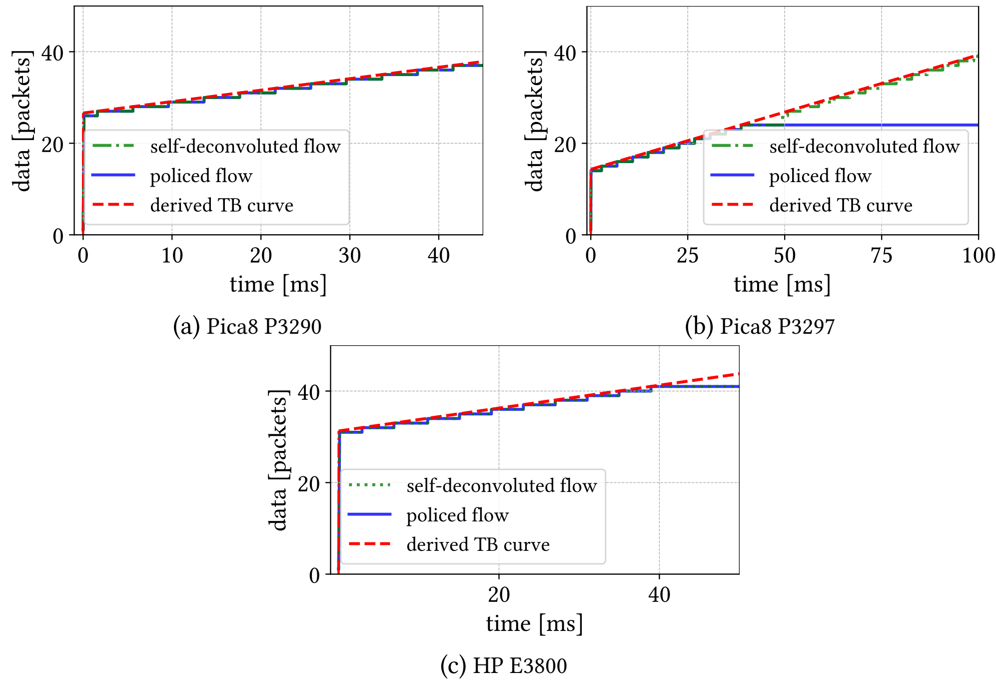


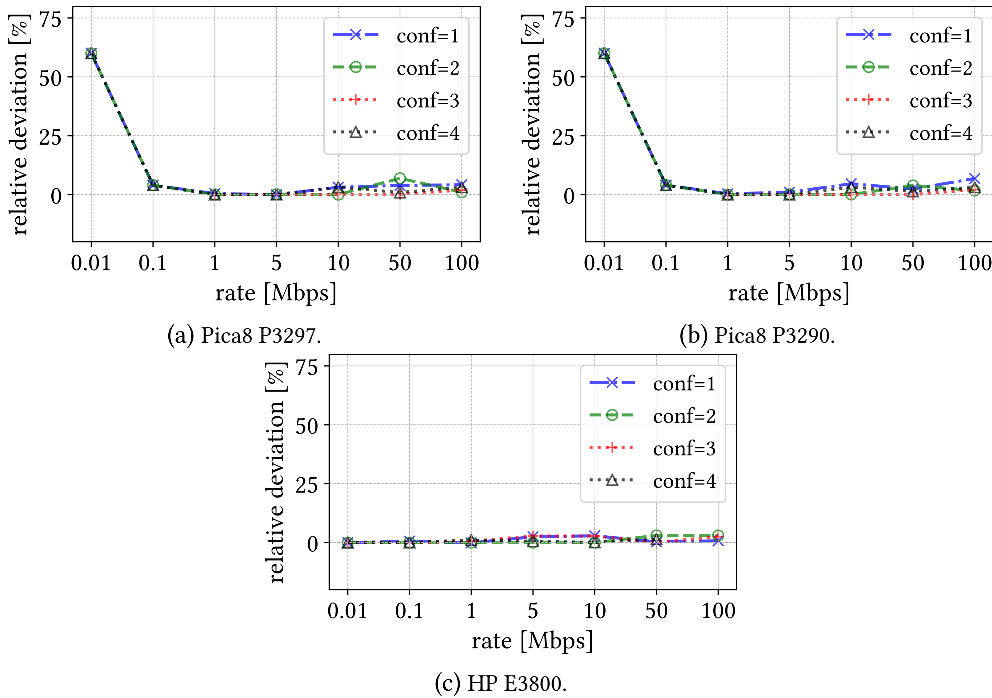
Figure 4.10: An initial part of the policed traffic trace, its min-plus self-deconvolution, and the derived token bucket arrival curve for the considered switches. In all three scenarios, policing rate is 1 Mbps, and the packet size is 500 bytes. The configured burst size for *Pica8 P-3290* is 100 kbits (or 25 500 byte packets) and for *Pica8 P-3297* is 50 kbps (approx 12.5 500 byte packets).

predictable, since, in different scenarios with the same configured/expected rate, the switches police the traffic with similar accuracy. It indicates that these switches can be used in predictable networks (if the error is accounted for).

4.4.3.5 Burst Deviation

To study the burst deviation, we start with Fig. 4.12a which presents the relative deviation of the derived burst size from the configured one for *Pica8 P3297* switch. In this case, we consider four different scenarios with the same parameters except for the packet size (i.e., the number of flows is 1 and the configured rate is 1 Mbit). Firstly, it can be observed that the derived burst size is always higher than the configured/expected one, which can be detrimental in predictable networks. Also, the relative burst deviation depends on the configured one. For the higher values, the inaccuracy is usually below 5%. For example, the relative error is always lower than 3% for the configured burst size of 500 kbits. Surprisingly, for configurations with smaller burst sizes, the relative error can even exceed 100%. Moreover, even for the same configured burst sizes, the relative burst deviation varies with the packet size. This indicates that it is not possible to fully compensate for these inaccuracies with precise modeling. Even if we perfectly model the error of a device, users might generate flows with dynamically varying packet sizes. This is very common behaviour when transport protocol such as Transmission Control Protocol (TCP) are used.

Further, we note that the absolute burst deviation from the configured value of *Pica8 P3297* does not depend significantly on the configured one (see Fig. 4.12b). The absolute burst deviation is always between 0 and 16 kbits. This suggests that each meter introduces a similar amount of excess traffic



(a) Pica8 P3297.

(b) Pica8 P3290.

(c) HP E3800.

Figure 4.11: Achieved accuracy (in terms of relative deviation) of rate policing depending on the configured policing rate. We consider four different measurement configurations with the following parameters, $c1 \longrightarrow (b = 30$ kbps, $s = 100$ bytes, $n = 1)$; $c2 \longrightarrow (b = 100$ kbps, $s = 500$ bytes, $n = 1)$; $c3 \longrightarrow (b = 200$ kbps, $s = 1500$ bytes, $n = 1)$; $c4 \longrightarrow (b = 75$ kbps, $s = 1000$ bytes, $n = 1)$. Since it is not possible to configure the burst size on *HP E3800*, the corresponding parameters is ignored.

in the network, regardless of the configuration. Additionally, it is not possible to configure a meter of *Pica8 P3297* with a burst value lower than 13 kbits.

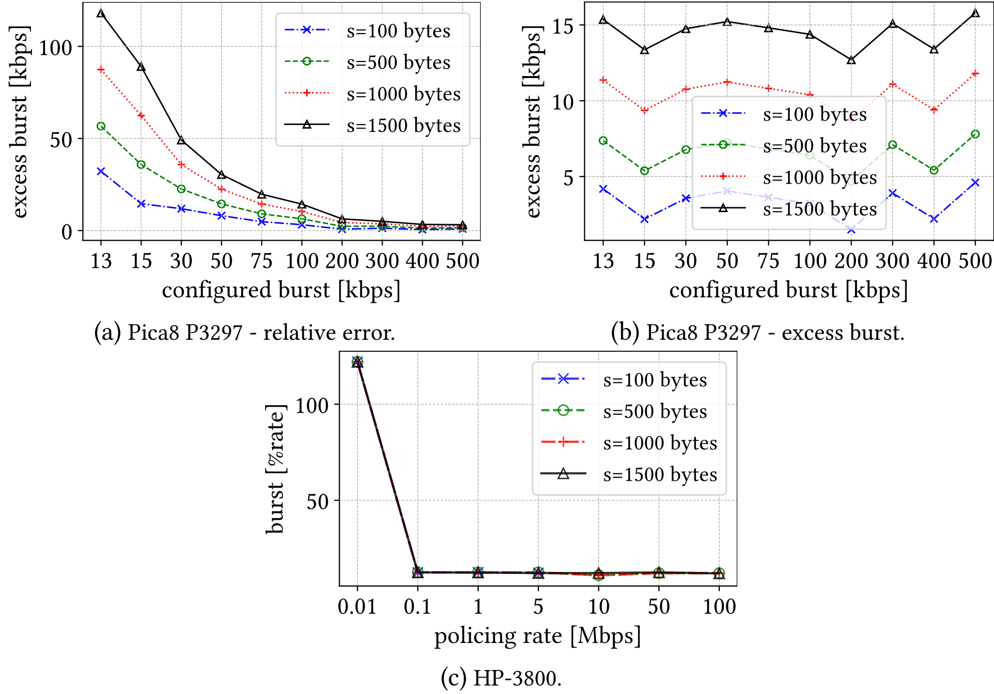
The results for *Pica8 P3290* follow the same trend, thus we omit to show them.

Regarding the *HP E3800* switch, since it does not support the configuration of the burst size, we present the results without varying this parameter. Fig. 4.12c presents the derived burst size for four different scenarios with varying policing rate. Even though we cannot configure the burst size of HP, it can be observed that the burst size is predictable (see Fig. 4.12c). That is, runs with different measurement parameters produce almost identical results. Furthermore, the results show that the burst size is correlated with the configured policing rate (see Fig. 4.12c). For instance, if the configured/expected rate is lower than 100 kbps, the measured burst size is slightly bigger (i.e., $\sim$ 12.5 kbits) than the maximal size of an ethernet packet (i.e., 1500×8 bits $=$ 12 kbits)[3]. For the higher rates, the burst size corresponds to $\sim$ 12.5% of the configured rate.

4.4.3.6 Microbursts at High Policing Rate

Fig. 4.13 illustrates three very short time snippets taken from different (high-rate) measurement runs for *Pica8 P3297*, *Pica8 P3290*, and *HP E3800*. The time snippets show the relative timestamps of pre-policed and policed traffic (i.e., packets) shortly after the token bucket is emptied (with an initial

[3]This is probably done in order to accommodate adding multiple Virtual Local Area Network (VLAN) tags (one tag is 4 bytes).



(a) Pica8 P3297 - relative error.

(b) Pica8 P3297 - excess burst.

(c) HP-3800.

Figure 4.12: Derived burst size from the measurements (with different packet sizes s) for *Pica8 P3297* and *HP E3800*. The derived rate for *Pica8 P3297* in all presented scenarios deviates at most 0.5% from the configured value.

burst). In theory, the token bucket algorithm generates tokens continuously. Therefore, since the pre-policed traffic is sent at a line rate, the outgoing policed traffic in the depicted scenarios should be uniformly spaced. For example, in the case of *Pica8 P3297* (see Fig. 4.13a), the time to generate enough tokens for one packet is 8 μs (policing rate is 100 Mbps and packet size is 100 bytes). Hence, an ideal token bucket algorithm should forward six uniformly separated packets every 8 μs. However, in practice, we observe that this is not the case. The packets are actually forwarded in small (micro) bursts (e.g., two packets for *Pica8 P3297*, see Fig. 4.13a). In fact, all the considered switches exhibit the same behavior (see Fig. 4.13). However, this effect is only observable at high policing rates (e.g., $r \geq 50$ Mbps) with small packet sizes (e.g., $s \leq 500$ bytes). Therefore, we can suppose that the token bucket is discretely filled with a time interval that can be greater than the time needed to generate enough tokens for a packet. This effect can be accounted for by increasing the derived burst size accordingly.

4.5 End-Host Policing

In order to evaluate the possibility and accuracy of traffic policing on end-hosts, in this section, we initially present how traffic policing can be realized on end-hosts. To do so, we present the implementation details (e.g., algorithm) of one traffic policing function integrated into one DPDK-based application. This application was used to deploy *Chameleon* system, which provides deterministic DP guarantees in data center networks. Additionally, we also discuss the potential sources which can cause inaccuracies in a such realization. The implementation details and the inaccuracy discussion

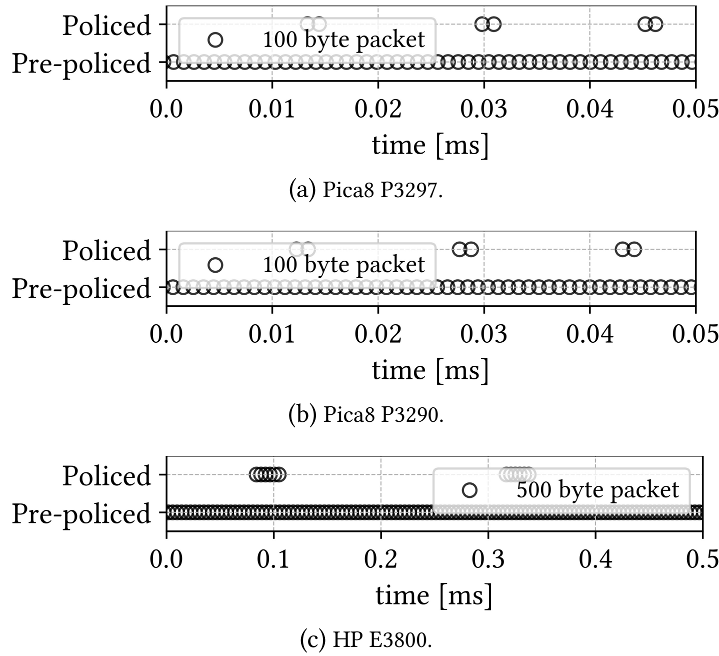


Figure 4.13: Microbursts generated by *Pica8* and HP devices at higher policing rates (i.e., r = 100 Mbps) with smaller packet sizes (e.g., $s \leq 500$ bytes).

represent first contribution of Sec. 4.5. Afterward, the accuracy of this end-host based approach is compared with the traffic policing results of one *carrier-grade* switch (second contribution of Sec. 4.5).

Additionally, *Chameleon* system is already part of one dissertation [Bem+20], however, the comprehensive implementation details (including the discussion) of traffic policing function (in Sec. 4.5.1.1) and the performance comparison with one *carrier-grade* switch (see Sec. 4.5.3) were not discussed. These two parts represent the main contributions of Sec. 4.5.

4.5.1 DPDK Application

In this part, initially, an overview of the DPDK application which was developed to realize *Chameleon* [Van+20] system is provided. More details about *Chameleon* system can be found in the background chapter (i.e., Chapter 2). Afterward, the details regarding traffic policing are presented in detail.

The overall architecture of the DPDK application is presented in Fig. 4.14. The main objective of the developed application is to connect VMs of users with the physical network in a deterministic manner, i.e., the application acts as an enhanced virtual switch. The VMs belonging to users run in QEMU 2.11.1 with KVM and they are connected to the developed DPDK application via *virtio* using a *vhost-net/virtio-net* para-virtualization [Ada+15]. Additionally, each VM is assigned one Rx/Tx queue pair. The DPDK application (based on 19.08 version) runs in a container with privileged access rights, hence, it has full access to the Network Card Interface (NIC). The DPDK application utilizes three separate cores: one for receiving packets (i.e., Rx part), one for sending packets (i.e., Tx part), and one for the control of the DPDK application.

Rx Part. In the receiving part, the DPDK application uses Virtual Machine Device Queues (VMDQ) technology available on Intel's NIC to sort the incoming packets into the physical Rx queue

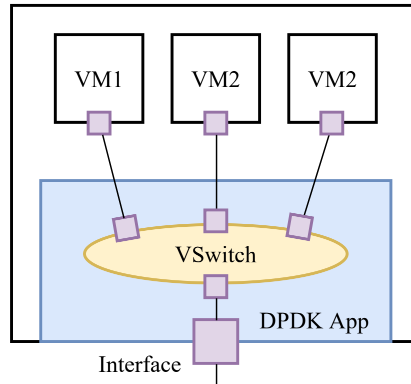


Figure 4.14: Architecture of DPDK application running on end-host.

dedicated to the corresponding VMs. The (software) Rx part of the DPDK application (running in an infinite loop) then pulls a batch of packets and forwards them to the corresponding VMs based on the MAC destination address and VLAN tag. Additionally, in the receiving part, there is no need for policing or any other additional functionality.

Tx Part. In the sending part, the DPDK application runs an infinite loop. In it, it pulls packets (in a batch) from queues connecting the VMs in a round-robin manner. After pulling the packets, the DPDK application adds the corresponding VLAN tags and it polices the traffic with traffic policing functions based on the configured parameters. More information regarding tagging can be found in [Van+20], while the implementation details of traffic policing will be presented in the next subsection.

To ensure that the DPDK application processes packets in a predictable manner, the following configurations were applied on each server before running the application.

- Hyperthreading and Turbo-Boost were disabled. Additionally, all power-saving features were disabled, and the running frequency of each Central Processing Unit (CPU) was set to the (maximal) base value. For example, on Intel Xeon E5-2650 v4 CPU, the running frequency was set to 2.2 GHz for each of the 24 cores.
- Three parts of the DPDK application (e.g., Rx, Tx, and control part) were running on isolated cores. The isolation was achieved by setting kernel *isolcpus* parameter for certain 3 cores and by pinning the DPDK application to those three cores.
- To isolate the available cache to the DPDK application, Intel's Cache Allocation Technology (CAT) was leveraged on each server.

4.5.1.1 Traffic Policing Function

The developed traffic policing functionality presented in Alg. 1 is integrated as a functional module in the previously introduced DPDK application. The traffic policing function processes the packets sequentially and it follows the leaky token bucket algorithm presented in Sec. 4.1.

In general, upon packet reception, the traffic policing function firstly updates the number of tokens available in a bucket. Afterwards, it checks if the updated number of tokens is sufficient for

Algorithm 1 Traffic Policing Pseudocode

```
    Input: Packet Descriptor p.
    Output: Boolean decision if a packet should be dropped or forwarded.
 1: fc = get_flow_config(p)                                   ▷ Get flow configuration parameters.
 2: curr_tsc = rte_rdtsc()                                    ▷ Get current timer.
 3: cycles = (curr_tsc - fc.last_tsc)
 4: gen_tokens = cycles * fc.rate_bps                         ▷ Calculate number of generated tokens.
 5: if (cycles != 0 && gen_tokens/cycles != fc.rate_bps) then ▷ Check for overflow.
 6:     gen_tokens = cpu_freq * fc.burst_bits
 7: fc.last_tsc = curr_tsc                                    ▷ Update timer.
 8: if ((fc.n_tokens + gen_tokens) > cpu_freq * fc.burst_bits) then ▷ Add generated tokens.
 9:     fc.n_tokens = cpu_freq * fc.burst_bits;
10: else
11:     fc.n_tokens = fc.n_tokens + gen_tokens;
12: packet_size = get_packet_size(p)                          ▷ Get packet size in bits.
13: if (fc.n_tokens > packet_size * cpu_freq) then            ▷ Check if there are enough tokens.
14:     fc.n_tokens -= packet_size * cpu_freq
15:     return 1
16: else
17:     return 0
```

sending the corresponding packet. To achieve that, the following procedure presented in Alg. 1 is used.

After a packet is received, the initial step is to determine to which flow the packet belongs (determined by destination MAC and VLAN tag) and what are the configuration parameters (i.e., fc) of that flow (see Line 1). The flow configuration parameters include both static and dynamic variables. Static variables are $flow_rate$ (in bits per second) and $burst_size$ (in bits), while dynamic variables are the current number of available tokens (initialized as $n_tokens = cpu_freq * burst_size$) and the Time Stamp Counter (TSC) of the last processed packet (i.e., $last_tsc$). The unit of the total number of available tokens is $bit * cpu_freq$. We use the previously mentioned unit to avoid the potential inaccuracy caused by rounding up from division (later in the algorithm). TSC is a 64-bit register which counts the number of CPU cycle since reset. Afterwards, the flow configuration parameters are loaded, and the current TSC is read and saved into a variable (see Line 2). Since hyperthreading, turbo-boost, and all power-saving features were disabled on each server, the CPU is running at a constant frequency. This means that the TSC counter increases in a predictable manner (e.g, the value of TSC for Intel Xeon E5-2650 v4 CPU, increases for $2.2 * 10^9$ cycles in one second), and it is not influenced by the total number of performed instructions.

The relative time (expressed as the number of CPU cycles) elapsed between two consecutive received packets can be calculated from the values of TSC counter of the current and previously processed packet (see Line 3). Hence, the total number of generated tokens during this time can be calculated by multiplying the number of elapsed cycles with the configured flow rate (see Line 4).

If the time elapsed between two consecutive packets is high, variable gen_tokens could overflow. This situation is handled by setting the value of gen_token to $cpu_freq * fc.burst_size$, if an overflow

indeed happens (see Lines 5-6). Afterwards, the current TSC time is saved (see Line 7) and the total number of generated tokens is added to the total number of available tokens (see Lines 8-11).

Finally, the total packet size including all headers is determined (see Line 12) and if the number of available tokens is sufficient, the packet is marked for forwarding (see Line 15). If there are an insufficient amount of tokens, the packet is marked for dropping (see Line 17).

Potential Sources of Error. Since this solution is software-based, there are multiple potential sources of error and they are discussed in the following.

- **Sending time of a NIC is not controlled**. After the traffic policing function determines if a certain packet should be dropped or forwarded, the corresponding packet is simply forwarded to the NIC which sends it into the network. However, the developed application does not control the sending time, thus, the corresponding packet can be delayed for an arbitrary amount of time (based on NIC), which could lead to decreased accuracy of rate and burst policing. This problem can potentially be solved by utilizing the planned sending time feature (e.g., Earliest TxTime First (ETF)) which is available on many newly developed NICs.
- **Variable code running time.** Firstly, obtaining the timestamp of a packet in the DPDK application occurs at the start of the traffic policing code (i.e., see Line 2). Secondly, the time to run the presented traffic policing code and the other code in the DPDK application can vary from packet to packet. Therefore, if two consecutive packets are processed at a significantly different rate, this could introduce inaccuracies.
- **Overloading the app.** In an extreme scenario, there could be 20 VMs connected to the developed traffic policing Tx part of the DPDK application running on one core. If all VMs generate a huge amount of traffic, the memory and the DPDK application could be overloaded, thus, introducing inaccuracies.

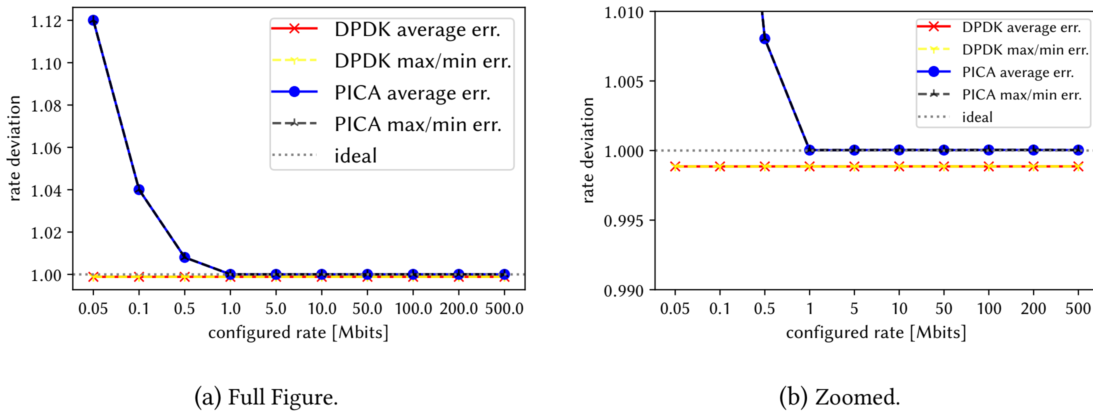
4.5.2 Measurement Setup

To measure the accuracy of the previously introduced software realization of traffic policing function, we simply run the DPDK application on one Dell R530 server and connect it via a 1 Gbps physical link to an Endace DAG 7.5G4 measurement card. The standard measurement procedure is used, as introduced in Sec. 4.3.2. This means that one VM generates the traffic at a maximal rate (e.g., equal to the virtual link rate of 1Gbps), and the accuracy of traffic policing is determined from the observed average values on the measurement card. The server runs Ubuntu 18.04 (4.15.0-66-generic kernel) and it has the following hardware specifications: *1)* Intel Xeon E5-2650 v4 @ 2.2 GHz CPU with 24 cores, *2)* 128 GB of Random Access Memory (RAM), *3)* X540 NIC. The DPDK application is configured to pull packets one by one (i.e., there is no batching) and to add 6 VLAN tags to each packet belonging to any of the configured flows.

