Technische Universität München
TUM School of Computation, Information and Technology

Performance Modeling, Optimization, and Applications for the Deployment of Programmable Packet Processors in Cloud Environments

Hasanin Harkous, M.Sc.

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

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

genehmigten Dissertation.

Die Dissertation wurde am 13.06.2022 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 17.11.2022 angenommen.

Performance Modeling, Optimization, and Applications for the Deployment of Programmable Packet Processors in Cloud Environments

Hasanin Harkous, M.Sc.

Abstract

The continuous advancement of user applications is imposing bigger challenges on the underlying communication networks. The expectations from these networks are surpassing basic connectivity to also include forwarding at high performance in terms of throughput, latency, etc., and being flexible to support new services and functionalities that range over the different layers of the networking stack such as load balancing, filtering, intrusion detection, and congestion control. To fulfill these requirements, different solutions such as Software-Defined Networking (SDN), Network Function Virtualization (NFV), and more recently P4 programmability were proposed. The P4 programmability is considered a promising technology since it extends the flexibility provided by the SDN paradigm by pushing programmability to the data plane of packet processors, and it enhances the performance of the NFV paradigm by enabling programmability on hardware accelerators. Although the integration of P4 into NFV cloud environments can have huge potential in terms of flexibility and performance, this integration also raises a new plane of problems and challenges in terms of the design and management of such environments.

The goal of this doctoral thesis is to measure and model the performance of P4 programmable packet processors toward enabling the optimal performance-aware management of NFV cloud environments with the P4 technology incorporated into it. This goal is reached step-by-step by addressing the following challenges.

First, since the P4 programming language is target independent, meaning that it can be used to program different types of processing platforms such as CPUs, NPUs, FPGAs, and ASICs, it is important to evaluate and fairly compare the performance of these different types of devices to assess their suitability for different use case scenarios. In addition, keeping in mind that P4 technology permits programming the data plane of packet processors, it is important to understand the relation between the complexity of the configured P4 pipeline and the packet processing latency on the running P4 device. This thesis contributes to the conduction of a comprehensive measurement campaign to understand the capabilities and limitations of different P4 programmable packet processors, with a special focus on understanding the effect of the complexity of the P4 data plane on the packet forwarding latency on different P4 devices. Moreover, this thesis contributes to the development of a novel tool for benchmarking the performance of P4Runtime-based controllers, which is used to evaluate the performance of a P4-compatible SDN controller.

The P4-based system is made up of programmable data and control planes that interact with each other. Different factors influence the performance of this system such as the forwarding latency at the control and data planes, the frequency of interaction between the control and data planes, etc. This thesis leverages the collected measurements in the first objective to model the performance of P4-based systems in two stages: (1.) A method is proposed to estimate the forwarding latency on P4 data planes as a function of the configured P4 programs; (2.) A feedback-oriented queueing system is used to model the performance of P4-based systems, whose data plane's forwarding latency is variable and can be quantified using the derived estimation method in the first modeling stage.

When P4 programmable packet processors are to be deployed in cloud environments, they create a heterogeneous pool of processing devices of distinct types each with different performance levels, capabilities, and limitations. This diversity in processing platforms gives an additional degree of freedom in selecting the platform that best hosts a certain workload with given Quality of Service (QoS) requirements. This is only possible if proper management of such an environment is realized. The thesis leverages the performance models developed in the previous objective to contribute two performance-aware optimization problems for the planning of the infrastructure substrate of P4-based cloud environments and the runtime embedding of processing workloads requests while satisfying QoS requirements.

Finally, we implement and analyze two applications to show the advantages of using programmable data planes in cloud environments for future networks. Given that cloud-native solutions are prevailing in cellular networks, we select the User Plane Function (UPF) as the first studied use case NF that can run in clouds and make use of the P4 technology. The thesis proposes and implements a microservice-based UPF to demonstrate the usability of the P4-based cloud environment and the management scheme proposed in the previous objective. The second application mitigates the P4 language's lack of support for programmable traffic management by contributing a solution that implements virtual queues to achieve this target. The programmable traffic manager application holds as a standalone application for customizing the rate and delay on a per slice basis to meet the required service level agreements, and it complements the UPF implementation as it enables the enforcement of different required QoS levels.

Kurzfassung

Die ständige Weiterentwicklung der Benutzeranwendungen stellt die zugrunde liegenden Kommunikationsnetze vor immer größere Herausforderungen. Die Erwartungen an diese Netze gehen über die grundlegende Konnektivität hinaus und umfassen auch die Weiterleitung mit hoher Leistung in Bezug auf Durchsatz, Latenz usw. sowie die flexible Unterstützung neuer Dienste und Funktionen, die sich über die verschiedenen Schichten des Netzwerkstapels erstrecken, wie Lastausgleich, Filterung, Intrusion Detection und Staukontrolle. Um diese Anforderungen zu erfüllen, wurden verschiedene Lösungen wie Software-Defined Networking (SDN), Network Function Virtualization (NFV) und in jüngerer Zeit die P4-Programmierbarkeit vorgeschlagen. Die P4-Programmierbarkeit gilt als vielversprechende Technologie, da sie die Flexibilität des SDN-Paradigmas erweitert, indem sie die Programmierbarkeit auf die Datenebene der Paketprozessoren verlagert, und sie verbessert die Leistung des NFV-Paradigmas, indem sie die Programmierbarkeit auf Hardwarebeschleunigern ermöglicht. Obwohl die Integration von P4 in NFV-Cloud-Umgebungen ein enormes Potenzial in Bezug auf Flexibilität und Leistung haben kann, wirft diese Integration auch eine neue Ebene von Problemen und Herausforderungen in Bezug auf den Entwurf und die Verwaltung solcher Umgebungen auf.

Das Ziel dieser Doktorarbeit ist es, die Leistung von programmierbaren P4-Paketprozessoren zu messen und zu modellieren, um ein optimales, leistungsbewusstes Management von NFV-Cloud-Umgebungen zu ermöglichen, in die die P4-Technologie integriert ist. Dieses Ziel wird schrittweise durch die Bewältigung der folgenden Herausforderungen erreicht.

Erstens: Da die P4-Programmiersprache zielunabhängig ist, was bedeutet, dass sie zur Programmierung verschiedener Arten von Verarbeitungsplattformen wie CPUs, NPUs, FPGAs und ASICs verwendet werden kann, ist es wichtig, die Leistung dieser verschiedenen Gerätetypen zu bewerten und fair zu vergleichen, um ihre Eignung für verschiedene Anwendungsszenarien zu beurteilen. Da die P4-Technologie die Programmierung der Datenebene von Paketprozessoren erlaubt, ist es außerdem wichtig, die Beziehung zwischen der Komplexität der konfigurierten P4-Pipeline und der Paketverarbeitungslatenz auf dem laufenden P4-Gerät zu verstehen. Diese Arbeit trägt zur Durchführung einer umfassenden Messkampagne bei, um die Fähigkeiten und Grenzen verschiedener programmierbarer P4-Paketprozessoren zu verstehen, wobei ein besonderer Schwerpunkt auf dem Verständnis der Auswirkungen der Komplexität der P4-Datenebene auf die Latenzzeit bei der Paketweiterleitung auf verschiedenen P4-Geräten liegt. Darüber hinaus trägt diese Arbeit zur Entwicklung eines neuartigen

Werkzeugs zum Benchmarking und Bewerten der Leistung von P4Runtime-basierten Controllern bei.

Das P4-basierte System besteht aus programmierbaren Daten- und Steuerungsebenen, die miteinander interagieren. Verschiedene Faktoren beeinflussen die Leistung dieses Systems, wie z. B. die Weiterleitungslatenz auf der Kontroll- und Datenebene, die Häufigkeit der Interaktion zwischen der Kontroll- und Datenebene usw. In dieser Arbeit werden die im Rahmen des ersten Ziels gesammelten Messungen genutzt, um die Leistung von P4-basierten Systemen in zwei Schritten zu modellieren: (1.) Es wird eine Methode vorgeschlagen, um die Weiterleitungslatenz auf P4-Datenebenen als Funktion der konfigurierten P4-Programme zu schätzen; (2.) Ein feedback-orientiertes Warteschlangensystem wird verwendet, um die Leistung von P4-basierten Systemen zu modellieren, deren Weiterleitungslatenz auf der Datenebene variabel ist und mit der abgeleiteten Schätzmethode im ersten Modellierungsschritt quantifiziert wird.

Wenn programmierbare P4-Paketprozessoren in Cloud-Umgebungen eingesetzt werden sollen, bilden sie einen heterogenen Pool von Verarbeitungsgeräten verschiedener Typen mit unterschiedlichen Leistungsniveaus, Fähigkeiten und Einschränkungen. Diese Vielfalt an Verarbeitungsplattformen bietet einen zusätzlichen Freiheitsgrad bei der Auswahl der Plattform, die für eine bestimmte Arbeitslast mit bestimmten Anforderungen an die Servicequalität (QoS) am besten geeignet ist. Dies ist nur möglich, wenn eine solche Umgebung richtig verwaltet wird. Die Arbeit nutzt die Leistungsmodelle, die in der vorherigen Zielsetzung entwickelt wurden, um zwei leistungsbewusste Optimierungsprobleme für die Planung der Infrastrukturgrundlage von P4-basierten Cloud-Umgebungen und die Laufzeiteinbettung von Workload-Anforderungen unter Einhaltung von QoS-Anforderungen zu lösen.

Abschließend implementieren und analysieren wir zwei Anwendungen, um die Vorteile der Verwendung programmierbarer Datenebenen in Cloud-Umgebungen für zukünftige Netzwerke aufzuzeigen. In Anbetracht der Tatsache, dass Cloud-native Lösungen in zellularen Netzwerken vorherrschen, wählen wir die User Plane Function (UPF) als ersten untersuchten Anwendungsfall einer NF, die in Clouds laufen und die P4-Technologie nutzen kann. In dieser Arbeit wird eine Microservice-basierte UPF vorgeschlagen und implementiert, um die Nutzbarkeit der P4-basierten Cloud-Umgebung und des im vorherigen Ziel vorgeschlagenen Verwaltungsschemas zu demonstrieren. Die zweite Anwendung entschärft die fehlende Unterstützung der P4-Sprache für programmierbares Verkehrsmanagement, indem sie eine Lösung für die Implementierung virtueller Warteschlangen zur Erreichung dieses Ziels bietet. Die programmierbare Verkehrsverwaltungsanwendung dient als eigenständige Anwendung zur Anpassung der Rate und Verzögerung auf Slice-Basis, um die erforderlichen Service Level Agreements zu erfüllen, und sie ergänzt die UPF-Implementierung, da sie die Durchsetzung verschiedener erforderlicher QoS-Stufen ermöglicht.

Contents

1. Introduction

Our society is continuously moving towards digitization where the digital and physical worlds are increasingly mingling. Communication networks play a crucial role in transporting digitized information from one physical location to another. For example, the Internet is one huge realization of a network that connects the world carrying information between content providers and consumers. The needs of user applications in terms of connectivity dictate the requirements that should be satisfied by the underlying networks. For example, while surfing web browsers made the best effort connectivity good enough, video calling and streaming require higher bitrates, and applications such as Internet of Things (IoT)s and Cyber-Physical Systems (CPS) now have more stringent Quality of Service (QoS) requirements in terms of the expected reliability and latency. On one hand, these emerging applications are requiring networks to be more **performant** in terms of throughput, latency, etc. On the other hand, they are demanding **flexible** networks that can support new functionalities that range over the different layers of the networking stack such as load balancing, filtering, intrusion detection, etc. [52].

Legacy networking infrastructures are mainly based on purpose-built hardware devices or middleboxes that can deliver high processing performance but are rigid in terms of functionalities that they can execute. In many cases, the requirements from networks are evolving rapidly making the rigid legacy networking infrastructure an unsatisfactory solution because of the lengthy development cycles and the high upgrading costs. To cope with the agility in network development, software-based solutions such as Software-defined networking (SDN) and Network Function Virtualization (NFV) emerged and developed to a mature state nowadays.

SDN is a networking paradigm that makes networks more flexible and manageable compared to legacy infrastructures. It decouples the functionality of the control and data planes and realizes an interface between them while permitting programmability at the control plane. NFV is the concept of running Network Function (NF)s as software applications on Commercial off-the-shelf (COTS) servers instead of using purpose-built hardware middleboxes. NFV enables network operators to leverage the advantages of software virtualization and cloud computing techniques, which proved its effectiveness in the Information Technology (IT) domain, in the networking domain. Virtualized Network Function (VNF)s can be flexibly instantiated, configured, scaled up or down, migrated, and terminated over different distributed data centers. Additionally, NFV is considered an appealing solution from a business point of view, as scaling and upgrading NFs takes place in software rather than hardware.

However, a purely software-based solution using NFV has limitations in terms of performance. For instance, COTS servers cannot process packets at line rate for computationally-intensive NFs, especially for small packets [46]. Additionally, the strict and deterministic latency requirements of emerging applications, like augmented reality and industrial IoT, can hardly be met using pure software-based NFV solutions [46]. Therefore, different acceleration techniques have been investigated to enhance the packet processing performance of VNFs. These solutions vary from software-based acceleration techniques, which speed up the packet processing of COTS servers, like the Data Plane Development Kit (DPDK) framework [103], to hardware-based acceleration techniques, which offload part of the packet processing tasks to hardware accelerators [45]. A sweet spot between the latter two approaches are programmable packet processors such as SmartNICs and programmable switches. This technology possesses the processing performance of hardware-based solutions while still offering the flexibility of software-based solutions through programmability [46].

Furthermore, the flexibility provided by traditional SDN-based solutions is still limited by the packet processing operations permitted by the architecture of the pipeline at the forwarding plane. OpenFlow (OF) is widely adopted as the de facto architecture for SDN data planes, where the communication between the control and data planes takes place according to the messages defined in this protocol. It specifies a limited set of matching fields and actions that can be applied on the data plane. Accordingly, the control plane's programmability and control over the data plane are restricted by the matching fields and actions exposed by the OF Application Programming Interface (API). The concept of protocol-independent programmable packet processors extends the SDN paradigm by pushing the programmability further to the data plane besides the control plane, wherein the packet forwarding pipeline at the data plane can also be customized.

In this direction, P4 was proposed as a programming language for programmable packet processors [18]. P4 is becoming a de facto standard language for describing the forwarding behavior of packet processors. Although it adopts the match-action abstraction in describing the packet-forwarding behavior similar to the OF protocol, it is differentiated by being protocol-independent, meaning that it can support defining arbitrary protocols' matching fields and custom actions to be applied. This provides higher flexibility by enabling the definition of innovative P4-based NFs and solutions at the data plane. Note that we use NFs and P4 programs interchangeably in this thesis. Moreover, P4 is target-independent, meaning that it can be used to program different processing platforms such as software switches, programmable Application-Specific Integrated Circuit (ASIC) switches, Field-Programmable Gate Array (FPGA)-based SmartNICs, Network Processor Unit (NPU)-based SmartNICs, etc. It also supports field-reconfigurability so that the packet forwarding pipeline of the packet processor can be reconfigured even after deployment.

The integration of P4 into NFV cloud environments has huge potential in terms of flexibility and performance. However, this integration also raises a new plane of problems and challenges in terms of the design and management of such environments. The goal of this doctoral thesis is to measure and model the performance of P4 programmable packet processors towards conducting performance-aware management of NFV cloud environments with the P4 technology incorporated. The objectives of this thesis are summarized as follows.

First, we evaluate the performance of P4 programmable packet processors by conducting a comprehensive measurement campaign to understand the capabilities and limitations of different P4 devices. Since the data plane programmability is an intrinsic property of P4 devices, we focus on understanding the effect of the P4 data plane's complexity on the packet processing latency in different P4 devices.

The measurements collected out of the first objective are prerequisites for deriving generic analytical models for the forwarding performance of a P4-based system made up of control and data planes, which is the second targeted objective.

The third objective is to manage NFV cloud environments made up of heterogeneous P4 programmable substrate while supporting Service Function Chain (SFC) embedding with QoS requirements. The feature of supporting QoS requirements can only be achieved when the forwarding performance of different P4 packet processors becomes predictable, a target accomplished in the previous objective.

The fourth objective is to demonstrate the advantages of including data plane programmability in cloud environments. This is achieved by proposing the following two applications. As deploying cellular networks in cloud-native environments is becoming a viable solution [53], the first application studies the advantages of deploying the User Plane Function (UPF) of the 5G Core networks into cloud environments enhanced with P4-based hardware accelerators. In this direction, we design and implement a novel microservice-based UPF using P4. The deployment of this design leverages the previously developed optimal management scheme that integrates hardware accelerators into NFV cloud environments. The second application is a programmable traffic management solution that enables customizing the traffic characteristics (rate and delay) of different network slices using data plane programmability. The second application is a stand-alone application that supports network slicing at the data plane, and it also complements the first application, i.e., the microservice-based UPF, by enabling QoS enforcement at the user plane of the 5G Core networks.

The research challenges accompanied by these goals are discussed in Section 1.1, while Section 1.2 presents the contributions achieved while addressing these challenges, and finally, Section 1.3 illustrates the outline of the remainder of the thesis.

1.1. Research Challenges

The integration of P4 programmable packet processors into the NFV cloud environments enhances the processing performance without sacrificing the flexibility attained via programmability. However, this integration also raises a new plane of problems and challenges in terms of the design and management of such environments. This section summarizes the main research challenges tackled in this thesis.

Measuring the Performance of P4 Programmable Packet Processors. After the introduction of the P4 programming language, many networking hardware vendors released packet processors that support P4 programmability. Although these devices exhibit a high degree of flexibility in programming their packet forwarding pipelines, the forwarding performance of these devices has not been entirely evaluated. Keeping in mind that performance is a crucial factor in determining the success or failure of any emerging technology for a given use case scenario, we focus in this part of the thesis on fairly comparing the capabilities and limitations of different state-of-the-art P4 programmable devices.

Different factors influence the performance of P4 programmable packet processors. To start with, P4 programmable packet processors belong to various classes of processing platforms such as Central Processing Unit (CPU), FPGA, NPU, ASIC, etc. Certainly, the evaluation methodology should be applicable to all these platforms, while still guaranteeing fairness in the comparison.

On the other hand, the capability of programming the forwarding pipeline adds a new factor that can cause variation in the forwarding latency of P4 programmable packet processors. This factor is the complexity of the P4 program which can result in a more complex packet processing pipeline of operations. Moreover, the fact that the P4 language is protocol-independent means that there are endless possibilities of P4 data planes that can be configured on these P4 devices. Accordingly, the impact of the pipeline complexity on the forwarding latency should be understood while considering the possibility that any arbitrary P4 program can run on these P4 packet processors.

Finally, the evaluation should be thorough where the different components that build a P4-based system ranging from the controller to the data plane should be analyzed. The lack of published works in this regard raises another challenge that demands in some cases implementing novel tools for conducting this evaluation, as in the case when benchmarking P4-based SDN controllers.

Modeling the Performance of P4 Programmable Packet Processors. While measurements give exact knowledge about the performance of devices under test, it is infeasible to cover the complete space of measurement possibilities. Alternatively, analytical models that build on top of measurement results play the role of abstracting and predicting the real performance of these devices more generically.

Different factors affect the performance of the P4 devices such as the type of processing platform, the running P4 program, the level of involvement of the control plane, the performance of the control plane, etc. The analytical performance models should cover the impact of these different factors, wherein these factors can be used for parametrizing the targeted models. At the same time, the models should be as accurate as possible in capturing the real performance of P4 devices.

Performance-aware Management of P4-based Cloud Environments. When P4 programmable packet processors are used in cloud environments, they create a heterogeneous pool of processing devices. This diversity in processing platforms gives an additional degree of freedom in selecting the platform that best hosts a certain workload with given QoS requirements. This is only possible if proper management of such an environment is realized.

On one hand, the heterogeneity of available packet processors dictates creating profiles that summarize the limitations and capabilities of the processing platforms. On the other hand, the requirements of the SFC workloads to be embedded in terms of demanded functionalities and required QoS levels should be taken into consideration. The optimal embedding of SFCs into a heterogeneous P4-based cloud environment should satisfy these constraints.

To support the NF embedding with QoS requirements, the optimization problem must have a clear expectation about the performance of the underlying P4 packet processors when running a given NF.

Finally, as the ordinary VNF embedding problem is already of Nondeterministic Polynomial Time (NP) complexity, the new version with performance awareness is even more complex. Thus, a greedy solution for this problem should be offered to guarantee that solving the problem and updating the embedding solution is reached in a reasonable execution time.

Applications Leveraging P4-based Cloud Environments. As cloud-native solutions are prevailing in cellular networks, we select the UPF as a candidate NF that can run in clouds and make use of the P4 technology. Being assigned a wide range of data plane processing tasks, the UPF represents an interesting candidate data plane function to be studied, which can leverage the previously developed optimal management framework. Moreover, designing a microservice-based version of the UPF for breaking its complexity at the user plane, similar to the approach adopted at the control plane in 5G networks, is a promising approach for enhancing the flexibility and performance of Beyond 5G (B5G) cellular networks. However, separating such a big monolithic function based on the microservice architecture design principles is not an easy task, not to mention its implementation.

On the other hand, given that the P4 language does not support programmable traffic managers, the development of a customizable traffic management solution to mitigate this issue by enabling the enforcement of QoS requirements at the data plane presents another challenging exercise.

1.2. Contributions

This section summarizes the major contributions of this doctoral thesis and the relation between these contributions toward the main goal of integrating P4 programmable packet processors into NFV cloud-based environments. Fig. 1.1 demonstrates the structure of the thesis in the context of studied research fields and applied methodologies. Each of the four major investigated research fields and contributions of this thesis is included in Chapters 3 to 6, respectively. Chapter 3 contributes a comprehensive understanding of the performance of the different components that build a P4-based system through conducting extensive measurements. Chapter 4 presents analytical models to characterize the forwarding performance of P4 packet processors. Chapter 5 contributes to the optimal management of NFV cloud environments made up of P4 programmable devices while leveraging the knowledge regarding the performance of these devices to satisfy QoS requirements. Chapter 6 Provides Proof of Concept (PoC) implementations of microservice-based UPF and programmable traffic manager solutions to showcase the advantages of adopting P4 programmable packet processors in cloud environments. In the following, we illustrate the contributions achieved in each chapter and the methods applied for this purpose.