Tab. 4.4 shows the considered parameters and their values used for collecting the measurement data.

Paremeter	Values
Num. VMs	1
Num. Flows per VM	1
Packet Size [byte]	100, 700, 1300
Rate size [kbps]	$5 \times 10^4, 10^5, 5 \times 10^5, 10^6, 5 \times 10^6, 10^7, 5 \times 10^7, 10^8, 2 \times 10^8, 5 \times 10^9$
Burst size [bits]	$3 \times 10^4, 10^5$

Table 4.4: Considered parameter and their values in the second measurement campaign.



(a) Full Figure.

(b) Zoomed.

Figure 4.15: Comparison of DPDK-based traffic policing and Pica8 P3297 for different configured policing rates. (a) Figure with full *y-axis* range, and (b) zoomed figure with shorter *y-axis* range.

4.5.3 Results

Fig. 4.15 illustrates the achieved rate policing accuracy of both, the presented DPDK application which includes traffic policing function and Pica8 P3297 switch. Initially, it can be observed that for low policing rates, Pica8 switch exhibits much higher inaccuracy (e.g., max. relative error is around 12%) compared to the DPDK application (e.g., the relative error is below 1%). This excessive deviation of rate (and burst) policing accuracy is common for many *carrier-grade* switching hardware devices, as demonstrated in Sec. 4.4.3. On the other hand, when the policing rate increases, the accuracy of Pica8 device also significantly increases. That is, for policing rates higher or equal to $1Mbps$, Pica8 device achieves higher accuracy compared to the DPDK-based solution.

The results for burst policing follow the same trend, thus, they are not shown.

4.6 Discussion

In the previous sections (i.e., in Sec. 4.5 and Sec. 4.5.3), it is shown that the *carrier-grade* switches suffer from some hardware limitations. Therefore, they exhibit inaccurate traffic policing when the configured traffic rate and burst values are low (e.g., rate policing is lower than $1Mbps$). Depending on the brand and model of the switch, these inaccuracies can occur for rate and/or burst policing. Two approaches can be followed to resolve these policing issues. Firstly, the hardware implementation of the switches can be further improved or better and potentially more expensive devices could

be used. Secondly, the policing inaccuracies could be modeled and accounted for in the network management frameworks. In the second case, accounting for these policing inaccuracies can lead to lower performance of such networks (e.g., lower network utilization).

To discover the impact of these policing limitations on the network performance (i.e., second case), the simulation tool presented as part of *Chameleon* [Van+20] system is used. *Chameleon* is a cloud provider system that delivers strict end-to-end delay guarantees to network flows. In *Chameleon*, a flow is characterized by a source-destination node pair, data rate, burst size, and the required delay. *Chameleon* relies on DNC theory, priority queuing, and a simple greedy algorithm to provide the delay-constrained path allocation to the incoming traffic flows, with no packet loss (more details are provided in the background section). In particular, given a network and a set of flows, we are interested to find out how the policing limitation of the switches can affect the utilization of the network. To do so, we use a 4-fat-tree network with 10 servers per rack, each hosting 10 virtual machines. The network switches are considered with four priority queues (each with different assigned delays), and 1 Gbps of the link bandwidth. The network flows are generated randomly according to service types used in our previous work [Van+20], normally distributed. The considered flows also include flows with low rate and burst requirements, thus, policing them might cause waste of resources. To show the impact of the hardware limitation, we compare four cases:

1. *Chameleon*: The policing is set to be 100% accurate (the perfect case).

2. *PICA-hw*: *Pica8 P3297* switches are deployed in the whole network. In this case, we consider the hardware limitation by setting the minimum burst size to 17 kbits (based on the results in Section 4.4.3.5). However, we assume that policing inaccuracies are resolved.

3. *PICA-hw-inacc.*: *Pica8 P3297* switches are considered similar to the previous scenario. In addition to the hardware limitations (i.e., setting the minimum allowed burst size to 17 kbits), according to Fig. 4.12b, the effect of burst policing inaccuracy is included.

4. *HP*: The network switches are considered to be HP E3800, which does not support burst policing. In this case, we generate a model based on Fig. 4.12c which translates the expected burst size and rate to the deployed policed configurations of the HP switches.

Considering these four cases, we compare the total number of accepted flows and the achieved network utilization in Fig. 4.16. Surprisingly, Fig. 4.16a shows that the cost of hardware limitation and policing inaccuracy can be very high if there are a lot of flows with low rate and burst requirements. In particular, compared to the perfect policed case (i.e., *Chameleon*), it can be seen that the network consisting of *Pica8* switches accepts around 50% fewer flows. Since the OF metering feature in *Pica8* switches cannot be configured with a lower burst size than 17 kbits, the flows with lower burst requirements are being policed with a higher burst. This leads to the waste of resources in the network, i.e., a lower number of accepted flows for *PICA-hw* case. Also, it can be seen that considering the burst policing inaccuracy (*PICA-hw-inacc.*) causes the network to accept even fewer flows than the *PICA-hw* case (overall around 35% of the perfect case). For the *HP* case, since it does not support configuring burst policing, the number of accepted flows is very low, around 2% of the perfect scenario. Similarly, the considered traffic policing limitations also decreased the network

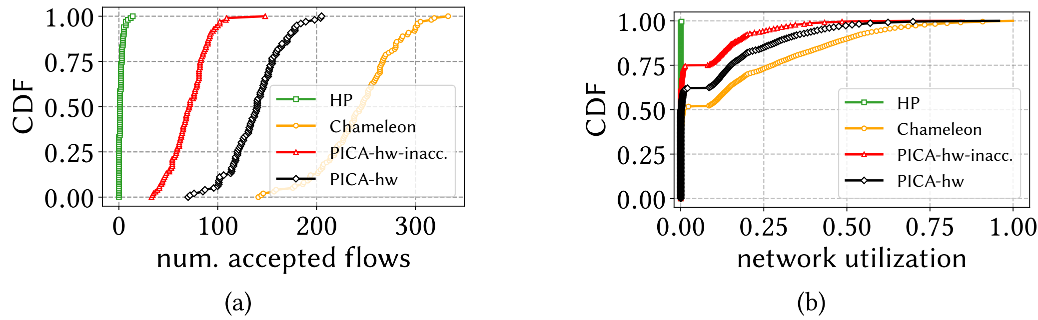


Figure 4.16: The role of accurate traffic policing in (a) number of accepted flows, and (b) network utilization.

utilization significantly (see Fig. 4.16b). These shocking values indicate that traffic policing plays an important role in the performance of the networks.

4.7 Conclusion

The contributions presented in this chapter represent one of the first steps towards measuring and modeling the performance of both, in-network and end-host based traffic policing. Initially, we have presented a generalizable DNC-based measurement methodology which can be used to determine and model the accuracy and precision of hardware- or software-based policing functions.

Afterwards, considering a set of modern *carrier-grade* OF switches, the impact of the policing on the packet processing time is measured. It was demonstrated that not all the evaluated switches (e.g., Dell S4048-ON) support OF traffic policing feature. For those who support, the policing feature has almost no impact on the packet processing time, which is interesting from the predictability point of view. However, we have observed these switches have some policing limitations, e.g., it is not possible to have a burst lower than a certain value in HP and *Pica8* brands. As a result, these switches may not be suitable for certain use-cases such as industrial networks, where usually the burst is small [GVK17]. Additionally, our measurements have shown that these switches do not perform the policing accurately, especially for traffic flows with low rate and burst size.

In addition, the implementation details of one end-host DPDK-based traffic policing function are presented and discussed. It was demonstrated that the aforementioned end-host traffic policing function offers higher policing accuracy for lower rates (e.g., $< 1Mbps$) compared to one *carrier-grade* switch. However, for higher rates the performance is reversed.

The study (in Section 4.6) was performed to find out what is the impact of traffic policing limitations and inaccuracies of *carrier-grade* switches in a realistic cloud network settings. To do so, the simulation tool presented in [Van+20] was used. The simulation results indicated that these limitations can actually have a significant impact on the network, especially in terms of the number of accepted flows and network utilization.

The contributions of this chapter represent one of the crucial building blocks for realizing QoS in virtualized programmable networks. That is, understanding how to model and measure the performance of traffic policing enables in virtualized programmable networks: *(1)* isolating the DP traffic belonging to different tenants, and *(2)* providing *end-to-end* DP guarantees to tenants.

4.8 Future Work

The contributions presented in this chapter represent one of the first steps towards understanding traffic policing and they open up several interesting avenues for future research. In particular, the burst and rate policing inaccuracies can be modeled and accounted for in the predictable network modeling frameworks, such as DNC. Moreover, it would be interesting to investigate if more accurate traffic policing can be realized with other programmable networking technologies, such as P4.

Moreover, a simulation study presented in the discussion section (see Sec. 4.6) indicates that the inaccuracies of traffic policing can have a significant impact on network utilization. Therefore, novel network management solutions which include and model these inaccuracies are needed. For example, including these inaccuracies in an admission control algorithm might increase the overall utilization of the network.

Chapter 5

Providing Control and Data Plane Guarantees in Programmable Networks

New technological frameworks and architectures such as Industry 4.0 [BAP17], 5G [PSS16], and even now 6G networks [Dan+20], let new applications appear on the horizon, such as remote vehicle control [FN12] or flexible factories. These applications often have very high Quality of Service (QoS) requirements. Firstly, they require the *end-to-end* per-packet Data Plane (DP) deterministic guarantees (e.g., bounded maximal packet delay, and no packet loss). Secondly, they also require consistent and timed network update guarantees from the Control Plane (CP). That is, a network should be updated in a consistent manner, without violating the guarantees of the already embedded DP flows. In addition, the time when a network is updated should be known. This is crucial in factories, where Automated Guided Vehicle (AGV)s typically have a predefined path which crosses many different wireless cells. Hence, they have to be often handovered, and during each handover procedure, it is crucial to ensure that the network is updated in a consistent and predictable manner. That is, all the rules that enable the connectivity should be established before an AGV is actually handovered. Failing to do so might result in connection interruption, and loss of control of AGV.

Naturally, to increase their revenue, virtualized programmable network should be able to support all kinds of services and applications. That is, they should also support traffic types which might require deterministic DP and CP guarantees. However, most of programmable network virtualization systems simply do not support such traffic types [She+09]; [Al-+14]; [Ble+16a]. In addition, the other *state-of-the-art* (*SotA*) systems that provide either deterministic DP guarantees or consistent network updates also do not offer adequate solutions. To the best of our knowledge, no system provides deterministic DP and consistent timed network update guarantees yet.

Therefore, in this chapter, we propose `NAGA`, a generic network architecture that integrates predictable (i.e., consistent and timed) CP operation into DP networks providing end-to-end latency and throughput guarantees. `NAGA` combines a centralized network logic with in-band network control; without the need for dedicated CP channels being physically isolated, `NAGA` revolves the need for dedicated control channels of many centralized concepts. Furthermore, to not limit itself to a close network area, such as data center networks, `NAGA` relies only on traffic control mechanisms as available in current programmable *SotA* switches. The measurement methodology presented in Chapter 4 enabled us to accurately measure the switches and to model them within the control logic

of NAGA. Therefore, in contrast to *SotA*, NAGA integrates the available features of programmable devices and puts them on the network edge; hence, it does not require full end-host control. Thus, NAGA can be integrated and deployed in virtualized programmable networks. This chapter documents the needed steps to realize a system with predictable DP and CP. It first reports on the system model followed by measurement results of existing programmable networking hardware. The measurement results show that existing hardware provides both predictable DP and CP operation nowadays, in contrast to existing Software-Defined Networking (SDN) devices which mostly rely on Open Flow (OF) [Van+19a]; [Bau+18]; [Kuź+18]. In addition, it uses network calculus to model and integrates predictable network control into programmable networks. Whereas simulation results provide a sensitivity analysis of NAGA for large-scale settings, a prototype implementation reveals that reconfiguring programmable networks in a timely and predictable way is indeed possible.

The content presented in this chapter is currently under submission:

- N. Đerić et al. "NAGA: A Deterministic Programmable Network with Update Timing Guarantees." Under Subimssion.

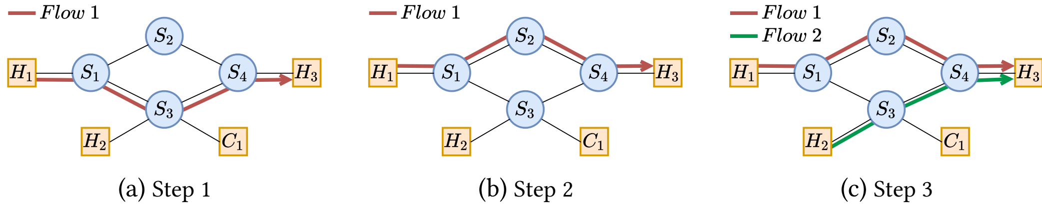
The structure of this chapter is as follows. Sec. 5.1 motivates NAGA in more detail. Sec. 5.2 introduces the system model and scheduling algorithm of NAGA . Sec. 5.3 reports on the predictability analysis of a carrier-grade programmable switch. Sec. 5.4 reports on the prototype implementation, evaluation and simulation results. Sec. 5.5 summarizes related work and Sec. 5.6 concludes the chapter and it outlines the possible future research directions.

5.1 Motivation and Contribution

SotA approaches [Van+19b]; [Van+20]; [Jan+15]; [Gro+15] which focus on achieving predictable latency suffer from two main issues: end-host-based control and lack of CP guarantees. While *SotA* consistent network update systems and solutions do not provide deterministic network update timing guarantees, hence, they are also not able to satisfy the stringent CP requirements. In the following, we explain these shortcomings in more detail, and in the end, we explain how NAGA resolves them.

1. End-host Control Assumption. *SotA* assumes that network operators can fully control end-hosts, which is, actually, a valid assumption in private Data Center (DC) networks. This consideration is made since *SotA* solutions [Van+20]; [Jan+15]; [Gro+15]; [Van+19b] meter the traffic at end-hosts (e.g., using Data Plane Development Kit (DPDK) or Linux TC). It ensures that the generated traffic follows the predefined characteristics (maximum flow rate and burst). Hence, it can be modeled by mathematical frameworks such as network calculus to provide deterministic end-to-end performance guarantees. However, full end-host control is not possible in non-data center scenarios. For example, in virtual networks [She+09]; [Al-+14] or Wide Area Network (WAN)s [Hon+13], it is not possible to control end-hosts, but only the forwarding devices in the network [She+09]; [Al-+14].

2. The Lack of Control Plane Guarantees. The second main issue with the *SotA* deterministic systems [Van+20]; [Jan+15]; [Gro+15]; [Van+19b] is that they do not consider CP at all. That is, they



(a) Step 1 (b) Step 2 (c) Step 3

Figure 5.1: An example of consistent reconfiguration procedure. Before embedding Flow 2, Flow 1 has to be rerouted consistently. C_1 is a centralized network controller running NAGA.

assume that there is an *out-of-band* CP network which distributes, inserts, and updates the forwarding rules reliably. In addition, some of the *SotA* systems (e.g., [Van+20]) use reconfiguration algorithms in order to achieve higher network utilization. In such systems, the already embedded flows might block the insertion of new arriving flows. Thus, reconfiguration algorithms (e.g., [Van+20]) aim to re-route already-embedded flows in order to make space for new flows. For example, in Fig 5.1, it is crucial to ensure that Flow 1 is re-routed before adding Flow 2. Hence, either the end-hosts or the forwarding devices must be reconfigured consistently, which is impossible without considering the CP at all. Any inconsistencies can lead to performance violations in the DP, which is not acceptable in deterministic networks.

3. Network Update Timing Guarantees. Most of the *SotA* consistent network update systems and solutions [Zho+21]; [NCC17]; [Jin+14] focus only on providing consistent updates, hence, they do not provide any network update timing guarantees. While some solutions [MSM15] provide stochastic update timing guarantees (e.g., 99.9^{th} percentile), they are designed for non-deterministic (control) networks. Thus, this group of *SotA* works simply cannot satisfy the aforementioned stringent CP requirements.

Proposed solutions in NAGA. To avoid utilizing end-hosts, in contrast to *SotA*, NAGA places all important functionalities (e.g., traffic metering) on the edge switches of a network. In Sec. 5.3, we demonstrate that indeed *SotA* programmable switches exist that provide the needed functionalities in a predictable (deterministic) way. Therefore, NAGA is a system that provides predictable end-to-end guarantees (both in DP and CP) without needing to control the end-hosts.

To provide deterministic network update time guarantees, NAGA relies on an in-band CP with performance guarantees, using the network calculus framework [Cru91a] to ensure that the CP rules are never lost and delivered predictably (see Sec. 5.2). Secondly, we present a scheduling algorithm that considers the processing time of adding/reconfiguring rules of the forwarding devices (see Sec. 5.2.2) in order to guarantee the predictability (with respect to timing) and consistency of the network updates (see Sec. 5.2).

5.2 System Model

This section outlines the NAGA's model. We consider a label-based network (based on Virtual Local Area Network (VLAN) tags) with two types of switches: edge and transit. Edge switches meter and tag all (added tags determine the path of a flow) packets coming from the connected hosts and forward them to the next hop in the network (according to the routing decision, determined by a

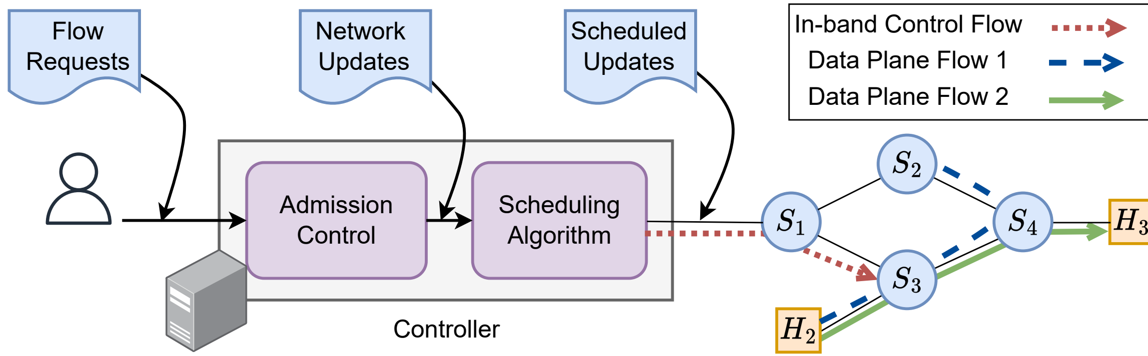


Figure 5.2: NAGA system model. Example: The controller configures S_3 to embed a flow (2) between H_2 and H_3 by sending a rule via in-band CP flow.

controller). Moreover, they also perform label-based forwarding for the traffic flows coming from the other network switches. The transit switches perform only the label-based forwarding. In fact, upon receiving a packet, they pop the outer tag and forward them accordingly. Further, transit switches are configured only once when bootstrapping the network. For example, in Fig. 5.2, S_1, S_3, and S_4 are edge switches, while S_2 is a transit switch (no directly attached host). The tagging rules are distributed on-demand by the centralized controller (or controller) to each edge switch via an in-band CP.

Multiple *SotA* deterministic [Van+20] and network update [Jin+14] systems consider similar network types as it simplifies network management. To be precise, in such networks (including label-based), reconfiguring a single flow requires only changing or modifying a single rule on one edge switch. Therefore, performing network updates (or sequence of flow reconfigurations) becomes less challenging compared to a classical SDN network. Moreover, such networks are more easily managed by a distributed SDN CP (needed for scalability), as network partition can be done based on edge switches.

Bootstrapping the in-band CP. To distribute (on-demand) flow updates to the edge switches, we use an in-band CP. Additionally, we propose to use flow embedding solutions that provide deterministic per-packet guarantees (i.e., bounded delay, no packet loss) for provisioning the in-band CP. To be precise, in this paper we re-use one *SotA* solution presented in the next subsection. Having deterministic guarantees on CP packets enable us to develop a novel network scheduling algorithm (presented in the two following sections) which ensures a predictable (guaranteed timing) and consistent update of forwarding rules.

Architecture & online flow Admission Control (AC). During the runtime, a controller manages the network. It listens to *end-to-end* flow requests and tries to embed them (Fig. 5.2). The controller uses the AC algorithm proposed in *Chameleon* [Van+20] (implementation is available on github [Van20]). This algorithm embeds flows in the network in an online manner by considering a queue-level topology [GVK17], and LARAC [Jut+01] algorithm for the routing. Further, it employs the Deterministic Network Calculus (DNC) framework [LT01]; [Cru91a]; [Cru91b] to provide deterministic guarantees. Moreover, by utilizing flow reconfigurations, it achieves higher network utilization compared to other similar *SotA* solutions [Gro+15]; [Jan+15].

Each flow request has the following parameters: *1)* a source host id, *2)* a destination host id, *3)* a maximal allowed delay, *4)* rate and burst requirements, and *5)* a unique *5-tuple* as the flow identifier (src. Internet Protocol (IP), dst. IP, network protocol, src. port, dst. port). After processing each flow request, the considered AC algorithm outputs the decision if the corresponding flow can be embedded into the network. If the flow can be embedded, the algorithm outputs a *sequence of the required flow updates* to embed the corresponding flow (see Fig. 5.2). We consider this *sequence of flow updates* as a single *network update*. The output flow update sequence can be of two types: *(1)* without required reconfigurations and *(2)* with required reconfigurations.

In the first case, the algorithm's output is a sequence of flow updates (with size one) that contains only one flow insertion (no needed reconfigurations). In this case, the controller has to insert only one tagging and metering rule on the corresponding edge switch, e.g., the edge switch S_3 for flow 2 in Fig. 5.2.

In the second case, the algorithm outputs a sequence of flow reconfigurations (i.e., rerouting), followed by a single flow insertion. Thus, before embedding the flow request (i.e., inserting the forwarding rule), the controller reroutes the corresponding sequence of flows in the network to free up the occupied resources. For example (see Fig. 5.2), before embedding *Flow 2*, it is necessary to reroute *Flow 1*. To do so, the controller has to first modify the already inserted tagging and metering rule of *Flow 1* on the edge switch S_3. After that, it inserts the rule of *Flow 2* on the edge switch S_3. To know when to add *Flow 2*, the reconfiguration time of *Flow 1* must be predictable.

To solve this problem, we propose a novel scheduling algorithm in Sec. 5.2.2 which provides timing guarantees and consistent network updates by utilizing deterministic in-band CP. Briefly, the algorithm processes the output of AC and schedules the sending time of each flow update from the controller to the corresponding edge switch.