It is of paramount importance to understand the performance of different P4 programmable packet processors to assess their suitability for different use case scenarios. The **first** major contribution in Chapter 3 targets this purpose, where we perform thorough measurements of the different characteristics related to these P4 devices. First, we propose an experimental methodology to reveal the latency of executing atomic P4 operations on arbitrary P4 targets. The evaluation results revealed the influential P4 operations on the data plane forwarding latency besides revealing the capabilities and limitations of different state-of-the-art P4 targets. Then, the performance of these P4 targets is inspected when dealing with scenarios where a scaled-up number of traffic flows needs to be processed. The control agent lying in the P4 target, which interacts with the controller on one side and with the data plane on the other side, is also examined where we measure its response to control plane commands in updating the status of the forwarding plane. Finally, we fill a research gap in the literature by implementing a novel tool for benchmarking the performance of SDN controllers that support controlling P4 targets using the P4Runtime (P4RT) framework [92]. We use this tool to understand the performance of the control plane in a P4-based system,

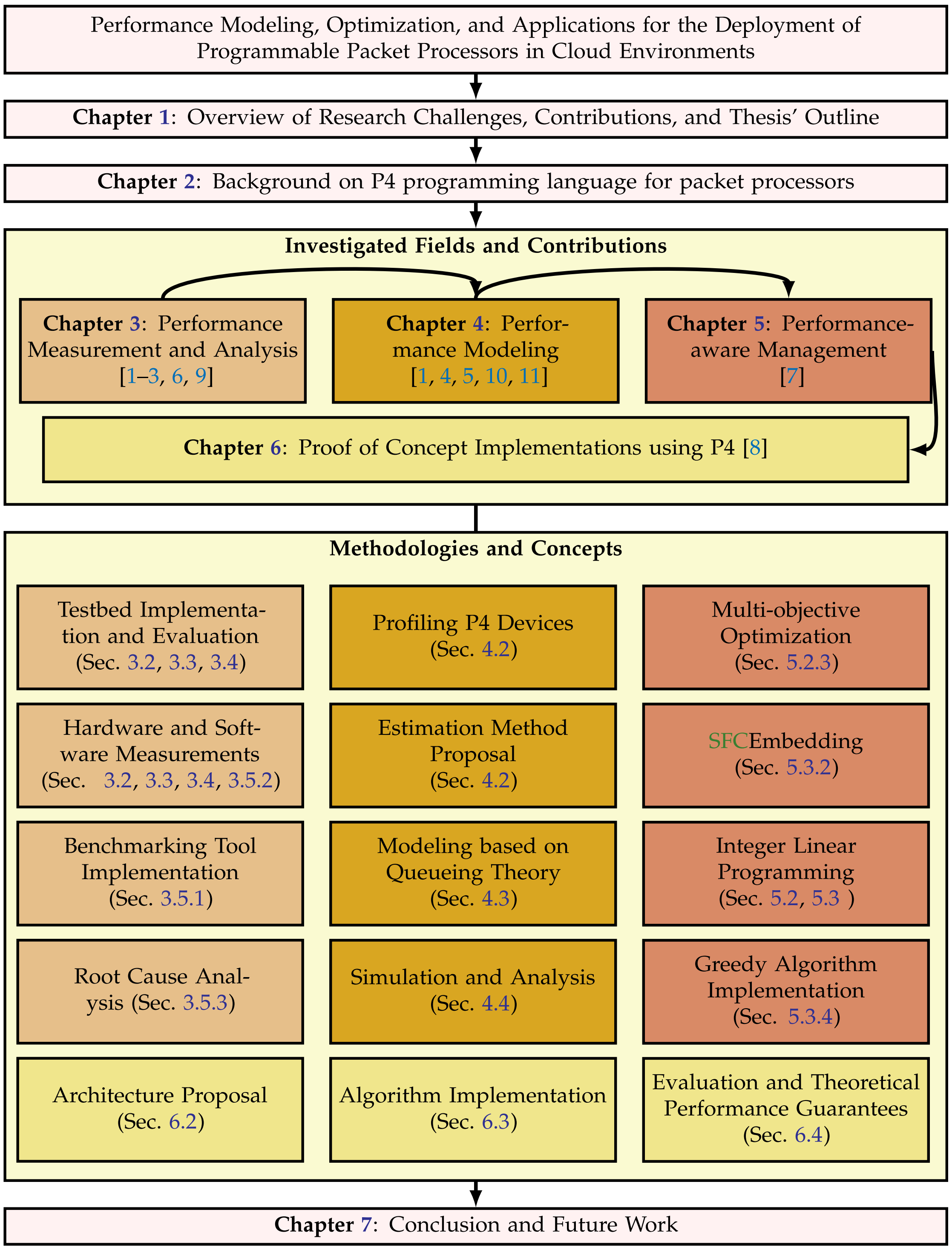

Performance Modeling, Optimization, and Applications for the Deployment of Programmable Packet Processors in Cloud Environments
Chapter 1: Overview of Research Challenges, Contributions, and Thesis' Outline
Chapter 2: Background on P4 programming language for packet processors
Investigated Fields and Contributions
Chapter 3: Performance Measurement and Analysis [1–3, 6, 9]
Chapter 4: Performance Modeling [1, 4, 5, 10, 11]
Chapter 5: Performance-aware Management [7]
Chapter 6: Proof of Concept Implementations using P4 [8]
Methodologies and Concepts
Testbed Implementation and Evaluation (Sec. 3.2, 3.3, 3.4)
Profiling P4 Devices (Sec. 4.2)
Multi-objective Optimization (Sec. 5.2.3)
Hardware and Software Measurements (Sec. 3.2, 3.3, 3.4, 3.5.2)
Estimation Method Proposal (Sec. 4.2)
SFCEmbedding (Sec. 5.3.2)
Benchmarking Tool Implementation (Sec. 3.5.1)
Modeling based on Queueing Theory (Sec. 4.3)
Integer Linear Programming (Sec. 5.2, 5.3)
Root Cause Analysis (Sec. 3.5.3)
Simulation and Analysis (Sec. 4.4)
Greedy Algorithm Implementation (Sec. 5.3.4)
Architecture Proposal (Sec. 6.2)
Algorithm Implementation (Sec. 6.3)
Evaluation and Theoretical Performance Guarantees (Sec. 6.4)
Chapter 7: Conclusion and Future Work

Figure 1.1.: Outline showing the flow of the thesis and the relation between the different chapters. The measurement results derived from Chapter 3 are used by Chapter 4 to derive performance models. Chapter 5 uses these performance models to optimize the management of NFV cloud environments. Chapter 6 demonstrates the advantages of the integration of P4 devices into NFV cloud environments. Each contribution is mapped to its corresponding applied methods by using the same color code.

where we evaluate the performance of the ONOS controller [109] when running in P4RT mode versus the case when running in OF mode. Furthermore, we perform root cause analysis using this tool to identify an implementation bottleneck in the ONOS controller, and then we propose a code patch for mitigating this bottleneck.

The **second** major contribution in Chapter 4 targets deriving analytical models for the forwarding latency of P4-based systems. These models build on top of the previously collected measurements to provide a more generic way of characterizing the performance of P4-based systems. First, we build a model that can predict the packet forwarding latency when running arbitrary P4 programs on different P4 targets. Then, a simple queueing theory-based model is proposed based on meaningful parameters of the P4-based system. This model is then further refined to better capture the real performance of P4 targets. The different proposed models are validated through simulations that vary a wide range of parameters and analyze their impact on the packet's sojourn time in the system. Finally, the developed models are used to derive constraints on the permissible traffic volume that can be handled by P4-based systems.

Following the thorough understanding of the performance of different state-of-the-art (SOTA) P4 targets and specifically their forwarding latency, the **third** major contribution in Chapter 5 targets leveraging this performance knowledge and models to optimally manage cloud environments built from P4 targets. A cloud environment can leverage P4 targets such as SmartNICs or programmable switches to enhance its processing capabilities without sacrificing programmability. The usage of these devices creates a pool of heterogeneous processing resources with distinct performance levels and capabilities. On the other hand, NFs described as P4 programs have different requirements in terms of demanded acceleration functions and desired QoS levels that need to be satisfied when embedding these functions into the P4-enhanced infrastructure. Although the SFC embedding problem is widely studied in the literature, this problem is still not addressed when the embedding is performed into a heterogeneous pool of P4 packet processors and when the performance awareness is a prerequisite for satisfying QoS requirements. The first studied optimization problem targets planning the infrastructure of P4-based cloud environments by selecting the optimal set of P4 packet processors that can handle an expected incoming traffic workload. The problem formulation searches for the best placement of a given set of NFs workload by mapping the requirements of these NFs to the capabilities and the performance of the different available P4 packet processors. The objective function targets maximizing the performance of the system while minimizing capital expenditure costs. The second optimization problem targets provisioning the optimal embedding of SFCs into P4-based cloud environments at runtime. The problem formulation searches for the optimal placement of SFCs into P4 packet processors while satisfying the functional and QoS requirements of these SFCs based on the previously acquired knowledge related to the performance and capabilities of the different available P4 packet processors. Furthermore, a greedy solution is designed and implemented to solve this problem faster. The two optimization problems are evaluated based on different use case scenarios and the results are analyzed to derive interesting insights.

The **fourth** major contribution in Chapter 6 targets proposing two interesting solutions that showcase the advantages of integrating P4 packet processors into cloud environments. The first selected use case is related to the UPF of 5G Core networks. We implement the UPF in a modular way using P4. Following the modular implementation of UPF in P4, we propose a microservice-based design for the UPF that could be used in B5G networks. We argue how such a microservice P4-based UPF can be managed and orchestrated using the previously proposed performance-aware management scheme for P4-based cloud environments. The second application is a P4-based programmable traffic manager solution that can be used to control the rate and delay of different network slices at the data plane. This application complements the UPF implementation as it enables enforcing different QoS levels for the different network slices supported in the 5G Core network.

1.3. Outline

The outline of the thesis is shown in Fig. 1.1. This chapter introduces the objective of the thesis, the encountered research challenges, and the contributions attained while addressing these challenges.

Chapter 2 provides all relevant background information on P4 programmability and its relation with SDN and NFV. It also provides a detailed explanation of the syntax of the P4 language, the interaction with the control plane, the compilation process, and the available state-of-the-art P4 targets.

Chapter 3 includes the first major contribution of the thesis, where experiments and measurements are conducted for understanding the performance of P4 packet processors. First, the impact of different P4 atomic constructs on packet forwarding latency is explored. Next, the device's reaction to scaling up the incoming number of flows is studied. Then, we study the device's responsiveness to control plane commands. Finally, we evaluate the performance of the control plane when dealing with programmable P4 data planes.

Chapter 4 elaborates on the second major contribution of the thesis related to deriving theoretical models for the performance of P4 packet processors. These models build on top of the previously derived measurements. The first model characterizes the packet forwarding latency at the data plane as a function of the complexity of the loaded P4 program. The second model is a simple model that uses queueing theory to abstract the full P4-based system including both the control and data planes. The third model refines the second simple model by considering the distributions of the service times at the data plane and the control plane rather than only considering the first moments.

Chapter 5 illustrates the third major contribution of the thesis, where it uses the previously derived performance models to optimally manage NFV cloud environments made up of P4 programmable packet processors. Two optimization problems are formulated and evaluated, where each problem deals with a different management stage. The first problem is concerned with the offline optimal planning for selecting and building a network's substrate, while the second problem is concerned with optimizing the runtime management of an already given built-up network.

Chapter 6 includes the fourth major contribution, which presents applications that leverage the integration of P4 devices in cloud environments. The first implemented application is a microservice-based UPF for B5G cellular networks. The second application is a programmable traffic management solution for customizing traffic characteristics (rate and delay) of different network slices. The second application is a stand-alone application and at the same time a complementary application for the UPF as it enables it to enforce different QoS levels based on supported slice requirements.

Finally, Chapter 7 concludes this thesis by summarizing the important research contributions. It also discusses the open possibilities for future research directions.