5.2.1 Achieving CP Consistency

To achieve predictable network updates, we use a similar approach as presented in Qjump [Gro+15]. We send CP packets in a uniform manner that matches the rule insertion or reconfiguration time of the edge switches (the behavior of the edge switch is measured in Sec. 5.3). Thus, the CP queues of edge switches are never filled up with more than one control packet. In addition, we also always send exactly one flow update per CP packet.

The scheduling algorithm presented in the next section relies on the two constraints that are presented in the following. The first constraint ensures that a single edge switch is never overloaded with more than one CP packet at any time. While the goal of the second constraint is to ensure that the dependent flow updates involving multiple switches are processed consistently.

Constraint 1: We first derive how often (i.e., separated uniformly with time interval δ) the CP packets should be sent from the controller towards an edge switch without filling up its CP queue. Before elaborating on the example, we define the format of the CP packets. The payload of each CP packet contains a single flow update message. These update messages are denoted by $u_m = (a_m, S_m)$, where u_m is the mth flow update, a_m is its match and action configuration, and S_m is the corresponding edge switch that should process the flow update. We explain our approach using an example and a message sequence diagram respectively depicted in Fig. 5.3. Given an empty

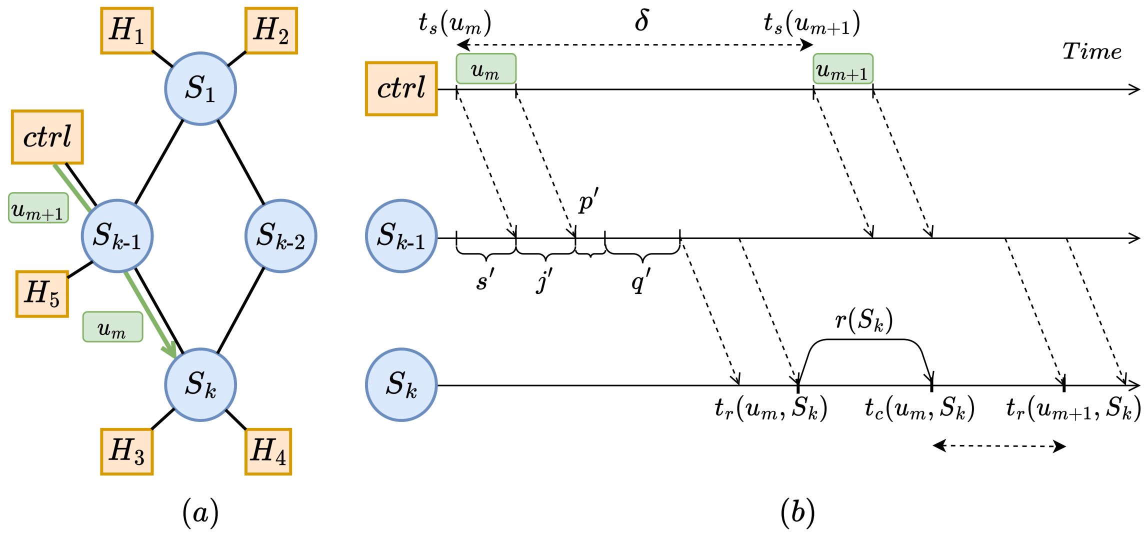


Figure 5.3: (a) Network example and (b) the respective CP message sequence chart. Two flow updates (e.g., u_m and u_{m+1}) are sent from the controller to the edge switch S_k. s', j', p', q' represent the total propagation, sending, processing, and queuing time update u_m experienced while traversing S_{k-1}.

network, let us assume that the controller aims to send two flow updates $u_m = (a_m, S_m)$ and $u_{m+1} = (a_{m+1}, S_{m+1})$ to the same edge switches (i.e., $S_m = S_{m+1} = S_k$). For example, u_m (u_{m+1}) could enable a flow between H_3 and H_1 (H_4 and H_2). Each flow update is encoded into payload of on CP packet. During the transmission of a CP packet from the controller to an edge switch, the packet can be delayed at various points in the network. For instance, the packet has to be received, processed, and en-queued at each transit switch on the path. Therefore, the update packet u_m will be received by edge switch S_m at time $t_r(u_m, S_m)$:

$$t_r(u_m, S_m) = t_s(u_m) + s(u_m) + k(u_m) + p(u_m) + q(u_m), \tag{5.1}$$

where $t_s(u_m)$ is the time when the controller sent the update packet u_m, and $s(u_m)$, $k(u_m)$, $p(u_m)$, and $q(u_m)$ are the total duration of sending time, propagation time, switch DP processing time (for packet forwarding), and queuing time that update packet u_m experienced in the network while going from the controller to the edge switch S_m. The sending time of each packet at each switch is calculated by dividing the packet size by the link rate. The total propagation time is derived based on the total path length between the controller and the corresponding edge switch. Moreover, the total DP packet processing time is calculated by summing up the value for each switch on the path between the controller and the corresponding edge switch. Different hardware switches usually exhibit different DP packet forwarding processing times [Van+19a]. For shortness, instead of writing (low-variable parameters) $s(u_m) + k(u_m) + p(u_m)$, in the following we will use simply write $z(u_m)$, where $z(u_m) = s(u_m) + k(u_m) + p(u_m)$. Eq. 5.1 can be re-written to:

$$t_r(u_m, S_m) \in \left[t_s(u_m) + min\Big(z(u_m)\Big), t_s(u_m) + max\Big(z(u_m) + q(u_m)\Big) \right]. \tag{5.2}$$

Note: The highest priority queues in the network are governed by DNC, thus, the queues of the switches will never be overloaded. That is, there will be no packet drops caused by traffic bursts and insufficient buffer space.

Further, after an edge switch receives the CP packet (containing one flow update), inserting (or reconfiguring) the corresponding rule takes some time. As Section 5.3.5 shows, for an edge switch S_m, this value is predictable and can be upper-bounded with a function $r(S_m)$. Henceforth, using the Eq. 5.2, the time when the edge switch S_m completes receiving and processing the flow update u_m can be bounded, and is denoted by $t_c(u_m, S_m)$:

$$t_c(u_m, S_m) \in \left[t_s(u_m) + min\Big(z(u_m)\Big), t_s(u_1) + max\Big(z(u_m) + q(S_m)\Big) + r(S_m)\right]. \tag{5.3}$$

To avoid overloading the CP queue of the edge switch with more than one packet, the next subsequently sent CP packet (u_{m+1} in our example) by the controller should arrive right after the edge switch finished processing the previous one (u_m in our example). This is satisfied when $t_r(u_{m+1}, S_{m+1}) - t_c(u_m, S_m) \geq 0$ (note that $S_m = S_{m+1} = S_k$). The worst-case is obtained when the first CP packet experiences the maximum amount of queuing, and the second one none, i.e., $min(t_r(u_{m+1}, S_k)) - max(t_c(u_m, S_k)) \geq 0$. By using Eq. 5.2 and Eq. 5.3, the first constraint can be written as:

$$t_s(u_{m+1}) + min\Big(z(u_{m+1})\Big) - t_s(u_m) - max\Big(z(u_m) + q(S_m)\Big) - r(S_m) \geq 0. \tag{5.4}$$

To ensure that the edge switch $S_m = S_{m+1} = S_k$ always has enough time to receive and process the previous CP packet containing the update (i.e., u_m), the controller should wait at least $\delta(S_m, S_{m+1})$ before sending the following CP packet (i.e., u_{m+1}) as in the following.

$$t_s(u_{m+1}) - t_s(u_m) \geq max\Big(z(u_m) + q(S_m)\Big) + r(S_m) - min\Big(z(u_{m+1})\Big) = \delta(S_m, S_{m+1}). \tag{5.5}$$

The first constraint can also be written as:

$$t_s(u_{m+1}) \geq t_s(u_m) + \delta(S_m, S_{m+1}). \tag{5.6}$$

Therefore, if the controller already sent flow update u_m at $t_s(u_m)$, the sending time of the following update $t_s(u_{m+1})$ should satisfy Eq. 5.6 to ensure that the CP queue of the edge switch is never utilized with more than one packet.

First, most of the variables (e.g., sending time) contained in z exhibit very low variance (e.g, propagation time can be assumed to be constant). Second, the two flow updates are sent to the same edge switch, i.e., $S_m = S_{m+1} = S_k$. Therefore, it can be assumed that $max(z(u_m)) - min(z(u_{m+1})) \approx 0$ (e.g., propagation times to the same switch cancel out). Hence, making $\delta(S_k, S_k) \approx max\big(q(S_k)\big) + r(S_k)$. The maximal total queuing time can be obtained from the considered DNC-based AC algorithm. E.g., since the flows are mapped to the highest priority queue, the total maximal queuing time is simply the number of hops multiplied by the maximum delay of the highest priority queue (i.e., in our case $0.1ms$). Hence, if there are 5 hops on the path to the corresponding switch, and the maximal edge switch reconfiguration time is $r(S_k) = 2ms$, $\delta(S_k, S_k) \approx 5 \times 0.1ms + 2.0ms = 2.5ms$.

Constraint 2: The second constraint ensures consistent network updates in scenarios when a controller wants to send two flow updates to two different edge switches, and the second update is dependent on the first one. That is, in such scenarios, the first update must be received and applied before the second one. To achieve this, we determine how much two CP packets (destinations are two

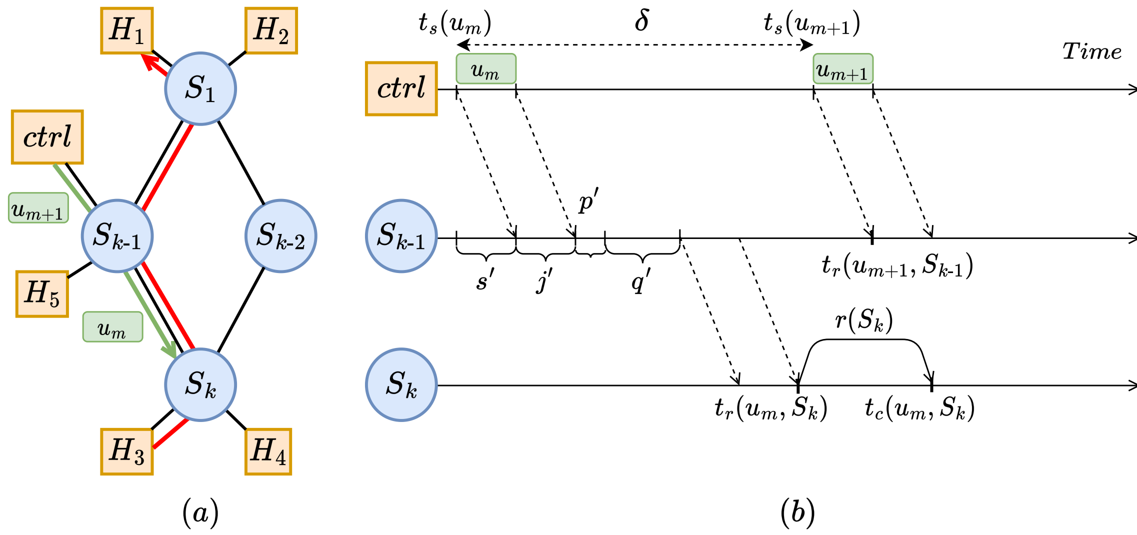


Figure 5.4: (a) Network example and (b) the respective CP message sequence chart.

different edge switches) have to be spaced in time to guarantee that the network update consistency is achieved.

Let us assume that there is one flow in a network, i.e., *Flow 1* between hosts H_3 and H_1 (see Fig. 5.4). The controller aims to schedule sending time of two flow updates $u_m = (a_m, S_m)$ and $u_{m+1} = (a_{m+1}, S_{m+1})$ to two different edge switches (i.e., $S_m = S_k$ and $S_{m+1} = S_{k\text{-}1}$). In our example, the first update (i.e., u_m) aims to reroute *Flow 1* from path S_k, S_{k-1}, S_1 to S_k, S_{k-2}, S_1, in order to make space for *Flow 2* (contained in u_{m+1}). Hence, flow update u_{m+1} is dependant on u_m.

From timing perspective, if we want to ensure that flow update u_{m+1} is applied after u_m, we can reuse Eq. 5.5. However, in this case, it is not possible to assume that $max(z(u_m)) \approx max(z(u_{m+1}))$ (as e.g., propagation times are different). In addition, since *Flow 1* (before reconfiguration) and *Flow 2* share the same link (i.e., link S_{k-1}, S_1), it also necessary to ensure that the packets which were on the previous path of *Flow 1* completed the transmission. That is, the previous path should not contain residual packets (e.g., path S_k, S_{k-1}, S_1 in our example). This can be ensured by simply adding the maximal guaranteed end-to-end delay $d(u_m)$ of *Flow 1* to Eq. 5.5. Therefore, the second constraint can be formulated as follows:

$$t_s(u_{m+1}) \geq t_s(u_m) + \delta(S_m, S_{m+1}) + d(u_m). \tag{5.7}$$

The value of $d(u_m)$ is provided by AC algorithms. This is also valid for other *SotA* deterministic systems.

5.2.2 Scheduling Algorithm

This section presents our network update scheduling algorithm (i.e., Alg. 2) that provides consistent and timed updates.

Parameters. We denote the number of edge switches in the network with k. Parameter $\delta(S_x, S_y); \forall x, y \in \{1, .., k\}$ is calculated based on Eq. 5.5, the properties of the network (e.g., distances between switches), and the type of deployed switch (e.g., the processing time is switch-dependent).

Algorithm 2 Scheduling Algorithm

```
1:  Parameters: δ(S_x, S_y), d_c(S_x), r(S_x); ∀x, y ∈ {1, .., k}
2:  States: 𝓕, T(S_x); ∀x ∈ {1, .., k}
3:  Routine:
4:  while true do
5:      𝓡 ← GETFLOWUPDATES()
6:      for i in [1, .., |𝓡|] do
7:          S_x ← GETDST(𝓡[i])
8:          t_1 ← T(S_x) + δ(S_x, S_x)                          ▷ Constraint 1
9:          t_2^max ← 0
10:         f ← GETRECONFIGURATIONS(𝓕, t_1 − t_f^max)
11:         for j in [1, .., |f|] do
12:             if DEPENDENT(𝓡[i], f[j]) then
13:                 S_y ← GETDST(f[j])
14:                 t_s ← GETSCHEDTIME(f[j])
15:                 d_d ← CALCFLOWDRAIN(f[j])
16:                 t_2 ← t_s + δ(S_y, S_x) + d_d               ▷ Constraint 2
17:                 if t_2 > t_2^max then
18:                     t_2^max ← t_2
19:         t_send ← MAX(t_now, t_1, t_2^max)
20:         SCHEDULE(𝓡[i], t_send)                              ▷ Schedule Flow Update
21:         T(S_x) ← t_send
22:         t_g ← t_send + d_c + r(S_x)                         ▷ Calculate Time Guarantee
23:         if ISRECONFIGURATION(𝓡[i]) then
24:             𝓕.INSERT(t_send, 𝓡[i])
```

The next input is $r(S_x)$ which shows the maximal reconfiguration time for edge switch S_x. Moreover, $d_c(S_x)$ is the maximal guaranteed delay for each in-band CP flow between the controller and each edge switch $S_x \in \{1, .., k\}$. These values are obtained from the performance measurements of network switches and the admission control algorithm.

States. We use dictionary T with edge switch IDs as keys and values being the time that the last CP message was sent towards that edge switch (Lns. 1-2). Additionally, $\mathcal{F}$ is a set of already-scheduled reconfigurations.

Routine. The scheduling algorithm runs an infinite loop (Ln. 4) and listens for the output of the AC algorithm, which is a sequence of flow updates (Ln. 5). In our case, a sequence of n flow updates contains $n - 1$ flow reconfigurations followed by 1 flow insertion. After receiving a sequence of updates, the scheduling algorithm processes them sequentially in the same manner (Ln. 6).

For each update $\mathcal{R}[i], i \in \{1, .., n\}$, the algorithm detects which edge switch it should send the update to (i.e., S_x) (Ln. 7). Afterwards, it checks when was the last CP message sent $T(S_x)$ to the corresponding edge switch S_x and adds the value of parameter $\delta(S_x, S_x)$ to it (Ln. 8). This time (i.e., t_1) represents the earliest possible sending time of a CP message containing the update while ensuring that the CP queue on the corresponding edge switch is not over-utilized (see **Constraint 1** in Sec. 5.2.1).

In the following part of the algorithm (Lns. 10-18), it is investigated if the currently processed update (i.e., $\mathcal{R}[i]$) depends on some of the previous ones. The flow updates can only be dependent on the previous flow reconfigurations (not insertions). That is, we need to ensure that the dependent rerouted flows are fully drained from the network before the flow in hand is updated (for the sake of consistency). To capture all the relevant reconfigurations f, we first load all the scheduled recon-

figurations from set $\mathcal{F}$ which occurred after time $Min(t_1, t_{now}) - t_f^{max}$ (Ln. 10). t_f^{max} represents the maximum time duration that ensures Constraint 2 is satisfied for all scenarios. Its value is calculated as $t_f^{max} = Max(\delta(S_x, S_y) + d_d), \forall x, y \in \{1, .., k\}$, where d_d is maximum end-to-end delay between two nodes in the network. The value of d_d can be calculated with well-known algorithms which can compute the longest path in a graph with no loops. Afterward, all the selected reconfigurations f are processed sequentially in a for-loop in the same manner. For each $f[j]$, initially, it is checked if the current flow update $\mathcal{R}[i]$ is dependent on it (Ln. 12). We consider them dependent if they share at least one edge in queue-level topology.

If they are dependent, the scheduling algorithm ensures that also the second constraint for flow update $\mathcal{R}[i]$ and the already scheduled reconfiguration $f[j]$ is met (Lns. 13-16). To do so, first, the edge switch (i.e., S_y) and the already scheduled time (i.e., t_s) of $f[j]$ are determined (Lns. 13-14). Secondly, the flow draining time which corresponds to the max DP delay of $f[j]$ is read (Ln. 14). Thereafter, the earliest possible sending time (i.e., t_2) based on the second constraint for the two updates ($\mathcal{R}[i]$ and $f[j]$) can be calculated with Eq. 5.7 (Ln. 16). The maximal t_2 earliest possible sending time is saved (Lns. 17-18), to ensure that the second constraint is met between all the dependent reconfiguration f and the currently processed flow update $\mathcal{R}[i]$.

The CP packet containing flow update $\mathcal{R}[i]$ is scheduled based on the maximal value of t_{now}, t_1, and t_2^{max} (where t_{now} is the current time). This ensures that all of the constraints are met. Additionally, the dictionary T is updated accordingly (Ln. 21).

After the scheduling time of a flow update is determined, the algorithm can derive the upper bound of flow update completion time. This is done by adding the maximal time needed to deliver the CP message (containing the update) to an edge switch (i.e., d_c) and maximal reconfiguration time (i.e., $r(S_x)$) of the corresponding switch to the scheduled sending time. Finally, if the scheduled update is a reconfiguration, it is inserted into the list of potential dependent updates $\mathcal{F}$ (Lns. 23-24).

Remarks. In some use cases, many flows might not require such strict guarantees, and provisioning them as such might lead to a waste of networking resources. To cope with this situation, `NAGA` can be adapted to support non-deterministic DP traffic. To do so, a deterministic AC can be used to manage only the highest strict priority queues of all networking devices for the initial bootstrapping of the in-band CP. All the other flows could be embedded with any other non-deterministic AC solution considering all the other strict priority queues (except the highest one). The only requirement here is the switches need to support priority queuing (at least 2 of them), and label-based forwarding (at least for CP traffic).

Further, in practice, packets can be lost due to the low occurring external factors (e.g., noise caused by electromagnetic radiation). Some sources [Err] estimate that the probability of losing a $1.5kB$ packet is around 10^{-9}. Hence, to minimize these effects and to protect the in-band CP, two following approaches can be taken (apart from using better error correction mechanisms). Firstly, instead of having one in-band CP flow towards each edge device, we could have k (disjoint) paths which carry the same CP rules (helps also with failures). Secondly, instead of sending one CP packet per rule, it could be possible to send N copies. Doing so would reduce the probability of such events drastically as both of these approaches represent parallel redundancy (if we consider that external factors cause independent packet loss). However, since during our experiments we never observed a packet loss caused by such factors, we did not include such protection mechanisms in `NAGA`.

5.3 Edge Switch Predictability Measurement

To realize `NAGA`, firstly, the transit switches have to perform deterministic label-based forwarding at line rate. According to a recent study [Van+19a], most carrier-grade programmable switches can satisfy this requirement. Secondly, in addition to deterministic forwarding, the edge switches must be able to tag and meter the network traffic. Moreover, they must exhibit predictable in-band CP operations, such as adding and reconfiguring the forwarding rules during the runtime. To fill this gap, we perform a comprehensive measurement study in this section.

In this part, we introduce the chosen edge device and our implementation in Sec. 5.3.1. We start with studying the predictability of the DP, which includes measuring the behavior of processing time (see Sec. 5.3.2), traffic metering (see Sec. 5.3.3), and buffer management (see Sec. 5.3.4). After that, we assess the measurement of CP predictability in Sec. 5.3.5.

5.3.1 Edge Switch Deployment

In this work, we propose to use Programming Protocol-independent Packet Processors (P4)-enabled Edgecore Wedge 100BF-32X System [Net21] as the edge switch (or shortly EdgeCore Wedge), powered by P4-programmable Intel Tofino^{TM} Intelligent Fabric Processor (IFP) ASIC (or shortly Intel Tofino^{TM}). We develop a network application, consisting of a P4 program (with just three tables in the ingress pipeline) and a local controller, written in Python. The logical representation is shown in Fig. 5.5. We first explain each of the tables and their roles and then introduce the developed Python application.

(1) VLAN-forwarding-tbl. All the packets received by the edge switch are processed by this table first. The main goal of this table is to implement label-based forwarding for all packets which are already tagged by the other edge switches. This table matches the packets based on the $vlan_id$ field of each packet's outer VLAN tag. Additionally, each rule has the same action type, i.e., to forward the packets on a certain port and priority queue (based on the $vlan_id$ field). We use the following notation to map $vlan_id$ field to the port and priority queue – $vlan_id = 10 \times port_id + queue_id$. For example, if a received packet has an outer VLAN tag with $vlan_id = 1405$, it will be forwarded to port 140 and priority queue 5. If a packet matches a rule in this table, it will not be processed by the other tables. Otherwise, it will be processed by the next table.

(2) tagging-tbl. The main goal of this table is to tag all the packets which did not match the first table. It means that these packets are coming from the directly connected hosts to the edge switch. This table matches the packets based on the five different header fields: `ip_src, ip_dst, l3_network_proto, l3_src_port, l3_dst_port`. Each matched packet is then tagged with up to 10 VLAN tags and forwarded to a certain port and queue (based on the corresponding rule). Moreover, each rule also meters the traffic, meaning that it colors the packets with either green, yellow, or red color according to the configured metering parameters and the token bucket state. Based on the colors assigned to each packet, the flows can be later on rate and burst limited. Additionally, this table contains exactly one special rule used to forward the in-band CP packets to the Central Processing Unit (CPU) port of a switch. If a packet misses the table, it will be dropped, while the metering table processes the matching packets (excluding control).

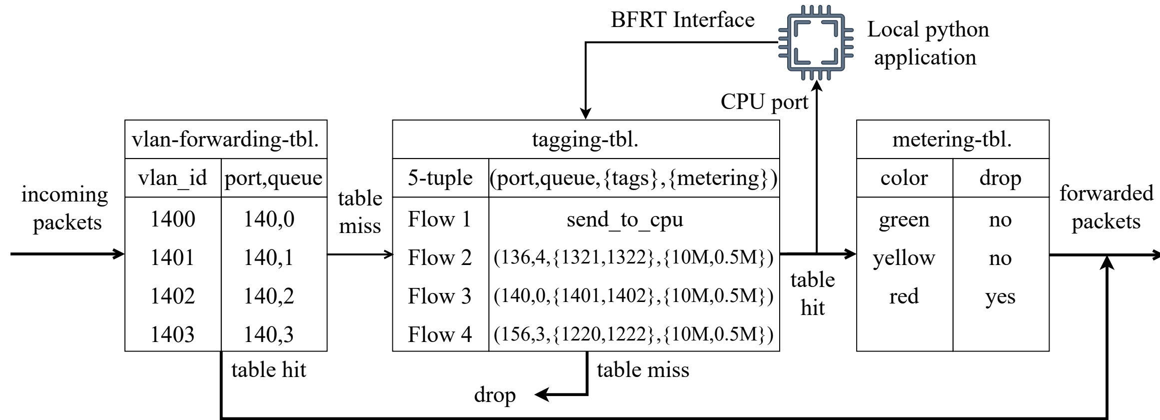


Figure 5.5: Logical representation of the developed P4 application and a local python application.

(3) metering-tbl. The main purpose of this table is to decide if the packet should be forwarded or dropped according to its color (determined by *tagging-tbl*). The packets colored in red are dropped, while the other ones are forwarded.

Local Control Plane Application.

We deploy an Ubuntu 18.04 and Intel P4 Studio™ SDE locally on the device. The in-band CP rules are forwarded during the runtime from the DP to the local CP on the switch (running Ubuntu) and then inserted into the *tagging-tbl.* To do so, the developed P4 application forwards the matching in-band CP packets from the DP to the CPU port of a switch. This CPU port appears as a network interface in the local CP. Hence, listening on the network interface representing the CPU port makes it possible to receive the in-band CP messages in the local CP (running Ubuntu). To listen (and react) to the in-band rules, we develop a Python application with a generic *socket* library and Intel's *BFRT* Python library. After decoding the CP rules, we utilize the developed application and *bfshell* program (provided by Intel as part of P4 Studio™ SDE) to insert or reconfigure the rules to the *tagging-tbl* table. Each flow rule consists of five fields: *(1)* 5-tuple match, *(2)* forwarding port and priority queue number, *(3) vlan_ids* to be added, *(4)* metering parameters (in kbps for rate, and kb for burst), and *(5)* a flag indicating if the rule is new (i.e., it does not exist in the *tagging-tbl*), or a reconfiguration of the already existing rule. To list all the values of these fields with simple ASCII encoding, we need in total 42B. Therefore, in a 1500B CP packet, the controller can fit up to 34 different rules. In our scenario, the controller sequentially writes the rules represented as plain text in the User Datagram Protocol (UDP) payload of each CP packet.

5.3.2 Data Plane Processing Time

Setup and Procedure. The testbed (Fig. 5.6) consists of: *(1)* a Device Under Test (DUT), i.e., EdgeCore Wedge, *(2)* a 10G optical tap, and *(3)* two servers: first one as the traffic generator with DPDK-based Moongen [Emm+15] running Ubuntu 18.04 and equipped with Intel Xeon E5-2650 v4 @2.2 GHz CPU and an Intel X520 Network Card Interface (NIC), and second one for capturing the traffic using a 4-port 10G nanosecond-precise Endace DAG 10X4-S measurement card. EdgeCore Wedge has 32 QSFP cages, where each connector can be used in 100G/40G mode, or it can be split into four 10/25G ports. Since the measurement card has only 10G ports, we split each QSFP port

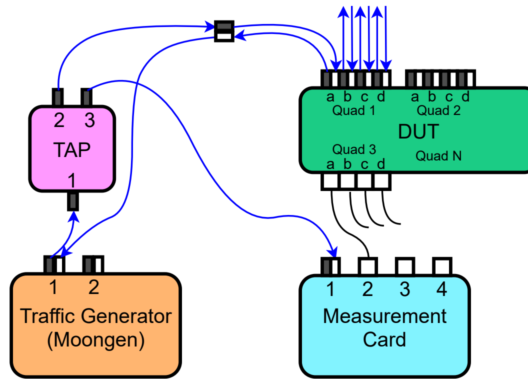


Figure 5.6: Processing time measurement setup.

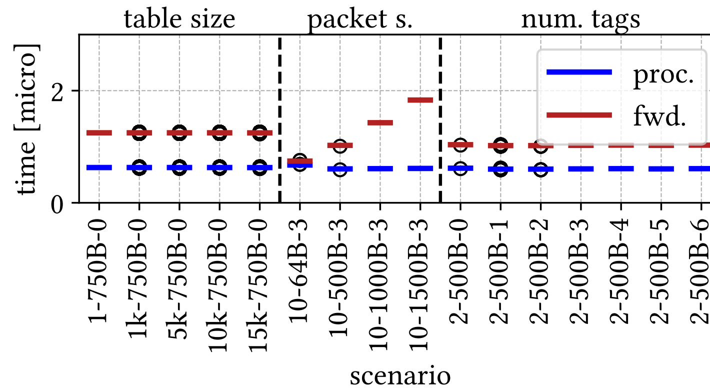


Figure 5.7: The total forwarding and processing time of the tagging and metering application on EdgeCore Wedge.

either with optical (blue lines connected to Quad 1 on Fig. 5.6) or copper (black lines) breakout cables. The generated traffic is firstly split (or replicated) by an optical tap into two different fibers (each fiber operates with 50% of power of the original signal). Since we use single-mode optical fibers (in this case, a pair of fibers is needed for one link), we split the traffic only on the sending fiber (i.e., on the fiber carrying the traffic generated by the traffic generator). The fiber from the Tap device port 3 is then connected to the measurement card port 1. By doing so, it is possible to obtain the timestamps of the sending traffic. Meanwhile, a fiber from port 2 of the measurement card is connected to the DUT through an optical breakout cable and one optical coupler. The generated traffic is then tagged, metered, and forwarded by EdgeCore Wedge accordingly (e.g., from Quad 1a to Quad 3a). Finally, Quad 3a is connected with a copper breakout cable to the measurement card, obtaining post-processing timestamps. By subtracting the sending timestamp from the post-processing timestamp, we can derive the total forwarding time of a packet. After that, the processing time is calculated by subtracting the packet sending time (i.e., packet size divided by link rate) from the total forwarding time.

To evaluate the DP processing time, we generate 30k packets with a variable size, at either a high rate (10G) or at a low rate (0.5-12Mbps). In case of a low rate, the packets are uniformly spaced with 1ms. We consider both high and low rates as the performance of some switches can depend on the rate. We generate 16 different scenarios, where the packet size is varied from 64 to 1500B, the

number of flow entries from 1 to 15k, and the number of added VLAN tags from 0 to 6. The exact parameter values of each scenario are presented as labels of the x-axis in Fig. 5.7. Each scenario label consists of three numbers (e.g., 1k-750B-0), *i)* the number of flows in the forwarding table (e.g., 1k), *ii)* the packet size (e.g., 750B), and *iii)* the number of added tags (e.g., 0).

Results. Fig. 5.7 illustrates both total forwarding time and processing time of EdgeCore Wedge running our P4 application (see Sec. 5.3.1). Overall, the forwarding time depends on the packet size. The derived processing time is very predictable (i.e., bound) and always between 600 and 690ns. This observation is expected, as the DP predictability of programmable hardware has been recently confirmed by a comprehensive measurement study [Bau+18].

5.3.3 Traffic Metering

Setup and Procedure. The NAGA architecture performs traffic policing at the edge of the network. The goal of traffic policing is to limit the flow to the maximal rate r and burst b. We use the native hardware meters of the EdgeCore Wedge powered by Intel Tofino™ which utilize a dual token bucket algorithm and three-color packet marking. For evaluating their accuracy, we use the measurement setup presented in Fig. 5.6, and a traffic generation procedure similar to a recent *SotA* work [Đer+21]. In this experiment, the various states of a hardware meter (e.g., number of tokens in a bucket) are tested by sending randomly-spaced packet bursts to the DUT. We consider the combination of the following parameters: number of meters as {1,10,100,1000}, configured rate as {1,10,50,100,150,200}(Mbps), burst {100,200,300,400,500,600,700,800,900,1000}(kb), and the packet size as {100, 600, 1100, 1500}(B). Also, the action is considered as *forward_to_port*. To derive the maximum rate and burst of a metered flow from the measurements, we utilize a similar methodology presented in [Đer+21], which is based on network calculus framework [LT01].

Results. Fig. 5.8 and 5.9 illustrate the mean accuracy of traffic policing while varying the configured rate and burst size. In the case of rate policing, it can be seen that the accuracy of the meter is very high and constant as the observed relative error is at most 0.4% (see Fig. 5.8). Moreover, all the considered parameters did not significantly impact the accuracy of rate policing. Regarding the burst metering, Fig. 5.9 shows that the accuracy is indeed very high. For higher configuration values ($b \geq$ 200kb), the error is less than 1%, while for the smaller burst values $b =$ 100kb, the worst-case relative error is around 10%. To account for this error in NAGA, for each end-to-end Northbound Interface (NBI) request, we reserve 10% higher metering values. Overall, EdgeCore Wedge powered by Intel Tofino™ offers a sufficient amount of hardware meters with a high accuracy, which makes it a suitable candidate for policing the traffic at the edge[1].

5.3.4 Buffer Management and Priority Queuing

As another requirement for deterministic guarantees for data and CP, we need to ensure that all forwarding devices buffer and enqueue network traffic in a predictable manner [LT01] (e.g., packets are correctly enqueued and not dropped). In NAGA , we utilize a DNC-based framework (see Sec. 5.2),

[1]Recently, many shaping-based approaches are developed which can offer even greater precision while avoiding packet drops [Shi20]. It is crucial to note that our focus in this paper is not to find the most accurate traffic limiting approach; hence, we consider this as an orthogonal problem.

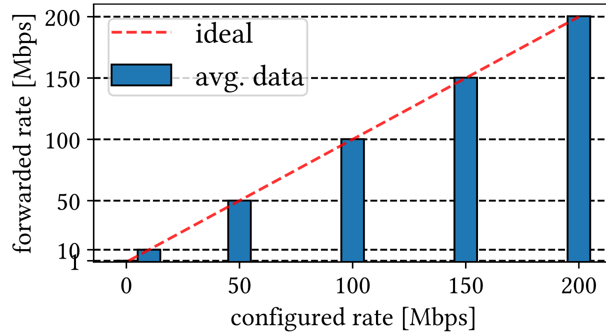


Figure 5.8: Rate metering accuracy.

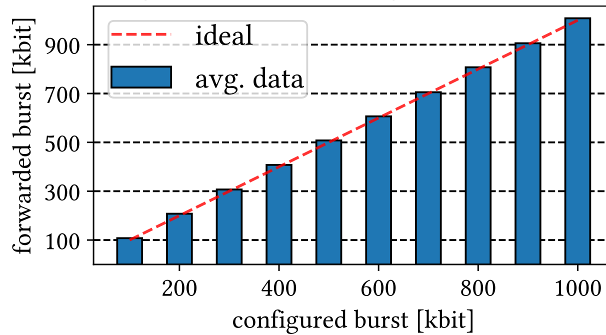


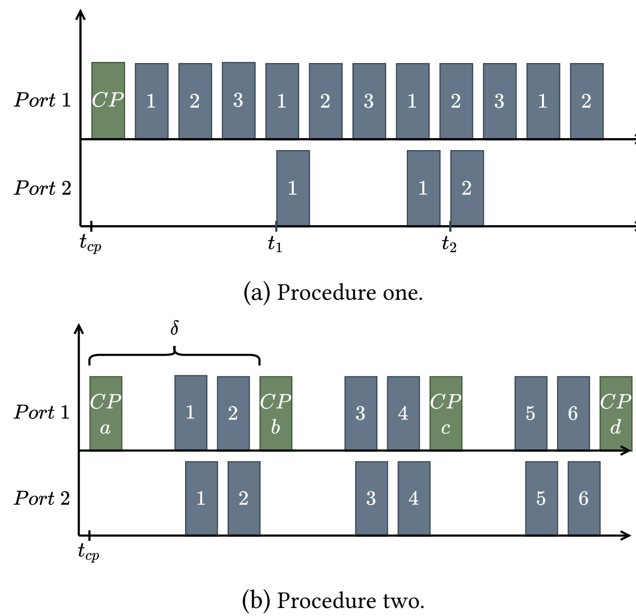
Figure 5.9: Burst metering accuracy.

which assumes that each forwarding device can be configured to have multiple priority queues (with a statically allocated amount of buffer space) per output port. Also, the considered framework relies on the *strict priority queuing* feature of the devices. Therefore, we evaluate whether the EdgeCore Wedge supports these features and if it can allocate a static (or predictable) amount of buffer space to each priority queue.

The traffic manager entity employed in Intel Tofino™ supports many queuing and buffer management strategies. For instance, it supports priority queuing, weighted fair queuing, static and dynamic buffer allocation. All these feature are fully supported by the APIs, provided as a part of Intel P4 Studio™ SDE. For example, in the static case, it is possible to allocate an arbitrary amount of buffer space to each priority queue (i.e., number of buffer cells). Hence, the traffic manager employed in Intel Tofino™ can definitely satisfy the requirements of DNC-based frameworks. Additionally, in the deployed network (see Sec. 5.4.1) we use Pica8 P3297 devices with lower amount of buffer space compared to the novel forwarding devices (the results are presented in a recent study [Van+19a]). Thus, for simplicity reasons, we do not measure the maximal amount of buffer space offered by Intel Tofino™. In fact, we simply assume that it offers at least the same amount of buffer space per priority queue as Pica8 P3297.

5.3.5 Control Plane Predictability

The proposed scheduling algorithm assumes that the edge switches can insert or reconfigure the rules predictably (i.e., bounded update time) from the DP (i.e., in-band CP). Thus, by conducting extensive measurements, we investigate if the presented P4 program and a local Python app (running on EdgeCore Wedge) achieve such performance in various scenarios.



(a) Procedure one.

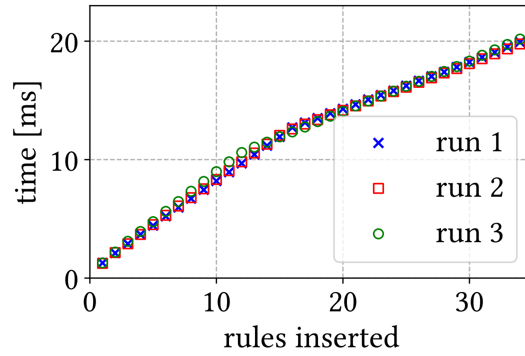
(b) Procedure two.

Figure 5.10: Different CP traffic generation procedures. In this scenario, the CP packet (green packet) contains a batch of 3 rule insertions (i.e., 1, 2, 3). (b) Example of the second CP traffic generation procedure. In this scenario, each CP packet contains a batch of 2 rule insertions (e.g., batch a contains rules 1 and 2). Ports 1 and 2 of the measurement card correspond to the QSFP Ports 1(a) and 3(a) of DUT.

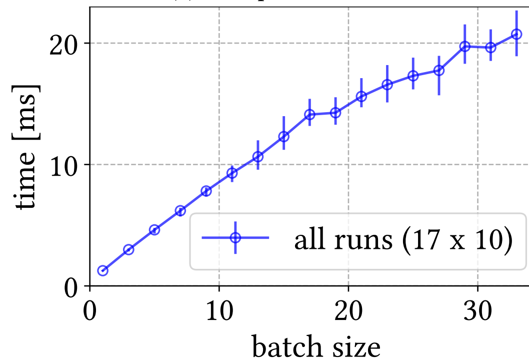
In general, Intel P4 Studio™ SDE offers three Python interfaces to insert or reconfigure the rules. Additionally, it provides C and C++ based APIs which can be used for the same purpose. All these interfaces/APIs provide different performance characteristics, and in this part, we do not aim to discover which one is the fastest. That is, we only confirm that one of them (i.e., the one used in our implementation/setup, see Sec. 5.3.1) achieves the required performance.

Setup and Procedure. We use the same measurement setup as presented in Fig. 5.6 with a different traffic generation procedure. At the beginning of each measurement run, we pre-populate the tagging and forwarding table with between 0 and 15000 rules. Afterward, the traffic generator sends an in-band CP packet (with the size of 1500B) containing a batch of insertion/reconfiguration rules (up to 34 rules can be encoded in the payload, see Sec. 5.3.1) to the DUT (see green packet on Fig. 5.10a). Each rule in the batch contains an action that instructs DUT to tag (with a variable number of tags), meter, and forward the matching DP traffic from QSFP Port 1(a) to QSFP Port 3(a). After the CP packet is sent, to check if the rule is successfully processed, the traffic generator sends DP packets at line rate which are sequentially matching the previously sent rules (depicted with dark blue color in Fig. 5.10a). The time to add or reconfigure a rule corresponds to the time difference between the reception of the in-band CP packet at DUT and the first matching DP packet forwarded on QSFP Port 3(a) of DUT. Based on Fig. 5.10a, the time to add rule n can be calculated as $t_n - t_{cp}$. If we use 1520B packets (including preamble and inter-packet gap) at 10G, the maximal measurement inaccuracy is $34 \times 1520 \times 8/10^{10} \approx 41\mu s$ (max. number of rules per control packet is 34). This value is significantly lower than the measured rule insertion and reconfiguration times.

Results. Fig. 5.11a shows insertion times of each rule for 3 different runs with the same configuration parameters. In total, it takes around 20ms for EdgeCore Wedge and the custom Python



(a) Example of 3 Runs

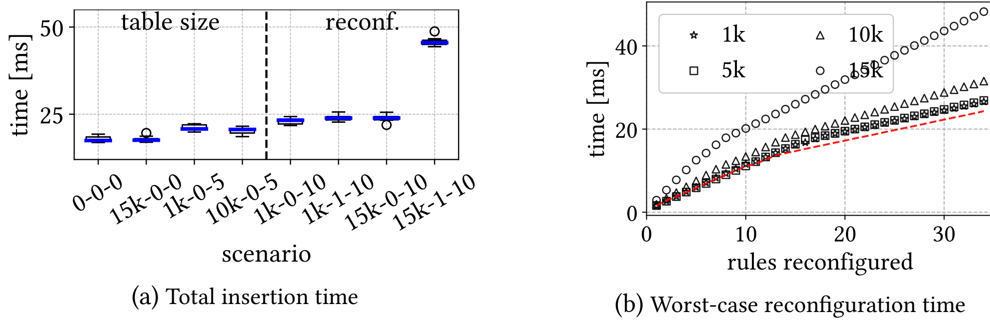


(b) Total Insertion Time

Figure 5.11: Control plane predictability.

application to process and insert 34 rules. The rules are inserted in a sequential order (i.e., n^{th} rule is always inserted just after the $(n\text{-}1)^{th}$ one). Additionally, inserting first 10-15 rules is slower (i.e., ≈ 1ms) compared to the later ones (e.g., ≈ 0.5ms). It can be seen that the runs produce almost identical results, meaning that inserting a batch of 34 rules is very predictable. Fig. 5.11b presents the results when the total number of rules in a CP packet is varied from 1 to 33 with increments of 2 (with 10 repetitions). The total time needed to insert all the rules belonging to the same batch is shown. The observed measurement results exhibit low variability and high predictability, similar to the previous scenario. For example, inserting a batch of 31 rules always takes around 20ms. Additionally, the presented results are almost identical as in Fig. 5.11a. For example, inserting the first 11 rules of a batch with 34 rules (see Fig. 5.11a) takes the same amount of time as inserting a batch of exactly 11 rules (see Fig. 5.11b), i.e., it takes approximately 10ms for both cases. It indicates that each rule in a batch is processed independently and that it is possible to generalize the results based on the biggest batch size (i.e., 34 rules is the maximum).

Fig. 5.12a illustrates the total time needed to add or reconfigure 34 rules in different configuration scenarios. We perform 10 measurement runs for each scenario and show the box plots. To denote each configuration, the following notation is used: *number of pre-populated rules-rule type-number of tags*. For example, in the case of label 1k-0-5, 1k indicates that there are 1000 pre-populated rules in the table, 0 means that rule type is an insertion, while 5 means that each rule adds 5 VLAN tags to matching packets. Firstly, the insertion time is propositional with the number of tags in the CP rules. For example, inserting 34 rules with 10 tags takes around 24ms, while with 5 it takes around 22ms (see



(a) Total insertion time

(b) Worst-case reconfiguration time

Figure 5.12: (a) Total time needed to insert or reconfigure a batch of 34 rules in different configuration scenario. (b) Worst-case reconfiguration time of each rule (in a batch) for 4 measurement scenarios with different amount of pre-populated rules (with 10 repetitions).

scenarios 1k-0-5 and 1k-0-10). Secondly, the number of existing rules in the table has a non-negligible impact on the total reconfiguration time. For instance, reconfiguring 34 rules in a table with 1000 existing rules takes around 25ms (see label 1k-1-10), while reconfiguring the same amount of rules in a table with 15000 rules takes almost 50ms (see label 15k-1-10). Therefore, the worst-case processing time of one batch of rules can be bounded based on the results observed when the number of existing rules in the table and the required tags are high. Fig. 5.12b shows the worst-case reconfiguration times of each rule for 4 scenarios with different amount of pre-populated rules (repeated 10 times). Similar to the results presented for insertion, the rules are reconfigured sequentially and exhibit low variance and high predictability. Later on, in Sec. 5.2, we use the data presented in this figure to estimate the worst-case processing time of a CP packet.

The previous results indicate that regardless of the number of existing rules in the table, it is possible to add or reconfigure a batch of rules delivered in one CP packet predictably.

Additionally, an edge switch should be able to insert multiple CP packets at runtime in a predictable manner. To show that it is possible, we create another measurement procedure as depicted in Fig. 5.10b. We send CP packets from the traffic generator to the edge switch uniformly spaced until we reach 15000 rules in the forwarding table. Every two consecutive packets are separated by the time interval of δ. To ensure the previous packet is completely processed before the arrival of the next one, we set the δ to be 10% higher than the observed worst-case time in Fig. 5.12b. We repeat this measurement 10 times for different batch sizes per packet (1, 10, and 30). Our observation confirms that not even a single rule is dropped or lost; in fact, all of them are processed successfully within the expected time based on Fig. 5.12b. We note that the processing time of these packets is decreased with time, similar to the effect presented in Fig. 5.11a.

5.4 Performance Evaluation

5.4.1 System Deployment

We implement `NAGA` in a real network and evaluate its performance, especially end-to-end DP and in-band CP update timing and consistency guarantees. To do so, we consider the topology presented in Fig. 5.13, taken from [Hon+13], with five logical switches, and six hosts. There are two transit (S_2 and S_5) and three edge switches (S_1, S_3 and S_4) in the network. We use SDN-enabled Pica P-3297 as

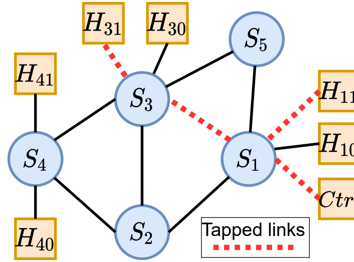


Figure 5.13: The logical test-bed network.

Flow Type	Rate[Mbps]	Burst[kb]	Delay[ms]
Class 1 (Low Delay)	U(8,40)	80	U(5,10)
Class 2 (High Rate/Burst)	U(40,120)	U(80,120)	U(10,100)
Class 3 (High Rate/Burst)	U(80,160)	U(120,160)	U(10,100)
Class 4 (Low Rate, High Burst)	U(5,8)	U(80,120)	U(5,50)
Class 5 (Low Rate, High Burst)	U(2,4)	U(80,120)	U(5,50)

Table 5.1: Considered flow types and their characteristics.

transit and EdgeCore Wedge as the edge switches. The first EdgeCore Wedge device corresponds to S_1, while the second one is split into two logical switches to emulate S_3 and S_4. To generate the DP traffic, each end-host uses a DPDK-based MoonGen traffic generator [Emm+15] with Intel x710-bm2 NIC. Moreover, the controller running the *admission control* and the proposed *scheduling algorithm* is connected to edge switch S_1. To measure the packet latency, we use a 10G *Endace* measurement card, and we set the speed of all the links in the network to 10G.