2. Background on P4 Programmable Packet Processors

Communication networks are meant to transport information from one physical location to another. This information is transformed into bits and bundled into packets. Packet processors are devices used to process these packets to guarantee their proper forwarding and handling. These packet processors include devices such as switches, routers, or Network Interface Card (NIC)s for packet forwarding, and middleboxes such as firewalls, load balancers, etc. for packet processing.

The rapid change in the tasks and capabilities required to be delivered by currently deployed communication networks made the upgrading cost of these networks very expensive. For this reason, packet processing devices are moving towards being more programmable, which enables reusing the same device for different purposes over time. Accordingly, we can observe technologies that support programmable packet processing, which permits reconfiguring the packet processing pipeline and the operations that packets will undergo when traversing this pipeline. There are currently different technologies that allow customizing the behavior of programmable packet processors. These include using micro-c language to program SmartNICs, using Protocol-oblivious Forwarding (PoF) [17] to program packet processors, or using extended Berkeley Packet Filter (eBPF) [105] to capture and manipulate some packets in the Linux kernel.

One of the most promising technologies related to programmable packet processors is Programming Protocol-Independent Packet Processors (P4) [18]. P4 is a domain-specific language for programming packet processors. It provides high flexibility in defining the packet processing behavior on a wide range of different processing platforms. These features made it appealing to both researchers and device manufacturers. For this reason, we will focus the rest of the thesis on the deployment of P4 programmable packet processors in cloud environments. We select the P4 technology for programming packet processors because of its expressiveness in describing the packet processing behavior of devices, its independence from any predefined protocols, and its target independence that permits programming any packet processor type. General background and definitions of P4 programming language and its relation with SDN and NFV are provided in Section 2.1. A detailed explanation of P4 programmability is provided in Section 2.2. The content of this chapter partly relates to background sections of publications [1,2,6–8]

2.1. What is P4?

In this section, we explain the P4 programing language and its relation with the other two well-known technologies SDN and NFV that deal with network programmability. The elaboration on this relationship is important to understand how P4 packet processors complement SDN and NFV in making networks more flexible, while still providing the opportunity to adopt high-performant hardware accelerators in cloud environments.

2.1.1. P4 Programming Language

P4 is a domain-specific language for programming packet processors. Its syntax allows programming packet processors with simple programs compared to other languages. For example, [14] shows that writing a program to do IPv4 routing on FPGA requires only 266 lines of code in P4, while it needs around 3889 lines of code in Verilog. P4 has three design objectives [18], which makes it appealing in many use cases. These are:

- Protocol-Independence: This means that the language is independent of predefined networking headers and protocols such as Ethernet or IPv4. In a P4 program, all headers and actions are defined from scratch. This allows reimplementing all known network functions like Layer 3 Forwarding (L3Fwd), or more interestingly, it enables the development of innovative protocols and network functions for solving emerging problems.

- Target-Independence: This means that the language can be used to program targets belonging to different processing platforms such as software switch or FPGA or ASIC switch, etc.

- Field-Reconfigurablility: This means that it is possible to reconfigure or reprogram a P4 target even after deployment.

P4 is used to develop a wide range of applications that serve different purposes. These applications targeted monitoring networks with In-Band Network Telemetry as in [48], active management of switches' queues and congestion control as in [8,47], and executing middle-box functions such as Load balancer [49], cellular networks'-related tasks [50], etc.

2.1.2. P4 and SDN

SDN is a networking paradigm that makes networks more flexible and manageable compared to legacy infrastructures. It decouples the functionality of the control and data planes and realizes a well-defined programming interface between them [19]. The OF protocol [106] is introduced as a key component for enabling SDN because it standardizes a way for the control plane to interact with the switch via the southbound interface. It abstracts the packet processing at the data plane into a series of match-action tables. It defines a set of messages that could be communicated between the control and data plane such as installing rules into match-action tables, fetching counter statistics, etc. The OF messages are based on predefined header fields that packets can match on such as IPv4 destination address, and a set of actions that can be executed upon matching such as forwarding to a specific port or dropping the packet. OF switches should have a packet processing pipeline compatible with the OF architecture to properly react to messages received from the controller.

However, this approach to SDN using OF switches was running into some issues. As data plane protocols change and evolve, the protocol-dependent OF standard used as the southbound interface between the controller and the switch had to be updated. For example, while the first version *1.0.0* of the OF protocol had 12 header fields that could be matched on, this number largely increased in the following versions [117]. As a result, OF has already grown significantly and will continue to do so as data plane protocols evolve. This requires a continuous changing of the OF standard, which takes a long time and makes the solution impractical in many situations.

The solution to this problem was to move programmability to the data plane, where network operators can have top-down control over the data plane's behavior besides the control plane. P4 programmable devices present a proper solution for this problem since these devices are protocol-independent and can be reprogrammed to support the ever-evolving data plane protocols. Note that it is possible to configure a P4 data path similar to that of the OF architecture, making the OF solution one possible realization of the P4 solutions. Programmable packet processors can be alternatively called programmable data planes, where the latter terminology is more in line with the SDN nomenclature.

2.1.3. P4 and NFV

The rapidly evolving innovations in the networking domain demand upgrading the network infrastructure continuously, making the lifecycle of any technology short. This results in reducing the return on investment of deploying new services and solutions because of the high upgrading costs. NFV was proposed to solve this issue by replacing function-dedicated equipment with COTS servers for leveraging the advantages of virtualization which has already proven its effectiveness in the IT domain. This involves implementing NFs in software to provide higher flexibility in managing and upgrading these virtualized NFs [107].

However, this NFV approach of using software running on COTS servers instead of purpose-built hardware can cause severe performance bottlenecks in terms of latency and throughput. This is the reason for developing many frameworks, such as the DPDK framework [103], that target enhancing the performance of NFs execution on COTS servers and CPU-based devices [45].

One adopted solution for this performance issue is to use hardware accelerators besides the COTS servers to boost the packet processing performance in an NFV architecture. P4 programmable packet processors provide a powerful solution in this scope as these devices, especially hardware-based ones, can achieve a very high packet processing performance compared to COTS servers, while still allowing for programmability and reconfigurability [46].

2.2. How does P4 work?

In this section, we explain the different components constituting a P4-based system, and how these components interact with each other.

2.2.1. P4 Data Plane: Language Syntax and Architecture

The latest release of P4, i.e., $P4_{16}$, separates the language syntax from the architecture of the targeted packet processor. Packet processors to be programmed are called P4 targets, and these can have different hardware implementations. Each P4 target is programmed based on the provided P4 architecture, where the sequence of available processing stages and the programmable ones out of these stages are specified. The architecture provides an interface to program a target, where it can also define APIs for extern functions that can be used to access built-in accelerators such as encryption engines. The V1model [118], shown in Fig. 2.1, is a commonly used P4 architecture that has the following stages:

- **Data Structures Definition:** First, headers are defined as structures containing different fields with different bit string sizes. It is also possible to define metadata to be associated with the packet when it is processed at different stages.
- **Parser:** The parser is the first programmable processing block after receiving a packet from an ingress port. It is used to define which headers are to be extracted from the received packet. It is programmed as a Finite State Machine (FSM) to define the states and the transitions while parsing a given packet.
- **Payload Buffer:** The payload of the packet is stored in the payload buffer, as the main processing is performed on the headers of the packet.

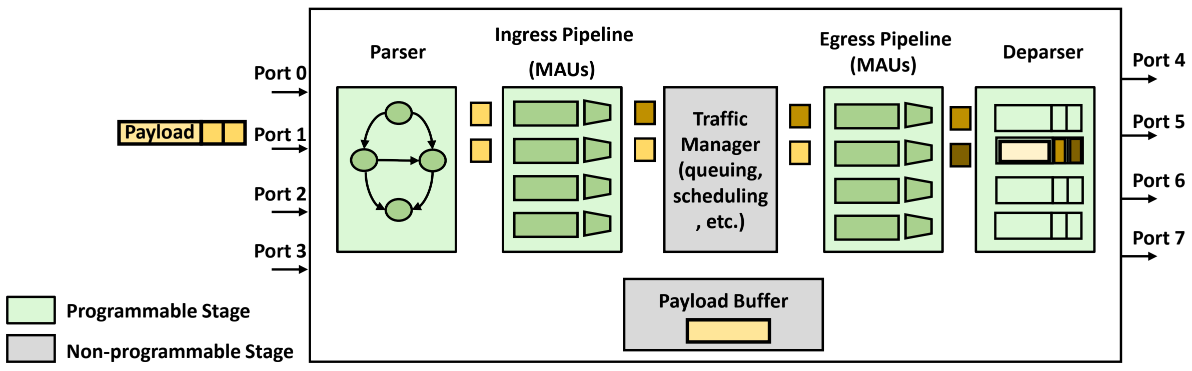


Figure 2.1.: Sample P4 data plane following V1model architecture [118]. The programmable blocks such as the parser, ingress and egress pipelines, and deparser are colored in green, while the non-programmable blocks such as the traffic manager and payload buffer are colored in grey. Headers are extracted from the packet in the parser stage. These extracted headers are manipulated in ingress and egress pipelines. Finally, the packet is reconstructed with the manipulated headers at the deparser stage before leaving the P4 data plane on the designated egress port.

- **Ingress Pipeline**: In this stage, the operations that transform the extracted headers are defined. The ingress stage is programmed as a sequence of instructions, which can trigger the activation of tables or Match-Action Unit (MAU)s. The MAU is mainly defined by specifying a key and a list of possible actions. A key is made up of a list of header fields or metadata on which packets can match. The matching type can be exact, Longest Prefix Match (LPM), or ternary. If a packet matches the key of a table entry, an action listed in the action list can be invoked. Actions are defined as function calls that can execute different operations on the packet. These operations include adding or removing headers from the packet, modifying fields of a header, modifying metadata related to the packet, or executing arithmetic and binary operations.

- **Traffic Manager**: After ingress processing, packets move to the traffic manager. The traffic manager takes care of packet queueing, replication, and scheduling. It organizes the flow of packets, especially when more than one packet needs to leave on the same egress port, or when packets need to be replicated. Currently, the traffic manager is non-programmable in all P4 targets.

- **Egress Pipeline**: The egress control block resembles a similar role to the ingress processing block except that it takes place after the traffic manager. This location enables accessing extra information related to the queueing process that took place in the traffic manager. For example, it is possible to extract the length of the queue and the time spent by the packet in the queue.

- **Deparser**: In this last stage, the packet is reassembled again by adding the manipulated headers back to it. After deparsing, the packet leaves the P4 target on the designated egress port.

The language also defines counters to collect statistics, meters to rate limit flows, and registers to store information to be shared across packets for stateful applications.

2.2.2. P4 Control Plane

In the following, we will describe the P4Runtime framework for controlling P4-based data planes, then we elaborate on how the Open Network Operating System (ONOS) controller supports and implements this framework.

P4Runtime

The P4RT framework [92] is the de facto runtime API for controlling P4 programmable data planes. It was first released in 2019 by the P4 language consortium. P4RT is designed to be:

- Target-independent, i.e., it can control data planes belonging to different types of processing platforms.
- Protocol-independent, i.e., new data plane protocols can be defined.
- Pipeline-independent, i.e., it can control different P4-based data paths or pipelines.

The API is built using Google's Protobuf [93], which is used for serializing structured data across languages and platforms. In a *p4runtime.proto* file [94], the remote procedure calls and the messages that could be communicated between the P4 data plane and the SDN controller are defined.

The client and server endpoints of a P4RT connection are implemented using Google Remote Procedure Call (gRPC) [95], which can be used to produce code in a variety of programming languages automatically, provide security methods, and enable bi-directional data streams. The server is located in the P4 device, while the client runs in the SDN controller. P4RT also allows pushing new P4 programs to the data plane during runtime to reconfigure its data path.