Traffic Classes & Flow Requests. We consider multiple flow types with different rate, burst, and end-to-end delay requirements (see Tab. 5.1). The considered flows types are based on the QoS requirements commonly found in different networking scenarios, e.g., industrial scenarios [C519]; [Cas21], remote network control [SK18], and even data-center networks [Rou05]; [WMZ19]; [Van+20]. Based on the considered flow types, *end-to-end* flow requests are generated on the NBI of the controller. Each flow request is generated between two randomly selected hosts with equal probabilities. The flow type, the values of rate, burst, and end-to-end delay parameters are generated randomly with a uniform distribution. Flow requests are generated sequentially online and are embedded into the network after being processed by the admission control and scheduling algorithm. The flow request generation procedure is continued until the admission control algorithm rejects a flow request.

Switch Configurations. The admission control algorithm relies on precise switch performance models (e.g., processing time, buffer space) of all the forwarding devices in the network. For EdgeCore Wedge devices, we rely on the results presented in Sec. 5.3, while for Pica P-3297, we take the data from [Van+19a]. We configure each device to have 4 priority queues per port. Even though EdgeCore Wedge has a larger buffer space (see Sec. 5.3.4), for the sake of simplicity, we model them in the same manner as Pica P-3297 devices. Since edge switch S_1 uses effectively 5 ports with 4 queues (controller does not receive packets), we configure each queue with 155kB of buffer space. S_3 and S_4 switches

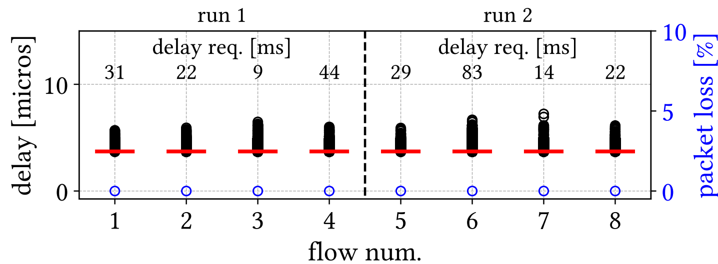


Figure 5.14: The measured end-to-end packet delay in the system.

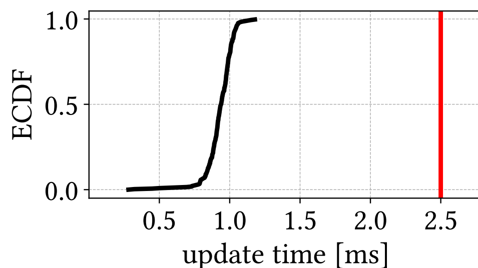


Figure 5.15: The measured update time (black line) vs. deadline (red line).

are realized by one EdgeCore Wedge device; hence, we configure and assign 77.5kB of buffer space per priority queue per port. For the transit switches S_2 and S_5, we consider the same values as S_1.

Scheduling Algorithm Input Parameters. We deploy the scheduling algorithm (alongside the admission control) on a controller as demonstrated in Fig. 5.13. The scheduling algorithm needs three input parameters which are network dependent: $\delta(S_x, S_y)$, $d_c(S_x)$, and $r(S_x)$.

These parameters depend on the following terms (see Sec. 5.2.1):
(1) Propagation delay: The deployed network is local, and the maximal cable length is 10 meters; hence, we assume this parameter is negligible.
(2) Maximal insertion and reconfiguration time of an edge switch, (i.e., $r(S_x)$). Since the deployed network is small, it cannot be filled with many flows (expected max $\sim$ 1000 flows). Thus, according to Fig. 5.12b, the worst-case insertion or reconfiguration time is around 1.96ms (table size is 1000, and batch size is 1).
(3) Total processing & sending time: The values of these parameters are in *ns* or μs [Van+19a] (see Sec. 5.3). Thus, we assume that they are negligible.
(4) Total queuing time: The in-band CP packets use the highest priority queues, with a buffer capacity of 155kB. Thus, the maximal queuing time of CP packets per hop is $0.1\mu s$.

Therefore, $\delta(S_x, S_y) = 1.96 + num_hops \times 0.1$ms; $\forall x, y \in \{1, .., k\}$. Nevertheless, since we have at most 2 hops in our deployment, we consider a static value for $\delta(S_x, S_y); \forall x, y \in \{1, .., k\}$ and set it to 2.5ms (including a 10% safety margin for the upper-bound calculations). Thus, in our deployment, we guarantee that every scheduled flow update is applied in the network within in maximum of 2.5ms.

Based on the calculated δ values, the rate r and burst b requirements of each in-band CP flow (with packet size 1500B) is calculated as: $r = 1500 \times 8/\delta = 4.8$Mbps and $b = 8 \times 1500 = 12$kb.

Deployment Scenarios. With the aforementioned parameters, we run two random scenarios in our network, in which 134, and 145 flow requests are embedded, respectively.

What Is Measured? Two measurement configurations are considered: DP and CP configuration. In the first one, the end-to-end latency of DP flows is measured to show that NAGA provides deterministic guarantees (i.e., no loss, bounded delay), even during network updates. To do so, we use optical taps to duplicate the traffic on the directional links between $H_{11} \longrightarrow S_1$ and $S_3 \longrightarrow H_{31}$. The duplicated traffic is forwarded to the measurement card. This setup allows us to observe the latency of each packet belonging to the flows between H_{11} and H_{31}.

In the second configuration, the goal is to show that the *scheduling algorithm* guarantees the predictability of rule updates. We show that after sending a flow update command from the controller (sending time is determined by the scheduler), the network is updated within a certain amount of time (i.e., in a maximum of 2.5ms as explained above). We use again optical taps on the links between the controller and the edge switches (i.e., $Ctrl \rightarrow S_1$) and link $S_1 - S_3$. We observe the time needed to update the network (or edge switches S_1 and S_3).

DP Results. Fig. 5.14 illustrates the observed *end-to-end* delay and packet loss of 8 randomly selected flows between hosts H_{11} and H_{31} for the three deployment scenarios. The end-to-end delay requirement of each flow is shown in the upper part of the figure. The results indicate there is no packet loss during the measurements and all the packets are delivered within the required delay bound. In most cases, the packets experienced a low queuing; thus, the box plots are very compact and look constant. In such cases, the only source of delay is the switch processing time and priority queuing overhead. The outliers correspond to the packets which experienced queuing. However, the buffer space available to the priority queues is never exceeded; thus, NAGA does not lose packets.

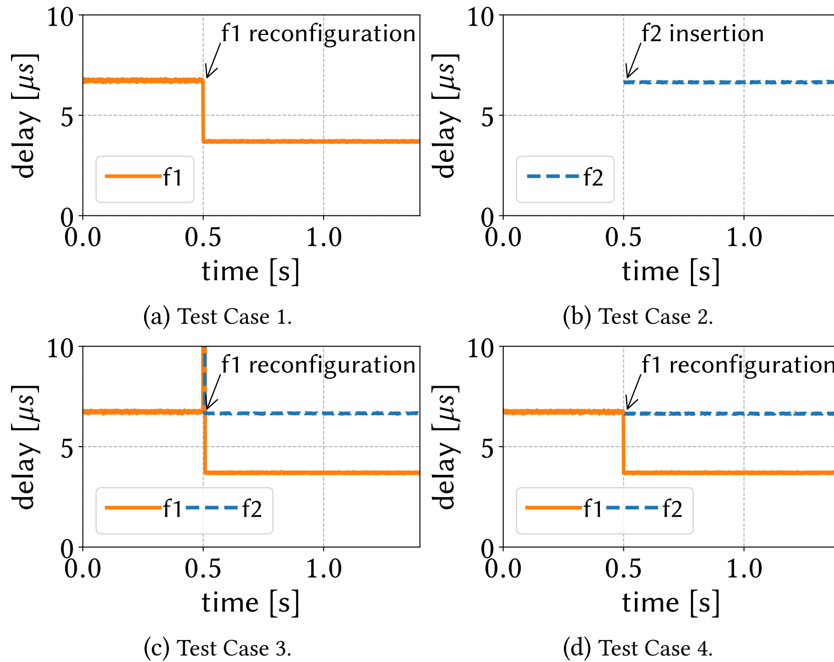
CP Results. Fig. 5.15 depicts the time between sending a scheduled rule update (on link $Ctrl. \longrightarrow S_1$) and the time when the update is successfully applied. Each scenario is repeated 5 times. NAGA ensures that the CP queue on each edge switch always have at most 1 CP message. Hence, the update times are very small (i.e., always less than 1.25μs), and they match the values observed in Sec. 5.3.5. However, compared to Sec. 5.3.5, we observe slightly higher values due to queuing, which is caused by the other high-priority traffic. Additionally, when the table size is small (e.g., $\leq$ 1000 entries), reconfiguring a flow is faster than adding; thus, the lower update values in Fig. 5.15 indicate flow reconfigurations. Finally, we never observe a dropped CP message or non-inserted/reconfigured flow.

5.4.2 Verifying Network Update Consistency

This part verifies the network update consistency of NAGA deployed in our testbed. The experiment uses the same network topology as in Fig. 5.13 with 10G links. We consider two flows: $f1 = (H_{30}, H_{10}, 8\text{Gbps}, 200\text{kb}, 10\text{ms})$ and $f2 = (H_{41}, H_{11}, 5\text{Gbps}, 150\text{kb}, 5\text{ms})$. Flows $f1$ and $f2$ are initially routed as $H_{30} \rightarrow S_3 \rightarrow S_2 \rightarrow S_1 \rightarrow H_{10}$ and $H_{41} \rightarrow S_4 \rightarrow S_2 \rightarrow S_1 \rightarrow H_{11}$, respectively. We measure the latency of $f1$ and $f2$ in four different test cases (see Fig. 5.16):

Test Case 1: In an empty network, at $t = 0$s, $f1$ is embedded. Then, at $t = 0.5$s, it is rerouted (by reconfiguring S_3) to a new path $H_{30} \rightarrow S_3 \rightarrow S_1 \rightarrow H_{10}$ with a lower delay. According to Fig. 5.16a, during the reconfiguration phase (around $t = 0.5$s), there is no increase or spike in the end-to-end delay. Moreover, we never observed a packet loss.

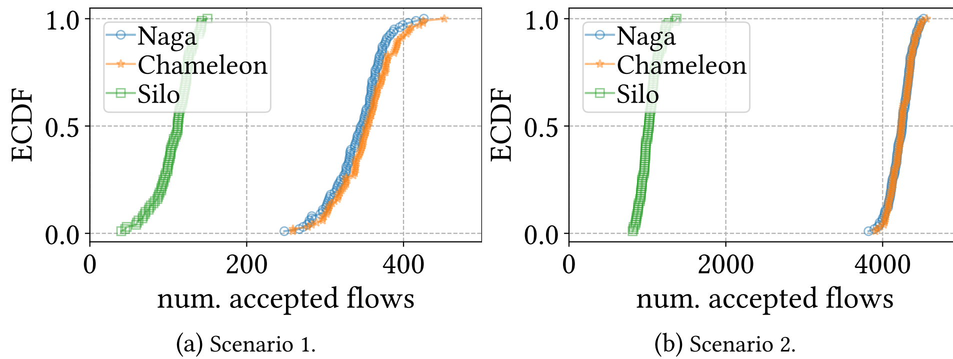
Test Case 2: Again, having an empty network, $f2$ is embedded in it at $t = 0.5$s (see Fig. 5.16b). After



(a) Test Case 1. (b) Test Case 2.

(c) Test Case 3. (d) Test Case 4.

Figure 5.16: Verifying the consistency of the flow updates with `NAGA`.



(a) Scenario 1. (b) Scenario 2.

Figure 5.17: The cost of in-band CP.

the embedding, it is evident that $f2$ has a stable delay, with no packet loss.

Test Case 3: Having $f1$ in the network, we add $f2$ and reroute $f1$ at $t = 0.5$s without considering the consistency (see Fig. 5.16c). Since the rules are updated inconsistently, the two flows overload the capacity of $S_2 - S_4$ link for a short period. Thus, the end-to-end delay measurements show that both flows experience a spike in the measured delay at around $t = 0.5$s. Further, at this time, we observed packet loss for both flows.

Test Case 4: Similar to test case 3, $f1$ is already embedded in the network. At around $t = 0.5$s, we add $f2$ and reroute $f1$ in a consistent way using `NAGA` (see Fig. 5.16d). Before adding $f2$, `NAGA`'s scheduler ensures that $f1$ is successfully rerouted (reconfiguring S_3). After that, $f2$ is added to the network by inserting a rule in S_4. The measurements show that `NAGA` can perform these flow updates without an increase in the delay of $f1$ during the update phase.

5.4.3 Cost of In-Band Control Plane

In this section, we evaluate the achieved number of accepted flows of NAGA, and compare it with two *SotA* systems that provide DP guarantees and use an out-of-band CP: *Chameleon* [Van+20] and *Silo* [Jan+15]. To do so, we use a network simulation tool that is available online [Van20], and it provides implementation of *Chameleon* and *Silo*. We further extend it by implementing the in-band CP bootstrapping logic of NAGA. In scenarios considering NAGA, we bootstrap the in-band CP flows at the start of the network and keep them active during the network lifetime, i.e., we keep the required resources reserved. Between each edge switch and the controller, we establish one in-band CP flow with the same flow requirements as in Sec. 5.4.1.

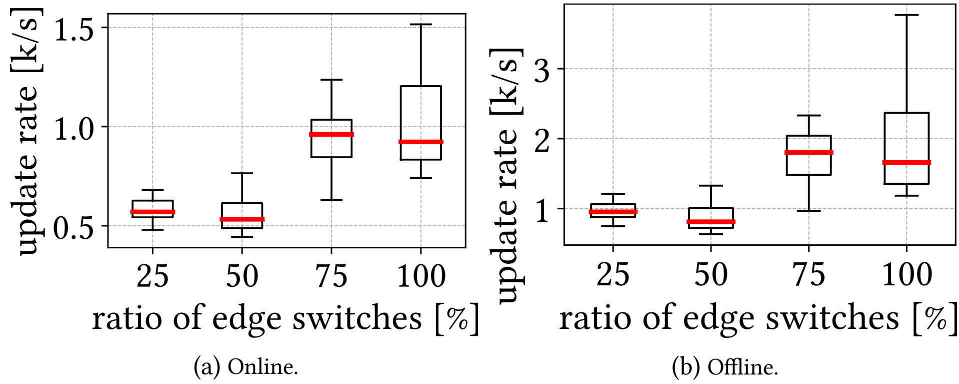
We perform the simulations on a large-scale network topology [Hoc+14], with 34 nodes and 42 links. We consider two simulation scenarios. In Scenario 1, we assume 10G links and switches with 155kB of buffer space per priority queue, while in the 2nd scenario, these values are considered as 100G and 3000kB, respectively. We use the same flow requirements (see Tab. 5.1) and generation procedure as presented in Sec. 5.4.1. However, to account for the propagation delay of the long-distance links, we added 30ms to each flow deadline (based on the network diameter length). To maximize the number of CP flows and generate the worst-case scenario, each node in the network is considered as an edge switch. Moreover, to maximize the network resource consumption of CP flows, we place the controller at the node with a minimum value of closeness centrality. The simulations are performed on a Windows 11 machine equipped with an Intel 12600k CPU, and 32GB of Random Access Memory (RAM). Each scenario is repeated 100 times with a randomized set of flow requests.

Fig. 5.17 shows the comparison of the number of accepted flows (i.e., network utilization) for the three considered systems. First, in comparison to *Silo*, *Chameleon* and NAGA accept on average around 3 and 4 times more flows depending on the scenario. The results indicate that the overhead of the bootstrapped in-band CP is overall very low, lying around 3% and 0.25%, for scenario 1 (Fig. 5.17a) and scenario 2 (Fig. 5.17b), respectively. The reason is, in NAGA, the in-band CP flows are low-rate and not bursty. Therefore, NAGA can provide consistent and predictable (in time) flow updates with negligible overhead.

5.4.4 Scalability Analysis

This section evaluates the NAGA's CP in terms of maximal flow update rate and update time guarantees. The investigated network is the same as the previous subsection, but with 100G links. The buffer space of the switches is set to 3000kB per priority queue. Again, the controller is placed on the node with the minimum closeness centrality.

We vary the percentage of the edge switches from 25% to 100% in the network. That is the number of hosts (hosts are connected only to edge switches), the number of flows in the network, and consequently, the number of flow updates. In this way, we can put pressure on the flow update scheduling task. Two scenarios are considered: *(1) online* and *(2) offline*. In the *online* case, the network updates arrive sequentially over time to the scheduler, similar to *SotA* approaches [NCC17]; [Zho+21]. In this case, each set of updates arrives after the scheduler processes and sends the previous one. In the case of *offline*, all network updates (i.e., the output of the admission control algorithm) are available at the beginning to the scheduler. Flow requests are generated in the same manner as in Sec. 5.4.3.



(a) Online.

(b) Offline.

Figure 5.18: Comparison of network update rate.

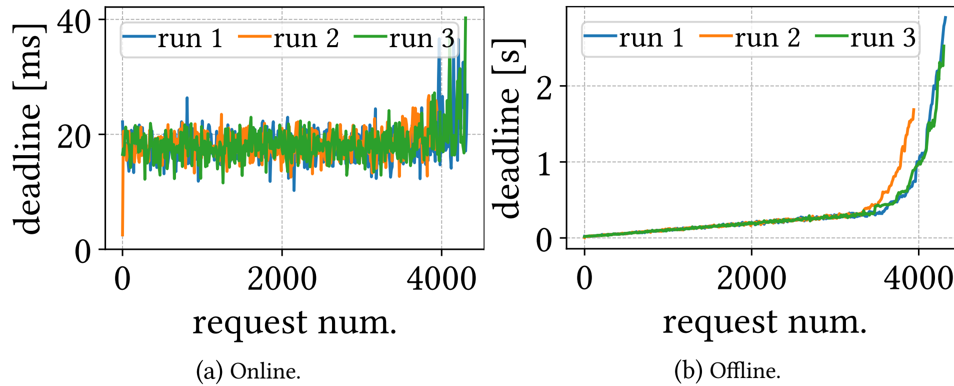
The results for the *online* case (Fig. 5.18a) indicate that the network update rate increases with the percentage of edge switches (25% to 100%), reaching values of around 1.5k per second. In the *offline* case (Fig. 5.18b), the results follow the same trend. That is, the maximal update rate per scenario increased to 3.5k. The achieved rates show that `NAGA` can deal with a wide range of network updates, depending on the distribution of the flow requests, and the admission control complexity.

Fig. 5.19a-5.19b show the average guaranteed deadline per 10 sequential flow requests, for *offline* and *online* scenarios, respectively. Only 3 selected runs per scenario are shown (100% of network nodes are edge switches), the other runs follow the same trend. The deadline values are calculated from the arrival time of the network update set (to the scheduler) until the time guarantee of the last flow update in the set.

In the *online* case, the average deadlines stay mostly around 20ms (Fig. 5.19a). Since network update sets arrive sequentially, this value is mostly influenced by propagation delay and queuing time in the network, as explained in Sec. 5.2. Towards the end of each run, the average deadlines start increasing since the network becomes highly utilized, and the dependent reconfigurations become necessary.

In the *offline* case (Fig. 5.19b), the derived deadlines increases with the number of processed flow update sets. This deadline is mostly influenced by the waiting time of the updates since the edge switches are overloaded with too many updates. In the beginning, the network is empty, thus, most of the network update sets do not contain dependent reconfigurations. Hence, the update rate is high and the deadlines increase linearly. However, as the network fills up, reconfigurations become necessary, which significantly slows down the network update rate. Thus, the deadlines start to increase significantly.

Remarks. Since `NAGA` provides deterministic network update timing guarantees, most of the network update solutions do not solve the same problem. Therefore, comparing them directly is not fair. For reference, many *SotA* consistent network update solutions can take between 500-1300ms to update a single flow in the network [Zho+21]. Therefore, such solutions could be potentially improved by considering the concepts presented and deployed in `NAGA`.



(a) Online. (b) Offline.

Figure 5.19: Comparison of network update deadlines.

5.5 Related Work

Deterministic Guarantees: The *SotA* contains systems that provide deterministic guarantees in the DP: Qjump [Gro+15], Silo [Jan+15], Loko [Van+19b], Chameleon [Van+20], and [DKW18]. Qjump [Gro+15] provides deterministic guarantees in the DP only. For instance, the guarantees are provided, by ensuring that at any given time, at most one packet per flow is transmitted in the network [Gro+15], or by utilizing network calculus [Jan+15]; [Van+19b]; [Van+20]. Additionally, in order to achieve higher utilization, Chameleon [Van+20] extends network calculus based solutions [Jan+15]; [Van+19b] by using a so called queue-level topology and reconfigurations.

However, most of the works [Van+20]; [Jan+15]; [Gro+15]; [Van+19b] assume that the end-hosts are controllable by the network provider. Moreover, they assume that the end hosts generate traffic according to agreed policies, while some [Gro+15] cannot even handle packet bursts. Further, [Gro+15]; [Jan+15]; [Van+19b] do not employ network reconfigurations, which can lead to a low network utilization. Finally, all of them [Gro+15]; [Jan+15]; [Van+20]; [Van+19b]; [DKW18], consider an out-of-band CP and network update consistency. In addition, some proprietary solutions can provide deterministic guarantees such as Profibus and Ethernet extensions [Dec05]; [GJF12]; [VHV12], and Time-Sensitive Networking (TSN) [Nas+18]. However, they are generally use-case-specific and not reconfigurable. In addition, they mostly use proprietary devices and drivers, while NAGA can be deployed on COTS equipment.

Consistent Network Updates: Another set of *SotA* [Rei+12]; [MW13]; [Zho+21]; [NCC17]; [Jin+14]; [GC12] belongs to the topic of consistent updates for programmable networks (see [FSV18] survey paper for a comprehensive overview on this topic). Usually, they focus on providing procedures to ensure the network keeps forwarding the packets towards their destinations, throughout the update process [FSV18]. To do this, the network provider must ensure that the update packets are ordered, delivered, and processed successfully by the forwarding device. Determining the order of the updates can be done with analytical models and algorithms, such as using dependency graphs [Jin+14]. Regarding the delivery of the packet, some works rely on acknowledge-based methods [MSM16]; [KPK14b], TCP retransmissions [DKW18], and network verification [Shu+21] to ensure the network update is received and processed by the device. However, these solutions suffer from two serious drawbacks; First, they cannot guarantee timely delivery of CP packets from the controller to the forwarding device; thus, cannot be used in deterministic networks. Second, they

are generally slow [Zho+21], while NAGA can perform significant number of network updates per second (see Fig. 5.18b).

5.6 Conclusion

This chapter takes an integral step toward predictable network operation: it investigates and contributes to a system with deterministic in-band CP operation of programmable networks. We developed NAGA, a system that provides both predictable DP and CP and achieves consistent and timely guaranteed network updates using a novel scheduling algorithm. We demonstrate the practicality of NAGA using a real-world prototype. Our evaluations confirmed our claims about determinism of NAGA regarding the strict QoS requirements in dynamic environments. Moreover, extensive simulations indicated that while NAGA has low in-band CP overhead, it is highly scalable, i.e., more than 3.5k flow updates per second in a realistic scenario. Finally, we showed that the in-band CP of NAGA introduces a low overhead in the system compared to *SotA* works.

Instead of relying on end-hosts (as other *SotA* works), NAGA uses only widely-available programmable network capabilities such as priority queuing and label-based forwarding. In addition, it also uses softwarized and centralized network logic. Therefore, NAGA can be deployed in virtualized programmable networks with minimal effort.

Chapter 6

Conclusion and Future Work

The rapid development in the technological sphere of Industry 4.0, Internet of Things (IoT), Cyber-physical System (CPS), and cloud computing introduced many novel applications such as automated manufacturing, autonomous driving, and remote surgery. Most of these applications have very high Quality of Service (QoS) requirements which legacy *best-effort* communication networks are not capable of providing. For example, in the case of remote surgery, the network between a robot operating on a human and a remote surgeon should guarantee low end-to-end latency, bounded packet loss, and high reliability. Similar examples occur in CPS systems where the control data exchanged between controllers and actuators should be delivered with certain QoS guarantees. Failing to provide these guarantees might have catastrophic consequences. For example, in CPS systems, if the end-to-end latency between a controller and a balancing robot reaches exceeds certain values, the robot might become out of sync, and the control could be lost. This can be especially dangerous if people are working besides it. Therefore, novel technologies and concepts are needed which can provide such high QoS requirements.

On the other hand, the recent cloud computing revolution demonstrated that the benefits of sharing physical resources are enormous and outweigh the drawbacks. For example, the cost reduction and ability to dynamically scale applications are much more valuable compared to the additional complexity of virtualizing the hardware and managing it. Therefore, to attain the same benefits, network operators could virtualize their programmable networks. However, doing both at the same time, providing high and guaranteed QoS in virtualized programmable networks is still challenging. In the literature, both of these problems are already studied separately, i.e., providing high QoS and virtualizing programmable networks. For instance, Deterministic Network Calculus (DNC)-based systems [Van+20]; [Jan+15] such as *Chameleon* are capable of providing deterministic Data Plane (DP) guarantees in Data Center (DC) networks. However, since they rely on end-host control (common in DCs), they are not usable in virtualized networks where hosts do not belong to the network operator. While virtualization solutions [Al-+14]; [She+09] on the contrary do not provide high QoS guarantees such as deterministic guarantees, which might be needed for CPS and similar systems. This thesis aims to overcome these challenges by developing concepts that enable QoS-aware virtualization of programmable networks. To be precise, firstly, one of the main objectives is to design and develop novel methodologies for ensuring that the Control Plane (CP) performance of virtual-

ized networks meets certain QoS requirements. Secondly, in this thesis, also general networking concepts (e.g., traffic policing) related to network virtualization are studied and enhanced further.

The contributions with the corresponding derived conclusion are presented in next section (i.e., Section 6.1). While the future work is presented in Section 6.2.

6.1 Summary

To begin with, the initially presented concepts in this thesis resolved a couple of management and orchestration issues that are coming from the CP of virtualized programmable networks. In Chapter 3, firstly it was demonstrated through measurements that different Network Hypervisor (NH) functions can have a significant impact on the Central Processing Unit (CPU) utilization of a NH. That is, enabling topology abstraction can reduce the CPU utilization of a NH up to 4× in certain scenarios. This observation motivated us to study the impact of topology abstraction on the optimality of Virtual Network Embedding (VNE) algorithms. To do so, an algorithm based on Integer Linear Programming (ILP) model that includes CP aspects was developed and presented. The evaluation showed that including NH functions (i.e., topology abstraction) is crucial for designing performant VNE algorithms. That is, the model including the topology abstraction in certain cases outperformed the baseline algorithm for around 50%, and can be used to optimize the resource usage in virtualized programmable networks.

Secondly, Chapter 3 presented a QoS-aware measurement-based methodology for provisioning the resources of a NH. To design it, initially, a comprehensive measurement study of one of the most well-known NHs, i.e., *FlowVisor*, was performed. The main objective of this study was to understand what parameters influence the CPU utilization of a *FlowVisor* and to design an accurate CPU estimation model. Even though many parameters (e.g., virtual network size) have an impact, it was possible to design a very accurate model. The developed model exhibited a mean average relative error of 4%. Subsequently, by utilizing the presented model, an algorithm for provisioning the resources of a NH was presented. The algorithm aims to allocate a minimal amount of CPU resources to a NH while ensuring that there is no performance degradation. By deploying it on a real test bed, it was possible to demonstrate its effectiveness. To be precise, even with randomly generated network topologies, it was possible to provision the resources of a NH with a negligible impact on its processing performance. The presented methodology enables network operators to utilize their resources more optimally while still ensuring that their NH operates with certain QoS. Additionally, even though this methodology focused on provisioning NHs, it can be utilized in other networking areas. That is, it can be applied to other softwarized networking functions (such as Network Function Virtualization (NFV)) to achieve the same benefits.

Afterward, Chapter 4 studied how to measure and model the performance of in-network and end-host based traffic policing and it presented two measurement studies. Traffic policing plays an important role in virtualized programmable networks as it enables the realization of DP isolation. Initially, a measurement methodology that can be used to extract the traffic policing parameters (i.e., rate and burst size) from the measurement data was presented. It is based on DNC mathematical framework, and the extracted parameter can be used as input parameters when designing (deterministic) systems. The presented methodology was then used to evaluate the performance of both,

in-network (five Open Flow (OF)-enabled switches) and end-host (an Data Plane Development Kit (DPDK) application) traffic policing. The traffic policing accuracy of the considered switches was very high (e.g., the relative error was under 0.1%) in cases when the configured rate and burst size was high (e.g., the rate was higher than $10Mbps$). However, in scenarios with low policing rate and burst, some switches exhibited very high inaccuracy (relative error exceeded values of 60%). In contrast, the performance of end-host traffic policing was constant and the relative error was usually below 1%. In the end, a simulation study was presented to demonstrate that the impact of traffic policing inaccuracies on the performance of one *state-of-the-art* (*SotA*) deterministic system is significant (e.g., the number of accepted flows was halved). The main takeaway is that the inaccuracy exhibited by in-network hardware devices is not negligible, and has to be taken into consideration when designing DP (isolation) concepts capable of providing QoS guarantees.

The final contribution of this thesis is `NAGA` system which provides deterministic data and CP guarantees. Firstly, to realize the DP guarantees, `NAGA` uses the DNC-based admission control algorithm and it offloads certain functionalities (i.e., traffic policing and tagging) to edge switches. The contributions presented in Chapter 4 were crucial for enabling the offloading. That is, by using the traffic policing measurement procedure, it was possible to measure the traffic policing performance of hardware devices, and to model it in the control logic of `NAGA`. Therefore, in contrast to the other approaches in the literature, `NAGA` does not rely on the control of end-hosts, thus, it can be used in virtualized programmable networks for either isolating the DP traffic or providing QoS guarantees to the tenants. Secondly, to realize the deterministic guarantees in the CP, `NAGA` relies on an in-band CP network and a carefully designed network scheduling algorithm. By utilizing the in-band CP over a deterministic DP network, it becomes possible to guarantee that the CP messages will be delivered reliably and within a certain time (i.e., no packet loss and bounded delay). Moreover, the proposed scheduling algorithm aims to avoid overloading the CP of networking devices and it ensures that the network is updated consistently. To verify the performance of `NAGA` system, we have deployed it on a real test-bed with Programming Protocol-independent Packet Processors (P4)-enabled hardware devices. The measurements showed that `NAGA` system does indeed provide DP guarantees and consistent network updates without end-host control while supporting high scheduling rate (i.e., over 3k network updates per second).

Having such guarantees is particularly interesting for high-mobility use cases. For instance, in an autonomous driving scenario, the vehicles are always moving and changing their network access points connecting them to a cloud system. Therefore, in such scenarios, network operators could significantly benefit from utilizing consistent and timed network updates to ensure that connectivity is always available.

The QoS-aware resources provisioning methodology presented in Chapter 3 and the CP part of `NAGA` presented in Chapter 5 take an important step towards enabling the realization of *end-to-end* CP guarantees in virtualized programmable networks. That is, the provisioning methodology ensures that the processing time of an NH always achieves certain QoS, even in dynamically changing scenarios. While the CP aspects of `NAGA` can be used for the realization of control network with update timing guarantees between an NH and networking switches.

6.2 Future Work

In the following, potentially interesting future research directions are presented and discussed.

Automated & smart measurements.

Certain concepts in this thesis rely on the values extracted from the presented comprehensive measurements. For instance, in Chapter 3.2, the resources of a NH are provisioned based on the developed estimation model that was generated from the measurement data. Similarly, the DNC theory used in Chapter 5 (see background in Sec. 2.3.2) relies on the *service curves* which are in fact performance models of networking switches (e.g., collected and presented in Chapter 5). However, in this thesis, the measurement data was collected only for certain scenarios while considering the fixed set of parameters. For example, in Chapter 3.2, to obtain the performance measurement data for two NHs the same physical machine was used to run them. Therefore, the derived estimation models are only valid for the considered physical machine. Changing it (e.g., using a different CPU) might influence the scaling dependencies of the model, thus, the model could require refitting with additional measurements. Likewise, the input values (i.e., maximal switch processing time) for DNC admission control algorithm in Chapter 5 are only valid for certain measured hardware switches.

Therefore, novel measurement tools are needed which should support automated *(1)* benchmarking and data collection, and *(2)* data analysis. First, novel tools should in an automatic way (if needed) install and configure a Device Under Test (DUT) (e.g., a NH or switch) and run the measurement test cases. Secondly, they should also incorporate smart data analysis. That is, the measurement data could be analyzed on the fly with either Machine Learning (ML) or Artificial Intelligence (AI) solutions, and the test cases can be adapted based on the learned observations. This could potentially save the measurement time and resources and enable faster extracting of influential parameters.

Traffic policing with programmable hardware.

The development of programmable DP hardware devices (i.e., based on P4 programming language) made it possible to realize novel custom traffic metering algorithms directly on hardware [He+21]; [Tha+21]. For example, in [He+21], the authors developed and deployed a token bucket algorithm directly on P4-based hardware switch which supports link speed up to $100G$. Firstly, this solution might offer better performance (e.g., higher traffic policing accuracy) compared to the *carrier-grade* OF-enabled hardware switches which exhibit extreme performance deviations in certain scenarios (as presented in Chapter 4). However, the evaluations of the already developed custom algorithms in the literature were mostly focused on average values, and not on the worst-case ones. Therefore, novel measurement studies based on the measurement methodology presented in Chapter 4 are needed. In addition to measuring the already existing ones, novel traffic policing algorithms could be developed focused on achieving the best worst-case performance.

Secondly, with programmable DP devices, it might become possible to develop traffic shaping solutions[1], which are typically not available on *carrier-grade* switches. However, since traffic shaping solutions typically require a lot of memory for packet queuing (or buffering), it is still unclear how to design traffic shaping solutions that can optimally utilize the available hardware resources (e.g., available memory).

[1] In traffic shaping, the packets are not dropped as in traffic policing, but rather delayed and sent later.

Scalability of deterministic systems.

One of the major drawbacks of systems that provide deterministic DP guarantees (e.g., NAGA or *Chameleon*) in Software-Defined Networking (SDN) networks is scalability. For example, in *Chameleon* [Van+20], the total time needed for an admission control algorithm (deployed in a centralized controller) time to determine if a certain flow request can be embedded without violating the guarantees of other already embedded flows take at least a couple of *milliseconds* and it sometimes exceeds 1 *second*. Therefore, if we assume that the average needed time is $10ms$, the *Chameleon* controller can only support adding around 100 flow requests per second. This value is insufficient for certain dynamically changing networks (e.g., DC networks with many micro flows). This issue can be resolved by utilizing horizontal scaling. That is, instead of one controller we could have multiple controllers which control the partitioned network. However, how to optimally partition a deterministic network, and enabled multiple distributed controllers to operate it while avoiding inconsistencies is an interesting research problem.

Providing stochastic and deterministic guarantees at the same time.

To provide deterministic guarantees, mathematical frameworks such as DNC [LT01] often base their calculations on the worst-case scenarios. This means that by default, DNC and DNC-based systems (and similar) are conservative and often sacrifice network utilization to provide the guarantees. However, many applications do not even require deterministic or even high QoS guarantees. For instance, since videos can be buffered on personal computers, streaming them from YouTube can be done over best-effort networks. Hence, providing deterministic guarantees to such applications might lead to a waste of resources. However, most of the systems including NAGA (presented in Chapter 5) that provide deterministic DP guarantees are not capable of providing different types of QoS guarantees (e.g., stochastic). Hence, new systems are needed which can support at the same time best-effort traffic in addition to the traffic with deterministic and stochastic requirements.

Bibliography

Publications by the Author

Journal Publications

[Đer+21] N. Đerić, A. Varasteh, A. Van Bemten, A. Blenk, and W. Kellerer. "Enabling SDN Hypervisor Provisioning through Accurate CPU Utilization Prediction." In: *IEEE Transactions on Network and Service Management* (2021).

Conference Publications

[Đer+] N. Đerić, A. Varasteh, A. Blenk, and W. Kellerer. "NAGA: A Deterministic Programmable Network with Update Timing Guarantees." Under Subimssion.

[Đer+17] N. Đerić, A. Blenk, A. Basta, and W. Kellerer. "Challenges and Solutions for a Flexible High-Performant SDN Hypervisor." In: *KuVS Fachgespräch" Network Softwarization"–From Research to Application*. 2017.

[Đer+18] N. Đerić, A. Varasteh, A. Basta, A. Blenk, and W. Kellerer. "SDN hypervisors: How much does topology abstraction matter?" In: *2018 14th International Conference on Network and Service Management (CNSM)*. IEEE. 2018, pp. 328–332.

[Đer+21] N. Đerić, A. Varasteh, A. Van Bemten, C. Mas-Machuca, and W. Kellerer. "Towards Understanding the Performance of Traffic Policing in Programmable Hardware Switches." In: *IEEE International Conference on Network Softwarization (NetSoft)*. 2021.

[He+18] M. He, A. Basta, A. Blenk, N. Deric, and W. Kellerer. "P4NFV: An NFV architecture with flexible data plane reconfiguration." In: *2018 14th International Conference on Network and Service Management (CNSM)*. IEEE. 2018, pp. 90–98.

[Hel+21] M. Helm et al. "Application of network calculus models on programmable device behavior." In: *2021 33th International Teletraffic Congress (ITC-33)*. IEEE. 2021, pp. 1–9.

[SĐK18] E. Sakic, N. Đerić, and W. Kellerer. "MORPH: An adaptive framework for efficient and Byzantine fault-tolerant SDN control plane." In: *IEEE Journal on Selected Areas in Communications* 36.10 (2018), pp. 2158–2174.

[Van+19a] A. Van Bemten, N. Đerić, A. Varasteh, A. Blenk, S. Schmid, and W. Kellerer. "Empirical Predictability Study of SDN Switches." In: *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. ACM/IEEE. 2019, pp. 1–13.

[Van+19b] A. Van Bemten, N. Đerić, J. Zerwas, A. Blenk, S. Schmid, and W. Kellerer. "Loko: Predictable latency in small networks." In: *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*. 2019, pp. 355–369.

[Van+20] A. Van Bemten, N. Đerić, A. Varasteh, S. Schmid, C. Mas Machuca, A. Blenk, and W. Kellerer. "Chameleon: Predictable Latency and High Utilization with Queue-Aware and Adaptive Source Routing." In: *Proceedings of the ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*. 2020, pp. 451–465.

[Var+19] A. Varasteh, S. Hofmann, N. Deric, M. He, D. Schupke, W. Kellerer, and C. M. Machuca. "Mobility-aware joint service placement and routing in space-air-ground integrated networks." In: *ICC 2019-2019 IEEE International Conference on Communications (ICC)*. IEEE. 2019, pp. 1–7.

Demos

[Đer+19] N. Đerić, A. Varasteh, A. Basta, A. Blenk, R. Pries, M. Jarschel, and W. Kellerer. "Coupling VNF orchestration and SDN virtual network reconfiguration." In: *2019 International Conference on Networked Systems (NetSys)*. IEEE. 2019, pp. 1–3.

General Publications

[Ada+15] D. Adami, L. Donatini, S. Giordano, and M. Pagano. "A network control application enabling software-defined quality of service." In: *2015 IEEE International Conference on Communications (ICC)*. IEEE. 2015, pp. 6074–6079.

[Al-+14] A. Al-Shabibi, M. De Leenheer, M. Gerola, A. Koshibe, G. Parulkar, E. Salvadori, and B. Snow. "OpenVirteX: Make your virtual SDNs programmable." In: *Proceedings of the third workshop on Hot topics in software defined networking*. 2014, pp. 25–30.

[BAP17] L. Barreto, A. Amaral, and T. Pereira. "Industry 4.0 implications in logistics: an overview." In: *Procedia manufacturing* 13 (2017), pp. 1245–1252.

[Bas+14] A. Basta, W. Kellerer, M. Hoffmann, H. J. Morper, and K. Hoffmann. "Applying NFV and SDN to LTE mobile core gateways, the functions placement problem." In: *Proceedings of the 4th workshop on All things cellular: operations, applications, & challenges*. 2014, pp. 33–38.

[Bas+17] A. Basta, A. Blenk, W. Kellerer, and S. Schmid. "Logically Isolated, Actually Unpredictable? Measuring Hypervisor Performance in Multi-Tenant SDNs." In: *arXiv preprint arXiv:1704.08958* (2017).

[Bau+18] S. Bauer, D. Raumer, P. Emmerich, and G. Carle. "Behind the scenes: what device benchmarks can tell us." In: *Proceedings of the Applied Networking Research Workshop*. 2018, pp. 58–65.

[BBK15] A. Blenk, A. Basta, and W. Kellerer. "HyperFlex: An SDN virtualization architecture with flexible hypervisor function allocation." In: *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*. IEEE. 2015, pp. 397–405.

[BCB10] N. F. Butt, M. Chowdhury, and R. Boutaba. "Topology-awareness and reoptimization mechanism for virtual network embedding." In: *International Conference on Research in Networking*. Springer. 2010, pp. 27–39.

[BEE16] K. Benzekki, A. El Fergougui, and A. Elbelrhiti Elalaoui. "Software-defined networking (SDN): a survey." In: *Security and communication networks* 9.18 (2016), pp. 5803–5833.

[Bem+20] V. Bemten et al. "Design, Implementation, and Evaluation of Mechanisms for Predictable Latency in Programmable Networks." PhD thesis. Technische Universität München, 2020.

[Ber+14] P. Berde et al. "ONOS: towards an open, distributed SDN OS." In: *Proceedings of the third workshop on Hot topics in software defined networking*. ACM. 2014, pp. 1–6.

[Bia+10] A. Bianco, R. Birke, L. Giraudo, and M. Palacin. "Openflow switching: Data plane performance." In: *2010 IEEE International Conference on Communications*. IEEE. 2010, pp. 1–5.

[Ble+15] A. Blenk, A. Basta, J. Zerwas, and W. Kellerer. "Pairing SDN with network virtualization: The network hypervisor placement problem." In: *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*. IEEE. 2015, pp. 198–204.

[Ble+16a] A. Blenk, A. Basta, M. Reisslein, and W. Kellerer. "Survey on network virtualization hypervisors for software defined networking." In: *IEEE Communications Surveys & Tutorials* 18.1 (2016), pp. 655–685.

[Ble+16b] A. Blenk, A. Basta, J. Zerwas, M. Reisslein, and W. Kellerer. "Control plane latency with SDN network hypervisors: The cost of virtualization." In: *IEEE Transactions on Network and Service Management* 13.3 (2016), pp. 366–380.

[Ble+16c] A. Blenk, P. Kalmbach, P. Van Der Smagt, and W. Kellerer. "Boost online virtual network embedding: Using neural networks for admission control." In: *Network and Service Management (CNSM), 2016 12th International Conference on*. IEEE. 2016, pp. 10–18.

[Ble+19] A. Blenk, A. Basta, W. Kellerer, and S. Schmid. "On the Impact of the Network Hypervisor on Virtual Network Performance." In: *2019 IFIP Networking Conference (IFIP Networking)*. May 2019, pp. 1–9. DOI: 10.23919/IFIPNetworking.2019.8816839.

[Bos+14] P. Bosshart et al. "P4: Programming protocol-independent packet processors." In: *ACM SIGCOMM Computer Communication Review* 44.3 (2014), pp. 87–95.

[BP02] A. Banchs and X. Perez. "Distributed weighted fair queuing in 802.11 wireless LAN." In: *2002 IEEE International Conference on Communications. Conference Proceedings. ICC 2002 (Cat. No. 02CH37333)*. Vol. 5. IEEE. 2002, pp. 3121–3127.

[BR13] Z. Bozakov and A. Rizk. “Taming SDN controllers in heterogeneous hardware environments.” In: *2013 Second European Workshop on Software Defined Networks*. IEEE. 2013, pp. 50–55.

[C519] 5. A. for Connected Industries and A. (5G-ACIA). *A 5G Traffic Model for Industrial Use Cases*. Tech. rep. 2019.

[Cas21] I. U. Cases. *Use Cases IEC/IEEE 60802 v1.3*. Dec. 2021. URL: http://www.ieee802.org/1/files/public/%20docs2018/60802-industrial-use-cases-0918-v13.pdf.

[Com17] R. S. F. Community. *Ryu SDN Framework*. https://osrg.github.io/ryu/. Accessed: 2020-07-21. 2017.

[Cor+12] R. D. Corin, M. Gerola, R. Riggio, F. De Pellegrini, and E. Salvadori. “Vertigo: Network virtualization and beyond.” In: *Software defined networking (EWSDN), 2012 European Workshop on*. IEEE. 2012, pp. 24–29.

[COS17] S. S. Craciunas, R. S. Oliver, and W. Steiner. “Formal scheduling constraints for time-sensitive networks.” In: *arXiv preprint arXiv:1712.02246* (2017).

[CRB09] N. M. K. Chowdhury, M. R. Rahman, and R. Boutaba. “Virtual network embedding with coordinated node and link mapping.” In: *INFOCOM 2009, IEEE*. IEEE. 2009, pp. 783–791.

[CRB12] M. Chowdhury, M. R. Rahman, and R. Boutaba. “Vineyard: Virtual network embedding algorithms with coordinated node and link mapping.” In: *IEEE/ACM Transactions on Networking (TON)* 20.1 (2012), pp. 206–219.

[Cru91a] R. L. Cruz. “A calculus for network delay. I. Network elements in isolation.” In: *IEEE Transactions on information theory* 37.1 (1991), pp. 114–131.

[Cru91b] R. L. Cruz. “A calculus for network delay. II. Network analysis.” In: *IEEE Transactions on information theory* 37.1 (1991), pp. 132–141.

[Dan+20] S. Dang, O. Amin, B. Shihada, and M.-S. Alouini. “What should 6G be?” In: *Nature Electronics* 3.1 (2020), pp. 20–29.

[DBK15] R. Durner, A. Blenk, and W. Kellerer. “Performance study of dynamic QoS management for OpenFlow-enabled SDN switches.” In: *2015 IEEE 23rd International Symposium on Quality of Service (IWQoS)*. IEEE. 2015, pp. 177–182.

[De +14] R. L. S. De Oliveira, C. M. Schweitzer, A. A. Shinoda, and L. R. Prete. “Using mininet for emulation and prototyping software-defined networks.” In: *2014 IEEE Colombian conference on communications and computing (COLCOM)*. Ieee. 2014, pp. 1–6.

[Dec05] J.-D. Decotignie. “Ethernet-based real-time and industrial communications.” In: *Proceedings of the IEEE* 93.6 (2005), pp. 1102–1117.

[Dij+76] E. W. Dijkstra, E. W. Dijkstra, E. W. Dijkstra, E.-U. Informaticien, and E. W. Dijkstra. *A discipline of programming*. Vol. 1. prentice-hall Englewood Cliffs, 1976.

[Dij59] E. W. Dijkstra. “A note on two problems in connexion with graphs.” In: *Numerische mathematik* 1.1 (1959), pp. 269–271.