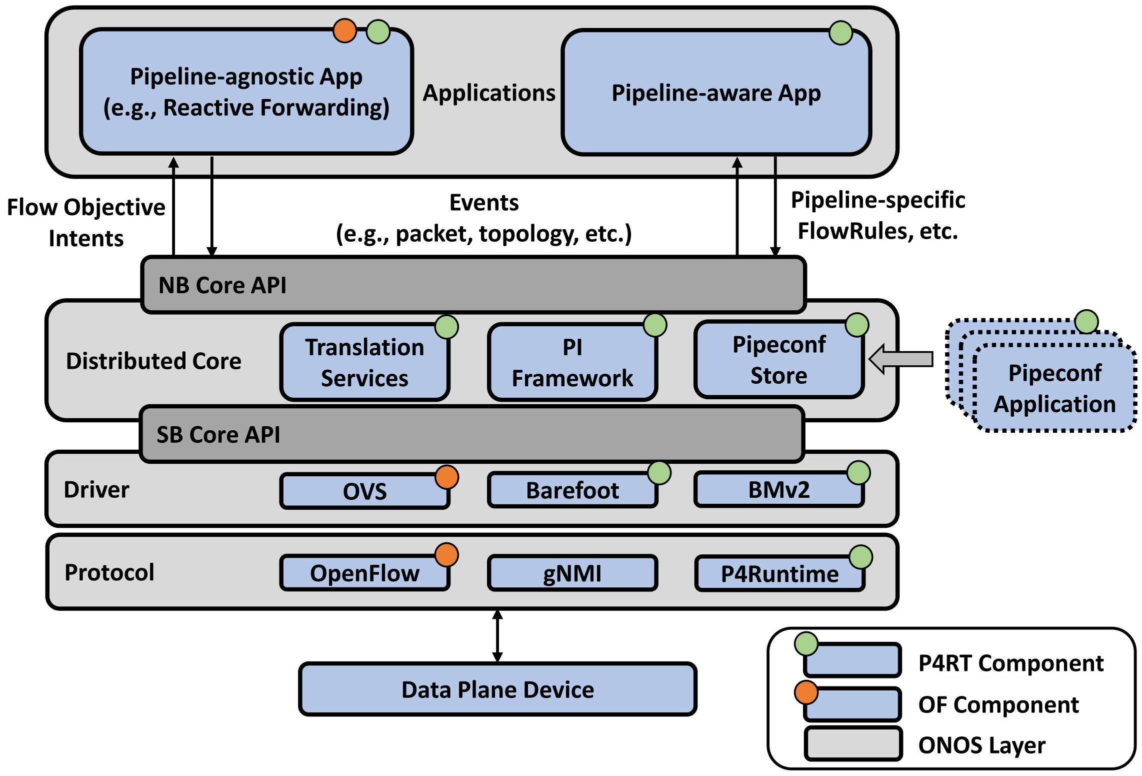


Figure 2.2.: Architecture of ONOS controller showing the different constituting layers. P4RT-related functional blocks are marked with green circles, while OF-related ones are marked with orange circles.

ONOS Controller

ONOS [28] is one of the few available SDN controllers nowadays that support controlling P4-based data planes using P4RT framework. This controller was first developed to support controlling OF-based switches. Then, it was extended to also support controlling P4-programmable data planes, wherein the forwarding data path can be reconfigurable.

The architecture of the ONOS controller, with the different constituting layers, is shown in Fig. 2.2. This figure distinguishes between P4RT and OF-based functional blocks. The role of the different layers of ONOS is described in the following:

- **Applications:** This is the top layer in ONOS architecture. The applications developed for controlling the networking behavior resides in this layer. These applications could be pipeline-agnostic, wherein the OF-based applications, such as *Reactive Forwarding*, are reused. Alternatively, these applications could be of the type Pipeline-aware, which means that they are specialized for controlling distinct customized P4 pipelines at the data plane.

- **Distributed Core:** This central layer contains the functional blocks that perform the main logic in ONOS. It communicates with the network functions running in the application layer via the NorthBound API, and with the lower layers via the SouthBound API. It contains the Pipeconf Store function, which stores and packages the files relevant to a given P4 pipeline to be controlled by ONOS. Additionally, it runs the Translation Services function, which translates the Protocol-Dependent entities and messages into Protocol-Independent (PI) representations to be compatible with P4 data planes.
- **Driver/Provider:** This layer contains the drivers of the different families of devices in the data plane to be controlled by ONOS. The driver of a device provides an interface for interacting with it.
- **Protocol:** This layer contains implementation of SouthBound protocols like OF and P4RT. ONOS encodes/decodes messages and sends/receives packets to the data plane based on these protocols.

2.2.3. P4 Compilation and Workflow

A P4 program is compiled in two steps. First, the standard open-source front-end compiler P4 Compiler (P4C) [96] is used to compile a given P4 program to generate an intermediate representation. It also generates a P4Info.json file, which contains information describing the tables and other relevant information about the compiled P4 pipeline needed for runtime control. This file acts as a 'contract' between the control plane and the data plane, where it is used by the controller to make API calls on top of defined tables and external instances. The second compilation step is done by the back-end compiler provided by the vendor of the P4 target. This back-end compiler generates the target configuration file, which is used to configure the data path of the P4 device. For example, this configuration file can be a bitstream file in the case of hardware P4 targets or a JavaScript Object Notation (JSON) configuration file for software P4 targets. Fig. 2.3 illustrates the compilation process of a P4 program as well as the runtime control of a P4 data plane.

2.2.4. P4 Targets

There are currently different P4 targets belonging to different processing platforms that support P4 programmability. Software P4 targets are mainly based on a compiler that translates P4 programs to executable software switches, while hardware targets should support this programmability at the silicon level. In the following, we will elaborate on some of the state-of-the-art P4 targets, whose properties are summarized in Table 2.1.

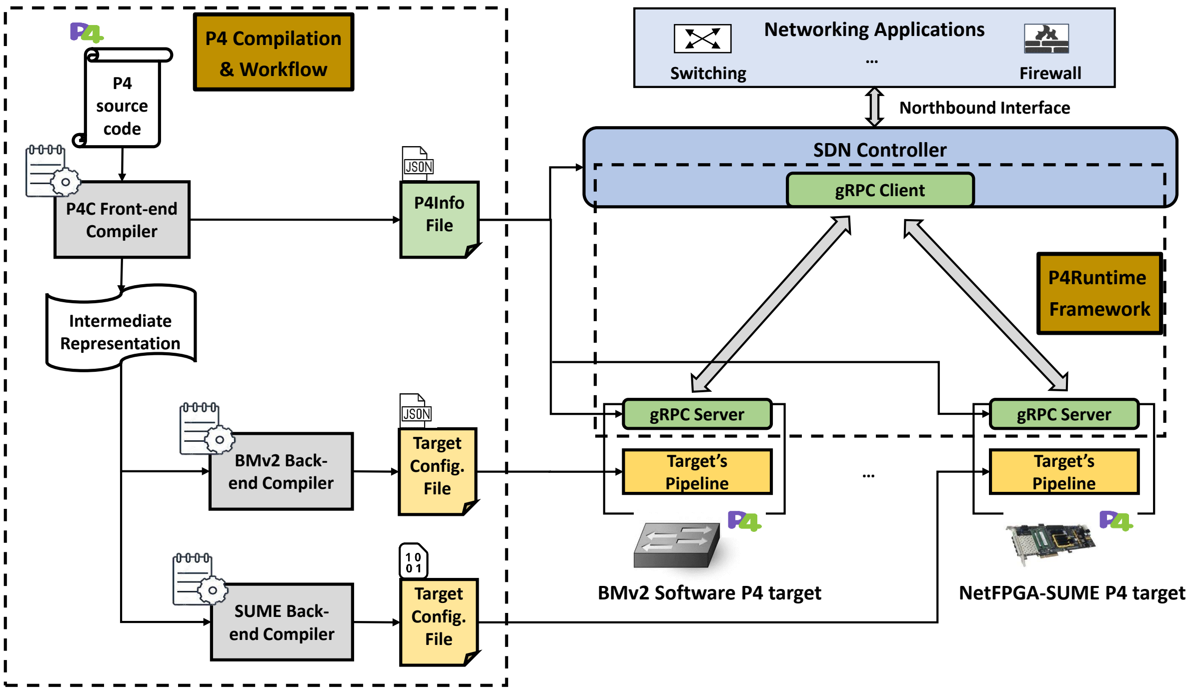


Figure 2.3.: The workflow for compiling a P4 program to be deployed in an SDN architecture with P4 programmable data planes is shown. While the P4Info file is necessary for the P4Runtime framework to take care of the communication between the data plane and control plane, the target configuration file reconfigures the packet processing pipeline of the two different P4 targets.

Behavioral Model (BMv2): The BMv2 [97] software switch is an open-source P4 software switch, provided by the P4 language consortium as a prototype implementation of software P4 targets. It is meant for prototyping P4 implementations, especially since it can be easily integrated with the mininet [98] emulator. It can process packets at a low processing rate compared to other P4 targets, reaching around 1 Gbps.

PISCES Software Switch: Shahbaz et al. extended the OF-based Open vSwitch (OvS) to allow packet processing behavior to be customized using P4. The new software switch is named PISCES [12], and it includes a compiler for analyzing P4$_{14}$ code and optimizing forwarding performance. While PISCES' performance is comparable to that of OvS, it takes 40 times fewer P4 lines of code to describe functions when compared to applying equivalent changes to OvS source code. PISCES was subjected to several optimizations to improve its processing performance until it matched that of OvS.

t4p4s DPDK-based Software Switch: DPDK [103] is an open-source framework for accelerating packet processing on different processors. It employs various methods to accelerate packet processing on software switches. First, all flows' incoming packets are distributed across different CPU cores reserved for the switch. Second, the data pipeline between the interfaces and the processing threads is entirely in userspace, eliminating the need for costly packet copying between user and kernel areas. Third, it processes packets in batches to make better use of the CPU's cache. t4p4s [16] is a compiler that converts P4 code into a target-independent C core program, which can then be used to execute the designed P4 datapath on top of the DPDK framework. On average, the compilation procedure takes a few minutes. When Non-Uniform Memory Access (NUMA) mode is enabled, two CPU cores (one as a master and one as a slave) are reserved for packet processing by default. t4p4s also employs the V1model architecture.

Agilio CX SmartNIC: Agilio CX 2x10GbE is a SmartNIC from Netronome [101]. It is an NPU with tens of purpose-built multi-threaded cores that enable high parallelism. The device's packet processing capability is furthermore boosted by hierarchical transactional memory and built-in accelerators. This SmartNIC adopts the V1model as a P4 architecture. Its back-end compiler generates a C implementation of the datapath, which is then used to create the firmware for the SmartNIC. Building and loading a firmware to the SmartNIC only takes a few minutes [100].

NetFPGA-SUME : FPGAs are a viable solution for meeting the requirements of low-latency and high-throughput concurrent packet processing while retaining programmability. By leveraging the P4→NetFPGA workflow [14], NetFPGA-SUME [15] allows easier programmability on FPGAs. An FPGA's hardware resources are primarily represented by the number of LookUp Tables (LUTs) and on-chip memories (Block Random Access Memory (RAM)s). Memory resources are used for various types of matching, including Ternary Content Addressable Memory (TCAM) for ternary matches and Static Random Access Memory (SRAM) for hash-based exact matches. The datapath is implemented as an Register-Transfer Level (RTL) implementation by the back-end compiler, which is then synthesized to a bitstream to program the FPGA chip. The entire procedure takes about an hour. NetFPGA's P4 architecture is *SimpleSumeSwitch* with only an ingress pipeline, unlike the V1model which has two pipelines abstracted as ingress and egress stages.