[Dix+13] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. "Towards an elastic distributed SDN controller." In: *ACM SIGCOMM computer communication review* 43.4 (2013), pp. 7–12.

[Dix+14] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. R. Kompella. "ElastiCon; an elastic distributed SDN controller." In: *2014 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE. 2014, pp. 17–27.

[DKR13] D. Drutskoy, E. Keller, and J. Rexford. "Scalable network virtualization in software-defined networks." In: *IEEE Internet Computing* 17.2 (2013), pp. 20–27.

[DKW18] N. Dorsch, F. Kurtz, and C. Wietfeld. "Enabling hard service guarantees in software-defined smart grid infrastructures." In: *Computer Networks* 147 (2018), pp. 112–131.

[Doc21] P. Documentation. *Configuring Metering in PicaOS*. https://docs.pica8.com/display/PicOS374sp/Configuring+Meter. Accessed: 2021-1-17. 2021.

[Emm+14] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle. "Performance characteristics of virtual switching." In: *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*. IEEE. 2014, pp. 120–125.

[Emm+15] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. "Moongen: A scriptable high-speed packet generator." In: *Proceedings of the 2015 Internet Measurement Conference*. 2015, pp. 275–287.

[ER59] P. Erdös and A. Rényi. "On random graphs, I." In: *Publicationes Mathematicae (Debrecen)* 6 (1959), pp. 290–297.

[Err] P. Error. https://networkengineering.stackexchange.com/questions/67473/what-is-an-acceptable-bit-error-ratio.

[Esp+20] L. Espe, A. Jindal, V. Podolskiy, and M. Gerndt. "Performance Evaluation of Container Runtimes." In: *CLOSER*. 2020, pp. 273–281.

[FFV06] P. Ferrari, A. Flammini, and S. Vitturi. "Performance analysis of PROFINET networks." In: *Computer standards & interfaces* 28.4 (2006), pp. 369–385.

[Fis+13] A. Fischer, J. F. Botero, M. T. Beck, H. De Meer, and X. Hesselbach. "Virtual network embedding: A survey." In: *IEEE Communications Surveys & Tutorials* 15.4 (2013), pp. 1888–1906.

[Flo15] P. Floodlight. "Floodlight." In: *Disponıvel em: http://www. projectfloodlight. org/floodlight* (2015).

[FN12] P. Fernandes and U. Nunes. "Platooning with IVC-enabled autonomous vehicles: Strategies to mitigate communication delays, improve safety and traffic flow." In: *IEEE Transactions on Intelligent Transportation Systems* 13.1 (2012), pp. 91–106.

[For+19] G. Forecast et al. "Cisco visual networking index: global mobile data traffic forecast update, 2017–2022." In: *Update* 2017 (2019), p. 2022.

[Fou21] O. Foundation. *OpenFlow 1.4 Specification*. Dec. 2021. URL: https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.4.0.pdf.

[FSV18] K.-T. Foerster, S. Schmid, and S. Vissicchio. "Survey of consistent software-defined network updates." In: *IEEE Communications Surveys & Tutorials* 21.2 (2018), pp. 1435–1461.

[Gar+98] S. Garg, A. Van Moorsel, K. Vaidyanathan, and K. S. Trivedi. "A methodology for detection and estimation of software aging." In: *Software Reliability Engineering, 1998. Proceedings. The Ninth International Symposium on.* IEEE. 1998, pp. 283–292.

[GC12] S. Ghorbani and M. Caesar. "Walk the line: consistent network updates with bandwidth guarantees." In: *Proceedings of the first workshop on Hot topics in software defined networks.* 2012, pp. 67–72.

[Geb+15] S. Gebert, M. Jarschel, S. Herrnleben, T. Zinner, and P. Tran-Gia. "Table visor: An emulation layer for multi-table open flow switches." In: *Software Defined Networks (EWSDN), 2015 Fourth European Workshop on.* IEEE. 2015, pp. 117–118.

[Gei+17] S. Geissler, S. Herrnleben, R. Bauer, S. Gebert, T. Zinner, and M. Jarschel. "Tablevisor 2.0: Towards full-featured, scalable and hardware-independent multi table processing." In: *Network Softwarization (NetSoft), 2017 IEEE Conference on.* IEEE. 2017, pp. 1–8.

[Giv+14] O. Givehchi, J. Imtiaz, H. Trsek, and J. Jasperneite. "Control-as-a-service from the cloud: A case study for using virtualized PLCs." In: *2014 10th IEEE Workshop on Factory Communication Systems (WFCS 2014).* IEEE. 2014, pp. 1–4.

[GJF12] P. Gaj, J. Jasperneite, and M. Felser. "Computer communication within industrial distributed environment—A survey." In: *IEEE Transactions on Industrial Informatics* 9.1 (2012), pp. 182–189.

[Gro+15] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft. "Queues Don't Matter When You Can {JUMP} Them!" In: *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15).* 2015, pp. 1–14.

[Gue+14] R. Guerzoni, R. Trivisonno, I. Vaishnavi, Z. Despotovic, A. Hecker, S. Beker, and D. Soldani. "A novel approach to virtual networks embedding for SDN management and orchestration." In: *Network Operations and Management Symposium (NOMS), 2014 IEEE.* IEEE. 2014, pp. 1–7.

[Gup+01] A. Gupta, J. Kleinberg, A. Kumar, R. Rastogi, and B. Yener. "Provisioning a virtual private network: a network design problem for multicommodity flow." In: *Proceedings of the thirty-third annual ACM symposium on Theory of computing.* ACM. 2001, pp. 389–398.

[Gur16] I. Gurobi Optimization. *Gurobi Optimizer Reference Manual.* 2016. URL: http://www.gurobi.com.

[GVK17] J. W. Guck, A. Van Bemten, and W. Kellerer. "DetServ: Network models for real-time QoS provisioning in SDN-based industrial environments." In: *IEEE Transactions on Network and Service Management* 14.4 (2017), pp. 1003–1017.

[GYG13] A. Gelberger, N. Yemini, and R. Giladi. "Performance analysis of software-defined networking (SDN)." In: *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems.* IEEE. 2013, pp. 389–393.

[Han] Y. Han. “A Framework for Development, Operations, and Management of SDN-based Virtual Networks.” In: ().

[Han+15] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee. “Network function virtualization: Challenges and opportunities for innovations.” In: *IEEE communications magazine* 53.2 (2015), pp. 90–97.

[Han+18] Y. Han, T. Vachuska, A. Al-Shabibi, J. Li, H. Huang, W. Snow, and J. W.-K. Hong. “ONVisor: Towards a scalable and flexible SDN-based network virtualization platform on ONOS.” In: *International Journal of Network Management* 28.2 (2018), e2012.

[He+15a] K. He, J. Khalid, S. Das, A. Gember-Jacobson, C. Prakash, A. Akella, L. E. Li, and M. Thottan. “Latency in software defined networks: Measurements and mitigation techniques.” In: *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. 2015, pp. 435–436.

[He+15b] K. He, J. Khalid, A. Gember-Jacobson, S. Das, C. Prakash, A. Akella, L. E. Li, and M. Thottan. “Measuring control plane latency in SDN-enabled switches.” In: *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. 2015, pp. 1–6.

[He+21] Y. He, W. Wu, X. Wen, H. Li, and Y. Yang. “Scalable On-Switch Rate Limiters for the Cloud.” In: *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE. 2021, pp. 1–10.

[HG99] J. Heinanen and R. Guerin. *RFC2698: A two rate three color marker*. 1999.

[Hoc+13] D. Hock, M. Hartmann, S. Gebert, M. Jarschel, T. Zinner, and P. Tran-Gia. “Pareto-optimal resilient controller placement in SDN-based core networks.” In: *Proceedings of the 2013 25th International Teletraffic Congress (ITC)*. IEEE. 2013, pp. 1–9.

[Hoc+14] D. Hock, S. Gebert, M. Hartmann, T. Zinner, and P. Tran-Gia. “POCO-framework for Pareto-optimal resilient controller placement in SDN-based core networks.” In: *2014 IEEE Network Operations and Management Symposium (NOMS)*. IEEE. 2014, pp. 1–2.

[Hon+13] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. “Achieving high utilization with software-driven WAN.” In: *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*. 2013, pp. 15–26.

[Hua+95] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. “Software rejuvenation: Analysis, module and applications.” In: *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*. IEEE. 1995, pp. 381–390.

[HYS13] D. Y. Huang, K. Yocum, and A. C. Snoeren. “High-fidelity switch models for software-defined network emulation.” In: *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. 2013, pp. 43–48.

[Jab+16] M. Jaber, M. A. Imran, R. Tafazolli, and A. Tukmanov. “5G backhaul challenges and emerging research directions: A survey.” In: *IEEE access* 4 (2016), pp. 1743–1766.

[Jan+13] K. Jang, J. Sherry, H. Ballani, and T. Moncaster. “Silo: Predictable message completion time in the cloud.” In: *Univ. California, Berkeley, CA, USA, Tech. Rep. MSR-TR-2013-95* (2013).

[Jan+15] K. Jang, J. Sherry, H. Ballani, and T. Moncaster. “Silo: Predictable message latency in the cloud.” In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 2015, pp. 435–448.

[Jar+11] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia. “Modeling and performance evaluation of an OpenFlow architecture.” In: *2011 23rd International Teletraffic Congress (ITC)*. IEEE. 2011, pp. 1–7.

[JB04] D. Jansen and H. Buttner. “Real-time Ethernet: the EtherCAT solution.” In: *Computing and Control Engineering* 15.1 (2004), pp. 16–21.

[Jin+14] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. “Dynamic scheduling of network updates.” In: *ACM SIGCOMM Computer Communication Review* 44.4 (2014), pp. 539–550.

[Jin+15] X. Jin, J. Gossels, J. Rexford, and D. Walker. “CoVisor: A Compositional Hypervisor for Software-Defined Networks.” In: *NSDI*. Vol. 15. 2015, pp. 87–101.

[Jin+19] H. Jin, G. Yang, B. Yu, and C. Yoo. “TALON: Tenant Throughput Allocation Through Traffic Load-Balancing in Virtualized Software-Defined Networks.” In: *2019 International Conference on Information Networking (ICOIN)*. 2019, pp. 233–238.

[JKE17] H. Jmila, M. I. Khedher, and M. A. El Yacoubi. “Estimating vnf resource requirements using machine learning techniques.” In: *International Conference on Neural Information Processing*. Springer. 2017, pp. 883–892.

[Jut+01] A. Juttner, B. Szviatovski, I. Mécs, and Z. Rajkó. “Lagrange relaxation based method for the QoS routing problem.” In: *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*. Vol. 2. IEEE. 2001, pp. 859–868.

[Kan+13] N. Kang, Z. Liu, J. Rexford, and D. Walker. “Optimizing the one big switch abstraction in software-defined networks.” In: *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*. ACM. 2013, pp. 13–24.

[KCB21] E. F. Kfoury, J. Crichigno, and E. Bou-Harb. “An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends.” In: *IEEE Access* 9 (2021), pp. 87094–87155.

[KCG14] A. Krishnamurthy, S. P. Chandrabose, and A. Gember-Jacobson. “Pratyaastha: an efficient elastic distributed sdn control plane.” In: *Proceedings of the third workshop on Hot topics in software defined networking*. 2014, pp. 133–138.

[Kho+14] R. Khondoker, A. Zaalouk, R. Marx, and K. Bayarou. “Feature-based comparison and selection of Software Defined Networking (SDN) controllers.” In: *2014 world congress on computer applications and information systems (WCCAIS)*. IEEE. 2014, pp. 1–7.

[Kni+11] S. Knight, H. Nguyen, N. Falkner, R. Bowden, and M. Roughan. “The Internet Topology Zoo.” In: *Selected Areas in Communications, IEEE Journal on* 29.9 (Oct. 2011), pp. 1765–1775. ISSN: 0733-8716. DOI: 10.1109/JSAC.2011.111002.

[Kni11] S. Knight *et al.* “The Internet Topology Zoo.” In: *IEEE Journal on Selected Areas in Communications* 29.9 (Oct. 2011), pp. 1765–1775. ISSN: 0733-8716. DOI: 10.1109/JSAC.2011.111002.

[KPK14a] M. Kuzniar, P. Peresini, and D. Kostic. *What you need to know about SDN control and data planes.* Tech. rep. 2014.

[KPK14b] M. Kuzniar, P. Peresini, and D. Kostić. “Providing reliable FIB update acknowledgments in SDN.” In: *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies.* 2014, pp. 415–422.

[KPK15] M. Kuźniar, P. Perešı************************************ni, and D. Kostić. “What you need to know about SDN flow tables.” In: *International Conference on Passive and Active Network Measurement.* Springer. 2015, pp. 347–359.

[KSG14] S. Kaur, J. Singh, and N. S. Ghumman. “Network programmability using POX controller.” In: *ICCCS International conference on communication, computing & systems, IEEE.* Vol. 138. sn. 2014, p. 70.

[Kuź+18] M. Kuźniar, P. Perešı************************************ni, D. Kostić, and M. Canini. “Methodology, measurement and analysis of flow table update characteristics in hardware openflow switches.” In: *Computer Networks* 136 (2018), pp. 22–36.

[Lam01] L. Lamport. “Paxos made simple.” In: *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), pp. 51–58.

[Laz+14] A. Lazaris, D. Tahara, X. Huang, E. Li, A. Voellmy, Y. R. Yang, and M. Yu. “Tango: Simplifying SDN control with automatic switch property inference, abstraction, and optimization.” In: *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies.* 2014, pp. 199–212.

[LC11] S.-Y. Lien and K.-C. Chen. “Massive access management for QoS guarantees in 3GPP machine-to-machine communications.” In: *IEEE communications Letters* 15.3 (2011), pp. 311–313.

[Lee94] D. C. Lee. “Effects of leaky bucket parameters on the average queueing delay: Worst case analysis.” In: *Proceedings of INFOCOM'94 Conference on Computer Communications.* IEEE. 1994, pp. 482–489.

[LHM10] B. Lantz, B. Heller, and N. McKeown. “A network in a laptop: rapid prototyping for software-defined networks.” In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks.* ACM. 2010, p. 19.

[Li+13] J. Li, Q. Wang, D. Jayasinghe, J. Park, T. Zhu, and C. Pu. “Performance overhead among three hypervisors: An experimental study using hadoop benchmarks.” In: *2013 IEEE International Congress on Big Data.* IEEE. 2013, pp. 9–16.

[Li+17] Z. Li, M. Kihl, Q. Lu, and J. A. Andersson. "Performance overhead comparison between hypervisor and container based virtualization." In: *2017 IEEE 31st International Conference on advanced information networking and applications (AINA)*. IEEE. 2017, pp. 955–962.

[Lim16] E. T. Limited. *Endace DAG 7.5G4 Datasheet"*. https://www.endace.com/dag-7.5g4-datasheet.pdf. Accessed: 2018-10-26. 2016.

[Lin+17] Y.-D. Lin, Y.-K. Lai, C.-Y. Wang, and Y.-C. Lai. "Ofbench: Performance test suite on openflow switches." In: *IEEE Systems Journal* 12.3 (2017), pp. 2949–2959.

[LT01] J.-Y. Le Boudec and P. Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*. Vol. 2050. Springer Science & Business Media, 2001.

[MAC18] A. Mestres, E. Alarcón, and A. Cabellos. "A machine learning-based approach for virtual network function modeling." In: *2018 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*. IEEE. 2018, pp. 237–242.

[Mas+14] T. Mastelic, A. Oleksiak, H. Claussen, I. Brandic, J.-M. Pierson, and A. V. Vasilakos. "Cloud computing: Survey on energy efficiency." In: *Acm computing surveys (csur)* 47.2 (2014), pp. 1–36.

[McK+08] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. "OpenFlow: enabling innovation in campus networks." In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008), pp. 69–74.

[Med+14] J. Medved, R. Varga, A. Tkacik, and K. Gray. "Opendaylight: Towards a model-driven sdn controller architecture." In: *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*. IEEE. 2014, pp. 1–6.

[Mij+16] R. Mijumbi, S. Hasija, S. Davy, A. Davy, B. Jennings, and R. Boutaba. "A connectionist approach to dynamic resource management for virtualised network functions." In: *2016 12th International Conference on Network and Service Management (CNSM)*. IEEE. 2016, pp. 1–9.

[Mij+17] R. Mijumbi, S. Hasija, S. Davy, A. Davy, B. Jennings, and R. Boutaba. "Topology-aware prediction of virtual network function resource requirements." In: *IEEE Transactions on Network and Service Management* 14.1 (2017), pp. 106–120.

[MN22] J. Mellado and F. Núñez. "Design of an IoT-PLC: A containerized programmable logical controller for the industry 4.0." In: *Journal of Industrial Information Integration* 25 (2022), p. 100250.

[MNR19] D. Malkhi, K. Nayak, and L. Ren. "Flexible byzantine fault tolerance." In: *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*. 2019, pp. 1041–1053.

[MSM15] T. Mizrahi, E. Saat, and Y. Moses. "Timed consistent network updates." In: *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. 2015, pp. 1–14.

[MSM16] T. Mizrahi, E. Saat, and Y. Moses. "Timed consistent network updates in software-defined networks." In: *IEEE/ACM Transactions on Networking* 24.6 (2016), pp. 3412–3425.

[Mun21] T. U. of Munich. *5G Research Hub Munich*. Feb. 2021. URL: https://www.5g-munich.de/.

[MW13] R. Mahajan and R. Wattenhofer. "On consistent updates in software defined networks." In: *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. 2013, pp. 1–7.

[Nao+08] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown. "Implementing an OpenFlow switch on the NetFPGA platform." In: *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. 2008, pp. 1–9.

[Nas+18] A. Nasrallah, A. S. Thyagaturu, Z. Alharbi, C. Wang, X. Shao, M. Reisslein, and H. El-Bakoury. "Ultra-low latency (ULL) networks: The IEEE TSN and IETF DetNet standards and related 5G ULL research." In: *IEEE Communications Surveys & Tutorials* 21.1 (2018), pp. 88–145.

[NCC17] T. D. Nguyen, M. Chiesa, and M. Canini. "Decentralized consistent updates in SDN." In: *Proceedings of the Symposium on SDN Research*. 2017, pp. 21–33.

[Net21] E. Networks. *EdgeCore Wedge 100BF-32X*. 2021. URL: https://www.edge-core.com/productsInfo.php?cls=1&cls2=5&cls3=181&id=335.

[Nun+14] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti. "A survey of software-defined networking: Past, present, and future of programmable networks." In: *IEEE Communications surveys & tutorials* 16.3 (2014), pp. 1617–1634.

[Nur+19] G. N. Nurkahfi, A. Mitayani, V. A. Mardiana, and M. M. M. Dinata. "Comparing FlowVisor and Open Virtex as SDN-Based Site-to-Site VPN Services Solution." In: *2019 International Conference on Radar, Antenna, Microwave, Electronics, and Telecommunications (ICRAMET)*. 2019, pp. 142–147.

[OO14] D. Ongaro and J. Ousterhout. "In search of an understandable consensus algorithm." In: *2014 USENIX Annual Technical Conference (Usenix ATC 14)*. 2014, pp. 305–319.

[Orl+07] S. Orlowski, M. Pióro, A. Tomaszewski, and R. Wessäly. "SNDlib 1.0–Survivable Network Design Library." English. In: *Proceedings of the 3rd International Network Optimization Conference (INOC 2007), Spa, Belgium*. http://sndlib.zib.de, extended version accepted in Networks, 2009. Apr. 2007. URL: http://www.zib.de/orlowski/Paper/OrlowskiPioroTomaszewskiWessaely2007-SNDlib-INOC.pdf.gz.

[PMK13] G. Pongrácz, L. Molnár, and Z. L. Kis. "Removing roadblocks from SDN: OpenFlow software switch performance on Intel DPDK." In: *2013 Second European Workshop on Software Defined Networks*. IEEE. 2013, pp. 62–67.

[PSS16] N. Panwar, S. Sharma, and A. K. Singh. "A survey on 5G: The next generation of mobile communication." In: *Physical Communication* 18 (2016), pp. 64–84.

[Qin+12] S. Qing, Q. Qi, J. Wang, T. Xu, and J. Liao. "Topology-aware virtual network embedding through bayesian network analysis." In: *Global Communications Conference (GLOBECOM), 2012 IEEE*. IEEE. 2012, pp. 2621–2627.

[Rei+12] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. “Abstractions for network update.” In: *ACM SIGCOMM Computer Communication Review* 42.4 (2012), pp. 323–334.

[Rod] G. Rodola. “Psutil package: a cross-platform library for retrieving information onrunning processes and system utilization.” In: *Google Scholar* ().

[Rot+12] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore. “OFLOPS: An open framework for OpenFlow switch evaluation.” In: *International Conference on Passive and Active Network Measurement*. Springer. 2012, pp. 85–95.

[Rou05] M. Roughan. “Simplifying the synthesis of Internet traffic matrices.” In: *ACM SIGCOMM Computer Communication Review* 35.5 (2005), pp. 93–96.

[Row+14] S. Rowshanrad, S. Namvarasl, V. Abdi, M. Hajizadeh, and M. Keshtgary. “A survey on SDN, the future of networking.” In: *Journal of Advanced Computer Science & Technology* 3.2 (2014), pp. 232–248.

[Ryu15] Ryu, SDN. *Framework community: Ryu SDN framework*. http://osrg.github.io/ryu. 2015.

[Sae+17] A. Saeed, N. Dukkipati, V. Valancius, V. The Lam, C. Contavalli, and A. Vahdat. “Carousel: Scalable traffic shaping at end hosts.” In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 2017, pp. 404–417.

[Sal+11] E. Salvadori, R. D. Corin, A. Broglio, and M. Gerola. “Generalizing virtual network topologies in OpenFlow-based networks.” In: *Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE*. IEEE. 2011, pp. 1–6.

[Sha+13] S. A. Shah, J. Faiz, M. Farooq, A. Shafi, and S. A. Mehdi. “An architectural evaluation of SDN controllers.” In: *2013 IEEE international conference on communications (ICC)*. IEEE. 2013, pp. 3504–3508.

[She+09] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. “Flowvisor: A network virtualization layer.” In: *OpenFlow Switch Consortium, Tech. Rep* 1 (2009), p. 132.

[Shi20] K. Shinde. “Nimble: Scalable Rate-Limiting on Today’s Programmable Switches.” PhD thesis. University of Illinois at Chicago, 2020.

[Shu+21] A. Shukla, K. Hudemann, Z. Vági, L. Hügerich, G. Smaragdakis, A. Hecker, S. Schmid, and A. Feldmann. “Fix with P6: Verifying Programmable Switches at Runtime.” In: (2021).

[Sie+16] C. Sieber, A. Basta, A. Blenk, and W. Kellerer. “Online resource mapping for SDN network hypervisors using machine learning.” In: *NetSoft Conference and Workshops (NetSoft), 2016 IEEE*. IEEE. 2016, pp. 78–82.

[SK17] E. Sakic and W. Kellerer. “Response time and availability study of RAFT consensus in distributed SDN control plane.” In: *IEEE Transactions on Network and Service Management* 15.1 (2017), pp. 304–318.

[SK18] E. Sakic and W. Kellerer. "Impact of adaptive consistency on distributed sdn applications: An empirical study." In: *IEEE Journal on Selected Areas in Communications* 36.12 (2018), pp. 2702–2715.

[SOK17] C. Sieber, A. Obermair, and W. Kellerer. "Online learning and adaptation of network hypervisor performance models." In: *Integrated Network and Service Management (IM), 2017 IFIP/IEEE Symposium on*. IEEE. 2017, pp. 1204–1212.

[SOS13] S. Scott-Hayward, G. O'Callaghan, and S. Sezer. "SDN security: A survey." In: *2013 IEEE SDN For Future Networks and Services (SDN4FNS)*. IEEE. 2013, pp. 1–7.