Tofino Chip: The Tofino chip from Intel is the first ASIC P4 programmable target. It follows the Protocol-Independent Switch Architecture (PISA) that allows adjusting protocols in software. It can process packets at a very high throughput reaching up to 12.8 Tbps [102].

In general, these P4 targets span a wide range of processing platforms. The software-based targets are mainly based on compilers that translate P4 programs to programs that run as software instances on servers with CPUs. The V1model architecture dominates for software-based P4 targets, while hardware-based targets often adopt special

Table 2.1.: A summary of the properties of different P4 targets. (-) is used when information is not available.

P4 Device	**Type**	**Processing Platform**	**P4 Architecture**	**P4 version**	**Vendor**
BMv2	SW	CPU	V1model	$P4_{16}$	Open Source
PISCES	SW/OvS	CPU	-	$P4_{14}$	Open Source
t4p4s	SW/DPDK	CPU	V1model	$P4_{16}$	Open Source
Agilio CX	HW SmartNIC	NPU	V1model	$P4_{16}$	Netronome
NetFPGA	HW SmartNIC	FPGA	SimpleSumeSwitch	$P4_{16}$	Xilinix
Tofino	HW Switch	ASIC	PISA	$P4_{16}$	Intel

architectures that are compatible with the device's silicon design. The processing performance of these targets widely varies based on the characteristics of the hosting processing platform.

2.3. Summary

In this chapter, we introduced P4 programmability and its relation to the other two technologies driving network flexibility, i.e., SDN and NFV. Then, we described the different components that build a P4-based system. We elaborated on the language constructs used for configuring the forwarding pipeline of the data plane, the P4RT framework for runtime control of P4 devices, the overall compilation process and workflow for deploying P4 targets, and the various available P4 targets.

This background information is important for thoroughly understanding the motivation behind using P4 programmability and its building blocks. This is required as the following chapters of the thesis are centered around this technology, where the performance of P4 programmable packet processors is benchmarked and modeled in Chapters 3 and 4, while its management and applicability in different use case scenarios are studied in Chapters 5 and 6.

3. Benchmarking the Performance of P4 Programmable Packet Processors

While P4 programmable packet processors are promising due to the flexibility they provide in customizing their packet processing behavior, it is crucial to understand their performance and identify their limitations. This performance can be the Achilles' heel in case the required performance level for certain use case scenarios is not met.

There are different performance metrics relevant to evaluating networking devices. These include throughput, forwarding latency, jitter, maximum delay, and power consumption. Among these, evaluating the forwarding latency is becoming more crucial with the emergence of delay-critical applications that require low and deterministic latency such as virtual/augmented reality and cloud gaming. For this reason, evaluating the forwarding latency of P4 programmable devices is given special attention in the rest of the thesis.

The packet processing in P4-based systems is influenced by different constituting components as shown in Fig. 3.1. The performance of these components is studied in this chapter. The following four performance criteria are specifically studied in the four sections of the thesis.

(1.) The first building component of P4-based systems is the data plane pipeline. It is responsible for processing incoming packets. This pipeline is reconfigurable and can be defined according to the loaded P4 program. This reconfigurability introduces the processing **complexity** of the loaded P4 program as a new factor that could influence the packet processing latency on the P4 target. The relation between the P4 pipeline complexity and the forwarding latency should be studied to understand the performance of P4 targets.

(2.) While still focusing on the target's data plane, the forwarding performance of the P4 targets can also be affected by the number of installed rules and the number of distinct flows coming into the P4 target. The stability of the forwarding performance of the P4 targets in response to a scaled number of incoming flows is another important aspect that should be studied.

(3.) On top of the data plane pipeline, the control agent interacts with the SDN controller. Analyzing the reaction time of this control agent to control plane commands is important for identifying any inconsistencies between the states of the control and data planes.

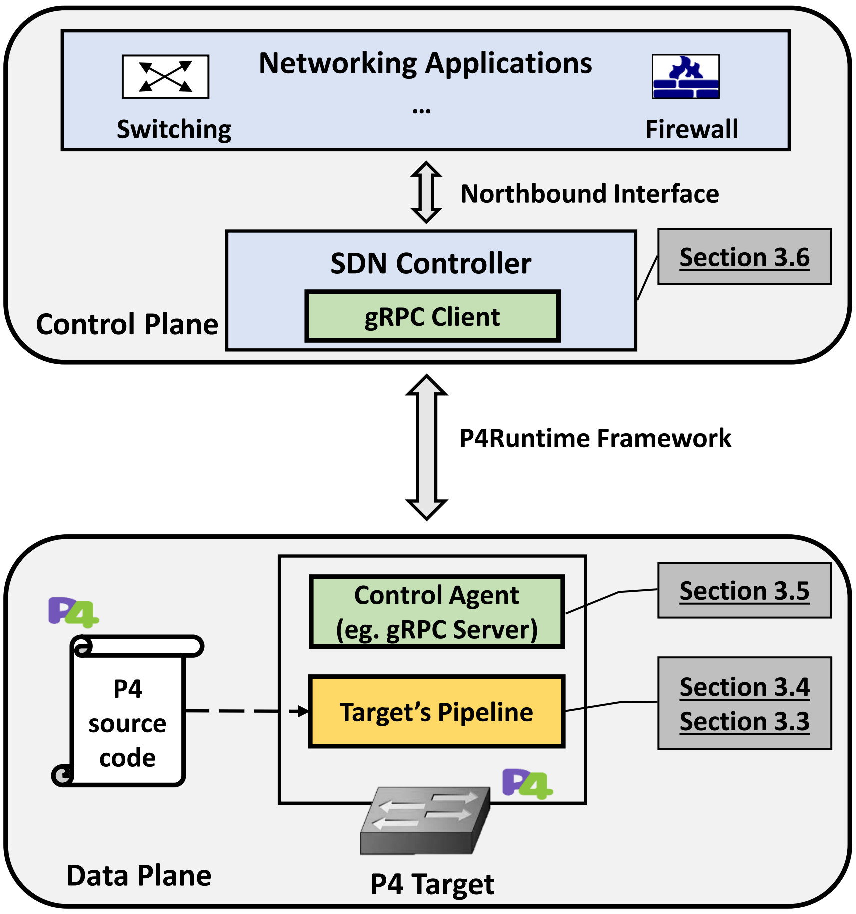


Figure 3.1.: Components building a P4-based system with control and data plane. The mapping between the blocks and the corresponding evaluation sections is highlighted in the figure.

(4.) The last investigated component is the SDN controller. The SDN controller intervenes in case the packet processing behavior for certain flows is not defined in the data plane. This is usually the case when a packet corresponding to a new flow has no matching rule in the data plane's match-action unit. The performance of the SDN controller plays a major role in determining the overall performance of P4-based systems.

This chapter is organized as follows. First, we discuss related works that deal with the performance evaluation of P4 programmable devices in Section 3.1. In Section 3.2, we conduct a detailed evaluation of the data plane's packet forwarding latency of different P4 programmable devices with a special focus on the influence of P4 programmability. The content of this section is based on our two published works [1,2]. In Section 3.3, we study the impact of MAU occupancy and flow scalability on the forwarding performance of P4 targets, where the presented results are based on our previous publication [3]. The control agent of P4 devices is evaluated in Section 3.4 focusing on its responsiveness to control plane update rules, where presented findings are based on our previous publication [3]. In Section 3.5, a benchmarking tool for evaluating the performance of P4Runtime-based SDN controllers is proposed and used for evaluating such controllers. The contributions presented in Section 3.5 are based on our publication [6]. Finally, Section 3.6 summarizes the findings of this chapter.

3.1. Related Work

The state-of-the-art related to measuring and evaluating the performance of the P4 data plane and SDN control plane is revisited in this section.

3.1.1. Performance Evaluation of P4 Data Plane

While literature is rich with works evaluating the performance of OF switches, there are few that focus on P4-based devices yet.

The work in [30] proposes WhipperSnapper as a suite for benchmarking P4 targets. The suite includes evaluating the performance of the P4 target when executing different $P4_{14}$ operations. The BMv2 and PISCES software switches, as well as that of the P4FPGA emulator [13] are evaluated. Although $P4_{14}$ is now the legacy version of P4, still this work is considered the first dedicated work related to evaluating the performance of P4 devices. Authors of [32] model different performance metrics based on the type of the P4 target. The resource utilization is selected for ASIC-based devices such as the Tofino-based P4 switch, while latency and throughput are selected for software switches such as t4p4s. In [41], detailed measurements and analyses for understanding the performance of Agilio CX SmartNIC are conducted.

Other works focused on evaluating the performance of P4 devices when executing specific functionality. For example, [40] evaluates the latency when using hashing extern function on NetFPGA, t4p4s, and SmartNIC P4 targets. [37–39] evaluate the performance of SmartNIC and Tofino P4 targets when performing in-network event processing, while [35, 36] evaluate the performance of NetFPGA and Tofino when executing stateless load-balancing functionality.

Although the objective of [33] is on optimizing the placement decision of P4 programs between SmartNICs and software switches, a limited evaluation of table entry modification response time is conducted in this work. The latter is used as an input parameter for solving the placement problem. However, the description of the experimental setup for conducting this measurement is limited, which hinders the applicability of these results beyond the scope of that paper.

The performance evaluation of P4 data planes conducted in this chapter shares the same objective as these research works. However, the evaluation conducted in this chapter is meant to be comprehensive by varying many parameters and by considering different P4 targets to enable comparing the collected results for deriving useful conclusions. The findings of this chapter allow for comparing different P4 targets in terms of performance capabilities and limitations.

3.1.2. Performance Evaluation of SDN controllers

The controller is the key component in the SDN architecture, as it is responsible for controlling and managing the data plane switches. For example, it takes care of processing packets that do not match any installed rule in the data plane. The performance of the SDN controllers plays a major role in determining the overall performance of the SDN system. While many works try to benchmark and evaluate the performance of SDN controllers with OF as a southbound interface, there is none that benchmarks controllers with P4RT for controlling P4-based switches. This is due to the maturity of the OF protocol compared to the relatively new P4RT framework. In the following, some benchmarking tools for evaluating the performance of OF-based SDN controllers are described.

Scott et al. present a simulator for troubleshooting SDN in [51]. This simulator addresses the issues that arise when developing SDN platforms as network management services. The troubleshooting techniques used are based on inspecting logs for relevant information. The simulator then automatically identifies the bare minimum of inputs required to reproduce an identified bug.

The first open-source tool proposed for benchmarking OpenFlow Controllers is Cbench [112]. This tool emulates an arbitrary number of OpenFlow switches. Each virtual OpenFlow switch establishes a connection with the SDN controller and goes through the handshake process. When benchmarking begins, each switch sends as many Packet-In messages to the controller as possible in order to evaluate its performance. The tool can assess the throughput and latency of traffic exchanged between virtual switches and the controller.

OpenFlow Controller Probe (OFCProbe) is a benchmarking tool for OF controllers developed in [21]. The tool is used to compare the performance of three different OF controllers: Floodlight [108], NOX [110], and Maestro [111]. While the benchmarking approach used by OFCProbe is similar to that used by Cbench in terms of simulating virtual OF switches, OFCProbe focuses on more advanced design goals:

- Platform independence which enables the tool to run on the majority of common system architectures.
- Scalability, which allows the tool to run on multiple cores, CPUs, and even hosts.

- Modularity to allow easy evolution of the tool.
- Detailed statistics that allow monitoring the performance of the OF controller per switch and over time. When dealing with multiple switches, this allows evaluating the fairness of OF controllers.

OFCProbe was further extended in [22] to allow configuring the topology of the emulated network and reporting additional performance indicators such as network topology discovery time. The extended tool is used to evaluate various performance metrics of the ONOS [109] controller.

In [27], Tootoonchian et al. investigate the impact of the OF controller performance on the overall performance of SDN networks. NOX, NOX-MT (maximum throughput), Beacon [42], and Maestro controllers are investigated. The results show that the performance of these controllers was better than what is expected/reported in the literature.

Shalimov et al. [25] present an open-source benchmarking tool that can compare various efficiency indexes. Performance, scalability, and security are among these indexes. A thorough examination of various SDN controllers (NOX, Beacon, Floodlight, Maestro, etc.) is carried out, with various indexes indicating that the evaluated controllers need to be improved before they can be used in production environments.

Alencar et al. [26] evaluate the performance of two Java-based SDN controllers, Floodlight and Beacon. CPU utilization, memory, and Java virtual machine (JVM) memory consumption are recorded. Monitoring the performance of these controllers showed software aging effects.

Authors of [23] compare the performance of the OpenDaylight [113] SDN controller to that of the Floodlight controller. The Cbench tool is used to evaluate throughput and latency results. When compared to Floodlight, the results showed that OpenDaylight serves fewer requests on average. Furthermore, the authors propose enhancements to the Cbench tool to accommodate models of real-world traffic in data centers.

In [24], various state-of-the-art SDN controllers are surveyed. Then, they use Cbench to evaluate new controllers such as the ONOS controller. Based on their findings, they concluded that there is no dominant SDN controller and that the choice should be based on use case requirements.

Published works in literature do not contain any studies on the performance of SDN controllers when running in P4RT-mode for controlling P4 targets. Therefore, we had to fill this research gap in Section 3.5 by studying the performance of P4RT-based SDN controllers and by building the appropriate benchmarking tool for this purpose. The proposed benchmarking tool in Section 3.5 extends the OFCProbe tool [20] for benchmarking OF-based controllers described in this subsection.

3.2. Evaluating the Complexity of P4 Data Path

The forwarding latency of networking devices is an important performance metric. The forwarding latency is usually impacted by different factors such as processing platform, packet size, the load of the device, etc. In the case of programmable packet processors, such as P4 programmable devices, the forwarding latency is influenced by one more new variable, and this is the processing **complexity** of the loaded P4 program. In other words, if the loaded data path performs simple operations on the incoming packet, the processing latency will be smaller compared to the case when more sophisticated packet processing is applied.

Understanding the relationship between packet forwarding latency and pipeline complexity helps in understanding and predicting the packet forwarding performance of programmable packet processors. The latter enables network operators to better manage their networking infrastructure as they can provide performance guarantees (e.g., latency guarantees) to their tenants for a known traffic flow — an important feature in cloud data centers. Furthermore, measuring the forwarding latency of networking devices is the first building block for deriving analytical models of their performance as accomplished in Chapter 4. In addition, knowing the packet latency of running different network functions on different programmable devices in advance is critical for the optimal scheduling and provisioning of network functions in a heterogeneous P4-based environment, as will be discussed in Chapter 5.

Towards understanding the relationship between the forwarding latency and the pipeline complexity executed on a P4 device, it is clearly not possible to measure the forwarding latency of all possible P4 programs, as this could lead to an uncountable number of possibilities. Alternatively, we rely on two interesting observations of the P4 language that enables this endeavor:

- P4 abstracts the programming of packet processing pipelines into a limited set of P4 atomic constructs or operations such as parsing a header, adding a header to a packet, modifying the header, etc. The combination of this limited set of atomic P4 constructs builds a full P4 program that can describe realistic network functions.
- The P4 language limits performance variations at run-time by preventing loops with an unknown number of iterations, and by preventing dynamic memory allocation.

Therefore, we first measure and analyze the impact of each P4 atomic construct on the packet forwarding latency of different P4 programmable packet processors. This analysis paves the way toward deriving a model for characterizing the forwarding latency of P4 programmable devices as a function of the loaded P4 program as will be discussed in Section 4.2. The contributions presented in this section are based on our two published works [2] and [1]. The experimental setup and designed experiments

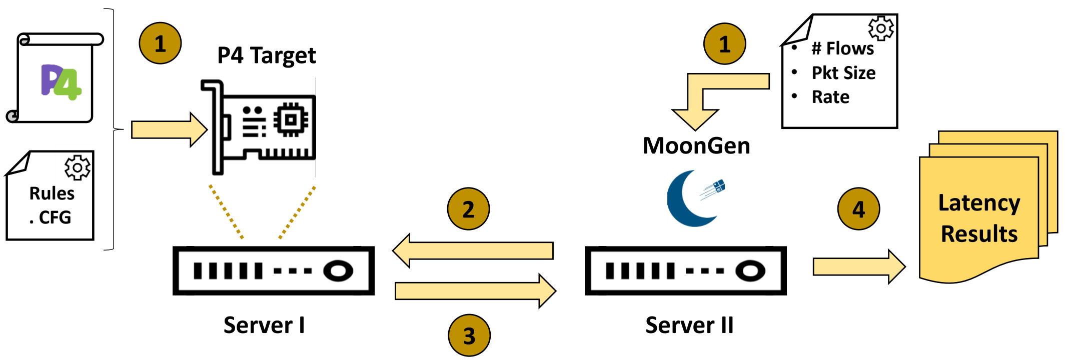


Figure 3.2.: Experimental setup made up of two servers connected by two 10 Gbps cables: Server I hosting the P4 device, and Server II hosting the packet generator MoonGen. MoonGen generates the measurement traffic and sends it to be processed in the P4 device hosted in Server I, and then analyzes the returned traffic to generate latency reports.

are described in Subsection 3.2.1, followed by Subsection 3.2.2 where the results of these experiments and presented and interpreted. In Subsection 3.2.3, we summarize the findings of this section.

3.2.1. Testbed and Experiment Design

In this section, we first describe the measurement testbed shown in Fig. 3.2 used for conducting the evaluation. Then, we elaborate on the different experiments designed for benchmarking the latency cost of executing different P4 constructs.

The experiments are carried out on two Nokia NDCS16RM AirFrame Compute Nodes, each has 16 cores (dual-socket Intel Xeon CPU E5-2630 v3 @ 2.40 GHz) and 64 GB of 2133 MHz DDR4 memory. Each server has an 82599ES 10-Gigabit Ethernet network interface card installed. Three different P4 targets belonging to different processing platforms are evaluated: *(1)* An Agilio CX 2x10GbE SmartNIC from Netronome [101], *(2)* A NetFPGA-SUME board with Xilinx Virtex-7 XC7V690T FFG1761-3 FPGA [99], and *(3)* An open-source DPDK-based software switch called t4p4s [103].

We attach the two hardware targets, i.e., NetFPGA-SUME and Agilio CX SmartNIC, to the PCI bus of Server I, while we run the t4p4s software switch on this server. We connect the two physical ports of each investigated P4 target to two interfaces of Server II, where MoonGen packet generator [31] runs. The MoonGen packet generator is a DPDK-based packet generator capable of generating more than 10 Gbps Ethernet traffic. It makes use of the hardware timestamping feature of modern commodity NICs to deliver accurate and precise latency measurements with sub-microsecond precision [31].

Table 3.1.: Parameters considered for evaluating the processing latency of atomic P4 constructs.

Varied Parameter	**Value**
P4 Constructs	Headers Parsing, Header's Fields Modification, Headers Modification, Headers Copying, Headers Removal, Headers Addition, Tables Addition
P4 Targets	Agilio CX SmartNIC, NetFPGA-SUME, t4p4s DPDK-based Software Switch
Packet Sizes (in Bytes)	256, 1000, 1500
Rate (in Gbps)	9 to 10

In each measurement case, we vary the P4 program to be loaded into the investigated P4 target. MoonGen generates packets and sends them over one link to the studied P4 target. The bitrate of generated traffic is configured to be equal to the line rate of the device under test, i.e., the maximum rate that could be handled by the device without packet drop. This rate is found to be 10 Gbps for hardware devices and 9.7 Gbps for t4p4s software switch. The generated traffic gets processed in the P4 device according to the loaded P4 program, then, they are sent back to MoonGen for measuring and reporting the per-packet latency. In the evaluation, we test the effect of the size of the packet on the forwarding latency by examining small, medium, and large-sized packets with values equal to 256, 1000, and 1500 Bytes, respectively. We collect over 100,000 data points of latency values for each measurement case using MoonGen.

Benchmarking the latency cost of executing different P4 constructs is divided into seven experiments, where each experiment considers a single P4 construct at a time. The P4 constructs under consideration include parsing headers, performing various header operations, and utilizing match-action tables. In each experiment, the P4 construct under test is incremented in different P4 programs, while ensuring that the examined P4 construct is the only variable in these programs. Accordingly, we can track any variation in the measured packet forwarding latency and relate it to the processing latency of the varied P4 construct. The pipelines are designed with different initial parsing stages to ensure that the parsing operation is the same within one experiment. All pipelines also include minimal processing of a single table that matches against the ingress port of any incoming packet and forwards it back to MoonGen by changing its egress port according to a pre-installed rule. The different parameters considered in this evaluation are summarized in Table 3.1. In the following, we describe the goal and the design of the different conducted experiments.

Headers Parsing

The purpose of this experiment is to quantify the processing latency cost of parsing headers in a P4 pipeline. We start with a baseline P4 program, denoted by **Base_1**, which parses a single Ethernet header. Then, we gradually increase the number of parsed headers up to 14 while measuring the packet forwarding latency of each tested P4 pipeline. The parsed-headers-stack consists of Ethernet, IPv4, UDP, Precision Time Protocol (PTP), and ten dummy headers each of size equal to 16 Bytes. The first four headers should be kept for the used packet generator, MoonGen [31], to properly measure the per-packet latency.

Header's Fields Modification

In this experiment, the latency cost of modifying the fields of a header is investigated. More specifically, we track the difference between modifying a single field of a header like the IPv4 destination address versus modifying multiple fields of that header as when modifying the source and destination IPv4 addresses of the IPv4 header. The experiment is designed to compare the latency of two triplets of P4 programs. In the first triplet, we modify a single field of Ethernet, IPv4, and UDP headers, while in the second triplet, we modify multiple fields of these headers to compare with.

Headers Modification

This experiment targets to investigate the impact of modifying a different number of headers in a P4 pipeline. First, we start with a baseline P4 pipeline, denoted by **Base_14**, that parses 14 headers. Then, we incrementally modify the 14 parsed headers in different P4 programs by changing one field of each header at a time. The latency results of the 14 different pipelines are measured and recorded. Given that headers cannot be modified without being parsed, **Base_14** is selected to be a common pipeline in all the cases to unify the parsing header operation in all the programs and to make sure that only header modification operation distinguishes different pipelines. A similar approach for selecting the baseline pipeline is followed in the remaining experiments to make sure that the examined P4 construct is the only variable in each experiment.

Headers Copying

The latency cost of copying-header operation, where one parsed header is copied into another parsed header in a P4 pipeline, is studied in this experiment. We select the **Base_14** P4 pipeline as a baseline pipeline, and then we increment the number of copied headers from 1 to 10 in different P4 programs while tracking the latency results.