[Spe09] O. S. Specification. "Version 1.0. 0 (Wire Protocol 0x01)." In: *Open Networking Foundation* (2009).

[Suk+16] K. Suksomboon, M. Fukushima, S. Okamoto, and M. Hayashi. "A dilated-CPU-consumption-based performance prediction for multi-core software routers." In: *2016 IEEE NetSoft Conference and Workshops (NetSoft)*. IEEE. 2016, pp. 193–201.

[SUK19] H. Saini, A. Upadhyaya, and M. K. Khandelwal. "Benefits of cloud computing for business enterprises: A review." In: *Proceedings of International Conference on Advancements in Computing & Management (ICACM)*. 2019.

[Tan+19] L. Tang, X. He, P. Zhao, G. Zhao, Y. Zhou, and Q. Chen. "Virtual network function migration based on dynamic resource requirements prediction." In: *IEEE Access* 7 (2019), pp. 112348–112362.

[TG10] A. Tootoonchian and Y. Ganjali. "Hyperflow: A distributed control plane for openflow." In: *Proceedings of the 2010 internet network management conference on Research on enterprise networking*. Vol. 3. 2010.

[Tha+21] V. S. Thapeta, K. Shinde, M. Malekpourshahraki, D. Grassi, B. Vamanan, and B. E. Stephens. "Nimble: Scalable TCP-Friendly Programmable In-Network Rate-Limiting." In: *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*. 2021, pp. 27–40.

[Thi+19] V. Thirupathi, C. Sandeep, N. Kumar, and P. P. Kumar. "A comprehensive review on sdn architecture, applications and major benifits of SDN." In: *International Journal of Advanced Science and Technology* 28.20 (2019), pp. 607–614.

[TPR14] S. Tomovic, M. Pejanovic-Djurisic, and I. Radusinovic. "SDN based mobile networks: Concepts and benefits." In: *Wireless Personal Communications* 78.3 (2014), pp. 1629–1644.

[Van20] A. Van Bemten. *Chameleon Controller*. https://github.com/AmoVanB/chameleon-controller. 2020.

[Van21] A. Van Bemten. *Chameleon: network controller*. https://github.com/amovanb/chameleon-controller. 2021.

[VHV12] B. Vamanan, J. Hasan, and T. Vijaykumar. "Deadline-aware datacenter tcp (d2tcp)." In: *ACM SIGCOMM Computer Communication Review* 42.4 (2012), pp. 115–126.

[VK16] A. Van Bemten and W. Kellerer. "Network calculus: A comprehensive guide." In: (2016).

[Wan+20] M. Wang, T. Zhu, T. Zhang, J. Zhang, S. Yu, and W. Zhou. "Security and privacy in 6G networks: New areas and new challenges." In: *Digital Communications and Networks* 6.3 (2020), pp. 281–291.

[WMZ19] J. Woodruff, A. W. Moore, and N. Zilberman. "Measuring burstiness in data center applications." In: *Proceedings of the 2019 Workshop on Buffer Sizing*. 2019, pp. 1–6.

[WS98] D. J. Watts and S. H. Strogatz. "Collective dynamics of small-world networks." In: *nature* 393.6684 (1998), p. 440.

[You] YouTube. *YouTube recommended upload encoding settings - youtube help*. URL: https://support.google.com/youtube/answer/1722171?hl=en#zippy=%2Cvideo-codec-h%2Caudio-codec-aac-lc%2Ccontainer-mp%2Cbitrate.

[Yu+08] M. Yu, Y. Yi, J. Rexford, and M. Chiang. "Rethinking virtual network embedding: substrate support for path splitting and migration." In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008), pp. 17–29.

[ZA20] A.-B. Zainab and M. B. Al-Somaidai. "Latency Evaluation of an SDN Controlled by Flowvisor and Two Heterogeneous Controllers." In: *International Conference on New Trends in Information and Communications Technology Applications*. Springer. 2020, pp. 3–16.

[Zac+18] G. A. Zachiotis, G. Andrikopoulos, R. Gornez, K. Nakamura, and G. Nikolakopoulos. "A survey on the application trends of home service robotics." In: *2018 IEEE international conference on Robotics and Biomimetics (ROBIO)*. IEEE. 2018, pp. 1999–2006.

[Zha+17] M. Zhao, A. Kumar, T. Ristaniemi, and P. H. J. Chong. "Machine-to-machine communication and research challenges: A survey." In: *Wireless Personal Communications* 97.3 (2017), pp. 3569–3585.

[Zho+21] Z. Zhou, M. He, W. Kellerer, A. Blenk, and K.-T. Foerster. "P4Update: fast and locally verifiable consistent network updates in the P4 data plane." In: *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*. 2021, pp. 175–190.

[Zhu+20] L. Zhu, M. M. Karim, K. Sharif, C. Xu, F. Li, X. Du, and M. Guizani. "SDN controllers: A comprehensive analysis and performance evaluation study." In: *ACM Computing Surveys (CSUR)* 53.6 (2020), pp. 1–40.

Acronyms and Abbreviations

SotA *state-of-the-art* 4–8, 11, 14, 18, 21, 23, 26, 27, 29–32, 47, 75–77, 85, 87, 88, 101–104, 108, 114, 123–126, 129

AC Admission Control 104, 105, 107–110

AGV Automated Guided Vehicle 101

AI Artificial Intelligence 74, 130

BFT Byzantine Fault Tolerance 11

BGP Border Gateway Protocol 9

CAT Cache Allocation Technology 26, 93

CI/CD Continuous Integration/Continuous Developement 18

COTS Commercial Off-The-Shelf 18

CP Control Plane ii, v–viii, 1–12, 14, 15, 19, 20, 27, 29, 31, 32, 42, 47–52, 55, 56, 59, 64, 67, 70, 80, 101–112, 114–118, 120–123, 125–129, 151, 152, 155

CPS Cyber-physical System 127

CPU Central Processing Unit i, 3–5, 7, 17, 29–70, 73, 74, 93, 94, 111, 112, 123, 128, 130, 152, 153

DC Data Center 4, 6, 9, 21, 24, 25, 102, 127, 131

DNC Deterministic Network Calculus 6, 7, 22, 24, 75, 77, 78, 80, 81, 83, 84, 97–99, 104, 106, 107, 114, 115, 127–131

DP Data Plane v–viii, 1–10, 15, 16, 19–22, 24, 26, 27, 29–36, 42, 47, 50, 51, 53, 70, 73, 75, 80, 85, 91, 98, 101–103, 106, 110–116, 118, 119, 121, 123, 125–131, 151

DPDK Data Plane Development Kit 18, 25, 26, 76, 91–93, 95, 96, 98, 102, 112, 119, 129, 151, 155

DSCP Differentiated Services Code Point 78

DUT Device Under Test 85, 112–114, 116, 130, 155

EdgeCore Wedge Edgecore Wedge 100BF-32X System 111–116, 119, 120, 155

ETF Earliest TxTime First 95

FV FlowVisor 50, 52–54, 59

ILP Integer Linear Programming 5, 128

Intel Tofino™ P4-programmable Intel Tofino™ Intelligent Fabric Processor (IFP) ASIC 114, 115

IoT Internet of Things 127

IP Internet Protocol 12, 16, 49, 51, 54, 64, 78, 85, 105

M2M Machine-to-Machine 1

MAC Media Access Control 10, 25, 87, 94

MEC Multi-access Edge Computing v, vii

ML Machine Learning 74, 130

NaaS Network as a Service v, vii, 18

NAT Network Address Translation 18, 74

NBI Northbound Interface 7, 24, 34, 36, 46, 52, 67, 114, 119

NFV Network Function Virtualization 9, 16, 18, 27, 128

NH Network Hypervisor 2–7, 19, 20, 29–37, 39–60, 62, 64, 65, 68–70, 73, 74, 128–130, 151–153

NIC Network Card Interface 15, 25, 92, 95, 112, 119

OF Open Flow 2, 4, 6, 7, 10, 12–15, 19, 22, 26, 27, 32, 34, 46, 51, 53, 54, 75, 80, 81, 84, 85, 97, 98, 102, 129, 130

OVX OpenVirtex 31, 33, 35, 74

P4 Programming Protocol-independent Packet Processors 1, 5–7, 9, 15, 16, 18, 22, 26, 111, 112, 114, 129, 130, 155

PC Personal Computers 33, 41, 50, 51, 152

PLC Programmable Logic Controller 1

QoS Quality of Service v–viii, 1–8, 11, 18, 22, 26, 30, 31, 47, 51, 70, 72, 73, 80, 98, 101, 119, 126–129, 131, 151

RAM Random Access Memory 17, 29, 32, 50, 95, 123

SBI Southbound Interface 7, 34, 36, 52

SDN Software-Defined Networking i, v, vii, 1–3, 5, 7–16, 18–27, 29, 32–34, 46, 50, 52–54, 70, 74, 80, 85, 102, 104, 118, 131, 151, 152

SLA Service-Level Agreement 75

TCP Transmission Control Protocol 12, 48, 49, 51, 54, 64, 89

TLS Transport Layer Security 12

TSC Time Stamp Counter 94, 95

TSN Time-Sensitive Networking 125

UDP User Datagram Protocol 112

VL Virtualization Layer 32, 70

VLAN Virtual Local Area Network 16, 25, 90, 93–95, 103, 111, 114, 117

VM Virtual Machine 17, 18, 25, 30, 33, 35, 45, 70, 81, 92, 93, 95, 151, 152

VMDQ Virtual Machine Device Queues 25, 92

VN Virtual Network 5, 7, 19, 20, 30, 31, 34, 35, 38, 39, 41–50, 52–54, 56–67, 69–73, 152

VNE Virtual Network Embedding 4, 5, 7, 20, 29–32, 36, 39, 41–43, 46, 47, 73, 74, 128

VNF Virtual Network Function 18, 48, 49, 64, 71–74, 154

VNR Virtual Network Request 30, 39–43, 45, 46

VPNE Virtual Private Network Embedding 39

WAN Wide Area Network 21, 102

List of Figures

1.1 Architecture of virtualized SDN network. Additionally, the main research areas of the contributions presented in this thesis are highlighted. In this scenario, the NH virtualizes the underlying physical network, and it provides two virtual SDN networks to two different tenants (i.e., to tenants *A* and *B*). Thus, enabling the tenants to control their own virtual network through their SDN controller. 3

1.2 An overview of the structure of this thesis. In this thesis, three different main research direction related to the QoS-aware virtualization of programmable networks are investigated. The first direction focuses on provisioning of the CP of virtualized SDN networks. While the following two directions are focused on the challenges of achieving guaranteed QoS in the DP. 8

2.1 (a) Legacy and (b) SDN network architecture. 11

2.2 P4 Processing Pipeline. 15

2.3 (a) Non-virtualized and (b) Virtualized server. Each black dashed rectangleon the right figure represents one Virtual Machine (VM)s. 17

2.4 Overview of network virtualization. A network hypervisor acts as a proxy between the tenants controllers and the physical infrastructure. A hypervisor can provide different virtualization policies to different tenants, i.e., in this case different levels of topology abstraction. 19

2.5 Example of an *arrival curve* (red dashed line) and a *service curve* (blue dotted line). 23

2.6 Example of a small network with 1G links and two flows, where each flow utilizes 0.5G. Embedding a new flow request of 1G between switch A and B is not possible without reconfiguration. 23

2.7 Chameleon architecture. Taken from [Van+20]. 24

2.8 Architecture of DPDK application running on end-host. 25

2.9 (a) Measured packet latency and loss of 5 flows in the deployed network managed by *Chameleon*. (b) Number of accepted flows in 8 different scenarios by *Chameleon*,*Qjump* and *Silo*. Source: [Van+20]. 26

3.1 Control and Data-plane measurement setup. 33

3.2 (a) Physical representation of the measurement topology, and the two considered topology abstraction cases: (b) transparent case with no abstraction and (c) big-switch abstraction. 34

3.3 CPU utilization with respect to the flow rate between two hosts when there are k number of switches between them. Left column of each box plot represents the case when Virtual Network (VN) is *transparently* embedded, while the right column box plots represents if the VN is embedded using *big-Switch*. 35
3.4 Mean measured CPU utilization for the transparent case with k = 10 switches in the data plane and the corresponding estimation models. 37
3.5 Estimation of CPU utilization based on the presented model for both abstraction cases, i.e., transparent (no) abstraction and big-switch abstraction. 38
3.6 Physical representation of the topology and three supported VNE abstraction levels. . . . 38
3.7 Required NH CPU resources based on topology abstraction. 43
3.8 Difference in achieved objective function values when topology abstraction is considered and when it is not. Normalized with the maximal achieved value for maximize data-plane utilization objective. 44
3.9 Distribution of topology abstraction levels of embedded VM when the NH CPU resources are varied. 45
3.10 Impact of the allocation of CPU resources of an NH on the control plane message latency and loss. Dashed black lines show the maximal and average observed CPU utilization during an unconstrained run, where all of the cores where allocated to network hypervisor (CPU limit is 800 %). The server running an NH has in total 4 physical cores, thus 8 hyperthreads, making the maximal CPU utilization of 800 %. 48
3.11 (a) Measurement setup consisting of 2 interconnected Personal Computers (PC)s, and (b) illustrations of the physical data plane grid topology of dimension $k = 3$ (black solid lines), with example grid VN of dimension $k_v = 2$ (red dashed lines), and a possible flow request spanning over two virtual and physical switches (blue dotted line). 50
3.12 Benchmarking procedure. PC1 emulates the data plane and the tenants SDN controllers, while PC2 is running the virtualization layer and CPU monitoring script. 53
3.13 Observed time series of CPU utilization (red line with left y-axis) during one measurement run with the following parameters: $t = 5$, $r_i \sim 184$, $k = 4 \times 4$, $v_i = 4 \times 4$, $l_i = 4$ and $p_i = 1$. Furthermore, the green scattered line with the right axis represents time series of the total observed flow request rate per second during the aforementioned run. 55
3.14 Distributions of measured CPU utilization for five measurement runs with identical evaluation parameters, i.e., $r_i \sim 300$, $k = 5 \times 5$, $v_i = 4 \times 4$, $l_i = 4$ and one host per virtual switch. The boxplots whiskers correspond to the 5% and 95% percentiles. The green line shows the mean total `OF_FlowModAdd` rate and its standard deviation. 55
3.15 Impact of limiting the CPU resources allocated to NH on the observed CP processing time latency (median, mean and maximal) for two different measurement cases. Every run with different CPU allocation limit is repeated 5 times, and the mean values of these 5 runs are shown with their corresponding confidence intervals. 56

3.16 (a)–(f) Impact of different evaluation parameters on the NH 90th percentile CPU utilization on a physical grid topology for 2 to 5 tenants. Plots show mean observed CPU utilization values of 10 runs along with the 95% confidence intervals assuming uniform distributions. The following parameters are presented: (a) path length with $r_i = 184$, $k = 4 \times 4$, $v_i = 4 \times 4$ and $p_i = 1$, (b) flow request rate with $k = 5 \times 5$, $v_i = 4 \times 4$, $l_i = 5$ and $p_i = 1$, (c) virtual topology size with $r = 430$, $k = 5 \times 5$, $l_i = 3$ and $p_i = 1$, (d) physical topology size with $r = 430$, $v_i = 3 \times 3$, $l_i = 3$ and $p_i = 1$, (e) number of virtual ports dedicated for connecting the hosts per switch with $r = 184$, $k = 4 \times 4$, $v_i = 4 \times 4$, $l = 4$. (f) shows the data of (a)–(f) as box plots for the different number of tenants (5% and 95% percentile whiskers are used). 57
3.17 Cumulative distribution function (CDF) of the absolute fitting error of the grid measurement data with a linear based model (Eg. 3.11). 60
3.18 Computation of p_i^e based on v_i. Values are interpolated (green dashed line) from the average number of virtual ports per switch for grid topologies (red crosses) and converge to 5 (thick dashed gray asymptote) as internal nodes in a grid topology connect to 4 other nodes and one host. 61
3.19 (a)–(f) Estimated and measured 90th percentile CPU utilization for 100 randomly generated scenarios with 2-5 tenants using the considered topologies (i.e., Internet2, NobelEU, Ring, Erdos-Reny and Watts-Strogatz). The scenarios are sorted based on the measured CPU utilization in the ascending order. The absolute error is shown as red starred line, while the mean observed error is shown as horizontal red dashed line. 63
3.20 Box plots of the prediction accuracy of the three models f_2 (Eq. 3.12, proposed model), f_1 (Eq. 3.11, proposed model without port scaling factor) and f_3 (Eq. 3.15, only rate based model - *SotA*). The dashed gray line indicates the average fitting error of the grid measurements, thus, applying the model on different topologies with randomly generated requests on average introduces a slight penalty, i.e., the average error increases by a few percents. 65
3.21 Impact of the number of virtual switches per tenant and topology type on the mean observed relative error for (a) proposed model and (b) the baseline. The date is separated in 6 uniformly created bins, and the mean values of each bin are shown. 66
3.22 Absolute error dependency on (a) the average number of virtual switches per VN, (b) the total `OF_FlowModAdd` rate, (c) the measured CPU utilization, and (d) the total number of tenants. The observed data on figures (a)–(c) is pre-processed by binning it into 10 equally sized bins. For each bin, mean and standard deviation are shown as error bars, while green stars indicate the corresponding maximum. 67
3.23 (a)–(f) Impact of 4 different NH CPU provisioning strategies on the observed message processing time of an NH. For each run, four different statistical properties are considered: median, mean, 90th percentile, and maximal latency. As we have 400 different scenarios, each box plot represents 400 observed values of a certain statistical property. . . 68

3.24 The demonstration architecture and scenario show *i)* a European-based data plane topology with a centralized server located in the proximity of Helsinki, Finland, while the edge servers are positioned in the proximity of big cities and *ii)* the virtualization provided by the virtualization layer, i.e., HyperFLEX, and *iii)* two tenants that share the physical architecture, highlighted in blue and orange colours. 71
3.25 Networking delay between the robot and the remote controller based on the given demonstration topology. During the first 46 seconds (left side of the red dashed line), the firewall Virtual Network Function (VNF) is located on the centralized server. The right side of the red line shows the network latency after the migration. 72

4.1 Diagram representing a leaky token bucket algorithm. The maximal token bucket size is b, while the rate of token generation rate is r. 78
4.2 Example of discrete data being policed (or constrained) by a policer based on the token bucket algorithm. The red line represents a token bucket arrival curve. 79
4.3 Diagram representing two-rate-two-bucket token bucket algorithm (as defined in RFC 2698). r_{PIR} is peak information rate, b_{PBS} is peak burst size, r_{CIR} committed information rate, b_{CBS} commited burst size. 79
4.4 Generic testbed. 81
4.5 An illustration of one time-wise sequence of pre-policed and policed packets with the corresponding token bucket state. The scenario depicts two packet groups, where each one contains at least 3 packets that are not part of the initial burst. 83
4.6 An illustration of one time-wise sequence of policed packets (measured traffic trace) with the corresponding self-deconvoluted function and the constraining token bucket arrival curve. 85
4.7 The measurement setup. 86
4.8 Impact of metering on the processing time. All the measurement data is aggregated. . . . 88
4.9 Impact of metering on the processing time for five different measurement scenarios. . . . 88
4.10 An initial part of the policed traffic trace, its min-plus self-deconvolution, and the derived token bucket arrival curve for the considered switches. In all three scenarios, policing rate is 1 Mbps, and the packet size is 500 bytes. The configured burst size for *Pica8 P-3290* is 100 kbits (or 25 500 byte packets) and for *Pica8 P-3297* is 50 kbps (approx 12.5 500 byte packets). 89
4.11 Achieved accuracy (in terms of relative deviation) of rate policing depending on the configured policing rate. We consider four different measurement configurations with the following parameters, $c1 \longrightarrow (b = 30$ kbps, $s = 100$ bytes, $n = 1)$; $c2 \longrightarrow (b = 100$ kbps, $s = 500$ bytes, $n = 1)$; $c3 \longrightarrow (b = 200$ kbps, $s = 1500$ bytes, $n = 1)$; $c4 \longrightarrow (b = 75$ kbps, $s = 1000$ bytes, $n = 1)$. Since it is not possible to configure the burst size on *HP E3800*, the corresponding parameters is ignored. 90
4.12 Derived burst size from the measurements (with different packet sizes s) for *Pica8 P3297* and *HP E3800*. The derived rate for *Pica8 P3297* in all presented scenarios deviates at most 0.5% from the configured value. 91
4.13 Microbursts generated by *Pica8* and HP devices at higher policing rates (i.e., $r = 100$ Mbps) with smaller packet sizes (e.g., $s \leq 500$ bytes). 92

4.14 Architecture of DPDK application running on end-host. 93
4.15 Comparison of DPDK-based traffic policing and Pica8 P3297 for different configured policing rates. (a) Figure with full *y-axis* range, and (b) zoomed figure with shorter *y-axis* range. 96
4.16 The role of accurate traffic policing in (a) number of accepted flows, and (b) network utilization. 98

5.1 An example of consistent reconfiguration procedure. Before embedding Flow 2, Flow 1 has to be rerouted consistently. C_1 is a centralized network controller running `NAGA`. . . . 103
5.2 `NAGA` system model. Example: The controller configures S_3 to embed a flow (2) between H_2 and H_3 by sending a rule via in-band CP flow. 104
5.3 (a) Network example and (b) the respective CP message sequence chart. Two flow updates (e.g., u_m and u_{m+1}) are sent from the controller to the edge switch S_k. s', j', p', q' represent the total propagation, sending, processing, and queuing time update u_m experienced while traversing S_{k-1}. 106
5.4 (a) Network example and (b) the respective CP message sequence chart. 108
5.5 Logical representation of the developed P4 application and a local python application. . . 112
5.6 Processing time measurement setup. 113
5.7 The total forwarding and processing time of the tagging and metering application on EdgeCore Wedge. 113
5.8 Rate metering accuracy. 115
5.9 Burst metering accuracy. 115
5.10 Different CP traffic generation procedures. In this scenario, the CP packet (green packet) contains a batch of 3 rule insertions (i.e., 1, 2, 3). (b) Example of the second CP traffic generation procedure. In this scenario, each CP packet contains a batch of 2 rule insertions (e.g., batch a contains rules 1 and 2). Ports 1 and 2 of the measurement card correspond to the QSFP Ports 1(a) and 3(a) of DUT. 116
5.11 Control plane predictability. 117
5.12 (a) Total time needed to insert or reconfigure a batch of 34 rules in different configuration scenario. (b) Worst-case reconfiguration time of each rule (in a batch) for 4 measurement scenarios with different amount of pre-populated rules (with 10 repetitions). 118
5.13 The logical test-bed network. 119
5.14 The measured end-to-end packet delay in the system. 120
5.15 The measured update time (black line) vs. deadline (red line). 120
5.16 Verifying the consistency of the flow updates with `NAGA`. 122
5.17 The cost of in-band CP. 122
5.18 Comparison of network update rate. 124
5.19 Comparison of network update deadlines. 125

List of Tables

2.1 Supported header fields in OF 1.0. 13

3.1 Values of Modeling Parameters 37
3.2 Simulation Parameters 41
3.3 Evaluation parameters. 51
3.4 Physical topologies considered and their number of nodes and edges and their density. As a comparison, 5×5 and 6×6 grids have densities of 1.6 and 1.67 respectively. 62
3.5 Distribution of parameters for the final evaluation. $U(x, y)$ denotes a uniform distribution between x and y. 62

4.1 Specifications of the evaluated switches: names, ASIC, CPU, and ports. 86
4.2 Considered Parameters 86
4.3 Supported Policing Flags, and Number of Meters 87
4.4 Considered parameter and their values in the second measurement campaign. 96

5.1 Considered flow types and their characteristics. 119