Headers Removal

In this experiment, we measure the latency cost of removing headers from a packet. We use **Base_14** P4 pipeline as a baseline pipeline. Then, we measure the forwarding latency of 10 P4 programs that vary only in the number of removed headers.

Headers Addition

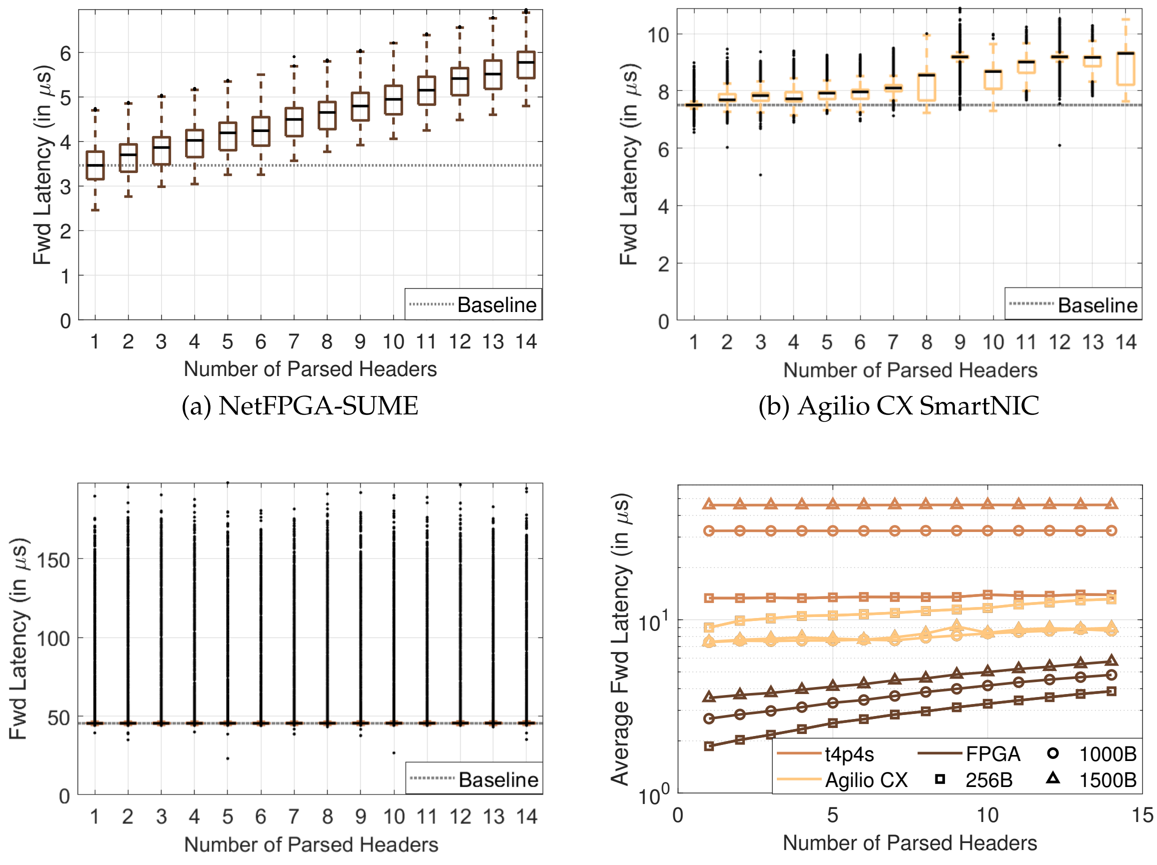
The latency cost of adding headers into a packet in a P4 pipeline is studied in this experiment. The baseline pipeline selected for this experiment is denoted by **Base_4**, where Ethernet, IPv4, UDP, PTP headers are parsed. Then, we add 1 to 10 headers incrementally into the header stack after the four parsed headers using ten different P4 pipelines while measuring the latency of each pipeline. Maintaining the order of the first four parsed headers at the beginning of the header stack is important for the operation of MoonGen, which uses the PTP protocol in measuring the per-packet latency [31]. For this experiment, the maximum generated packet size is selected to be 1300 Bytes instead of 1500 Bytes to ensure that the processed packets, with the newly added headers, never exceed the Maximum Transmission Unit (MTU) size. Moreover, it is important to decrease the rate of generated traffic sent to the P4 target when more headers are added to the packet to make sure that the bitrate of the processed packets, with the added headers, does not exceed the line rate of the P4 target or the link's capacity, i.e., 10 Gbps.

Tables Addition

The goal of this experiment is to measure the latency cost of adding tables into a P4 pipeline. Tables, or MAUs, are the basic processing units in a P4 program. We begin with the **Base_4** pipeline, which already includes a single table matching on the ingress port for writing the egress port according to a proactively installed rule. Then, we add 1 to 14 additional tables to the P4 pipeline's ingress stage while measuring the packet latency corresponding to different programs. Every new table performs exact matching on a single field and takes the same forwarding action.

3.2.2. Evaluation

In this subsection, we will present the evaluation results of the different experiments introduced in Subsection 3.2.1.



(a) NetFPGA-SUME (b) Agilio CX SmartNIC

(c) t4p4s switch (d) Average forwarding latency of all cases

Figure 3.3.: Measured forwarding latency as a function of the number of parsed headers. Packet size equals 1500 Bytes unless otherwise specified.

Headers Parsing

The box plots in Fig. 3.3 show the minimum, first quartile, median, third quartile, and the maximum of the measured packet forwarding latency in μs as a function of the number of parsed headers when the packet size is set to MTU. The NetFPGA-SUME card results, presented in Fig. 3.3a, show a linear increase in the forwarding latency as a function of the number of parsed headers. The measured latency is recorded to increase by around 2.3 μs as the number of parsed headers increases from 1 to 14. The parser states of a P4 program are compiled into FSM on the FPGA board, which appears to have a significant impact on the processing latency [44]. Nevertheless, it is observed that the NetFPGA-SUME maintains a stable performance while processing an increasingly complex pipeline with more parsing states as the distribution of measured packet latencies is observed to be consistent when the number of parsed headers increases.

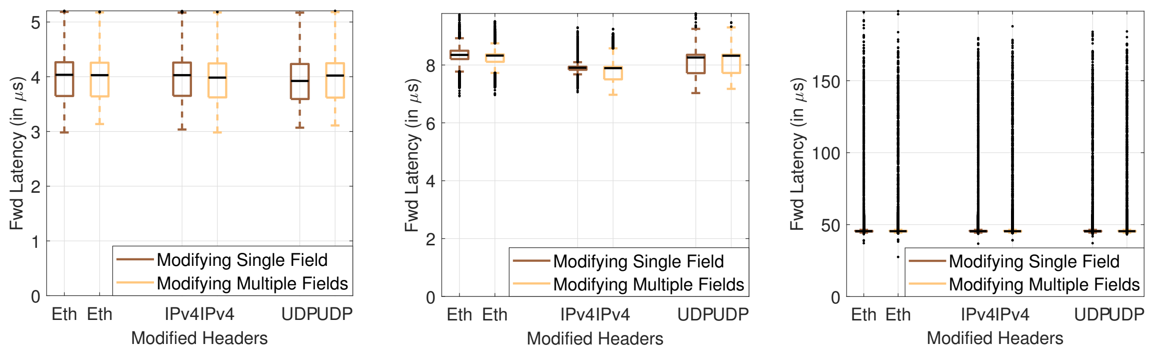
Fig. 3.3b corresponds to the measurement results of Agilio CX SmartNIC. It shows that the median forwarding latency of the **Base_1** pipeline when only a single header is parsed is around 7.5 μs. The latency then increases slightly when the number of parsed headers increases to 14.

The results corresponding to the t4p4s DPDK-based software switch are shown in Fig. 3.3c. The measurements show that the distribution of the forwarding latency is invariant when the number of parsed headers increases where the median is always almost equal to 45 μs. It is notable that while most of the measured packet latencies are centered around the median, there are few outliers recorded at values reaching as high as 200 μs. These outliers take place periodically and the reason for that is interpreted to be a result of batch processing executed in DPDK.

The average forwarding latency corresponding to the P4 targets is plotted in Fig. 3.3d in μs and in the logarithmic scale as a function of the number of parsed headers scale when the packet size is set equal to 256, 1000, and 1500 Bytes. On average, it can be observed that the forwarding latency of NetFPGA-SUME is smaller than that of Agilio CX SmartNIC, which is smaller than that of the t4p4s software switch. Similar to trends observed in the other subplots, the packet latency increases slightly in NetFPGA-SUME and Agilio CX SmartNIC cases but stays constant in the case of the t4p4s switch. Looking at the effect of varying the size of packets, we can observe that the latter results in shifting the curves, while slopes are still the same. This means that the variation of packet size contributes to a constant delay disregarding the packet processing defined in the P4 program, which is usually applied to the headers of a packet. This constant delay is due to the packet's payload handling, i.e., storing then forwarding it. While the relation between packet size and forwarding latency is proportional in the cases of NetFPGA-SUME and t4p4s, it is not the case for Agilio CX SmartNIC, where the forwarding latency is larger when the packet size is equal to 256 Bytes compared to the cases when the packet size is equal to 1000, and 1500 Bytes. The latter is interpreted to be a result of the frequent expensive memory access in Agilio CX SmartNIC when the packet size is small, i.e., the packet rate is high [29].

Header's Fields Modification

The measurement results, plotted as box plots, corresponding to the cases when modifying a single field of three different headers versus the cases when modifying multiple fields of these three headers are shown in Fig. 3.4. Different subplots correspond to the results of the three investigated P4 targets when the packet size is set equal to the MTU. In all these cases, the measured latency when modifying a single field of a header looks very similar to the latency when multiple fields of a header are modified. Recall that the deparsing stage of the P4 language is programmed using *emit* commands, where this *emit* command works at the level of headers to reconstruct the packet at the end of the packet processing pipeline. Accordingly, different P4 targets rewrite the



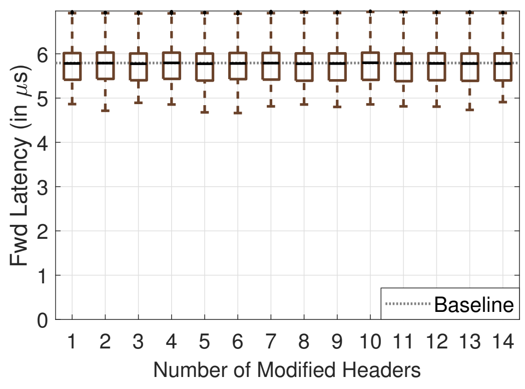
(a) NetFPGA-SUME (b) Agilio CX SmartNIC (c) t4p4s switch

Figure 3.4.: Measured forwarding latency as a function of the number of modified headers' fields for packet size equals 1500 Bytes.

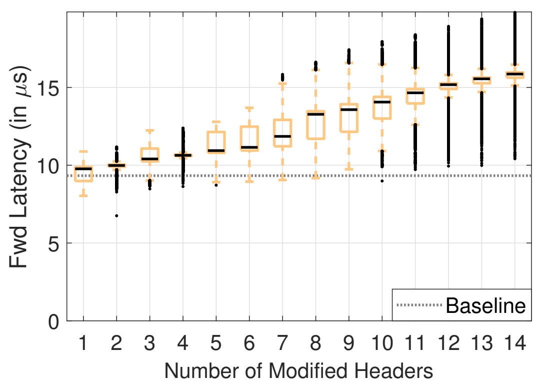
full memory space allocated to a modified header disregarding how many fields of that header are modified when the *emit* command is applied in the deparsing stage. Therefore, the packet latency of a P4 program does not depend on the number of fields modified within a header.

Headers Modification

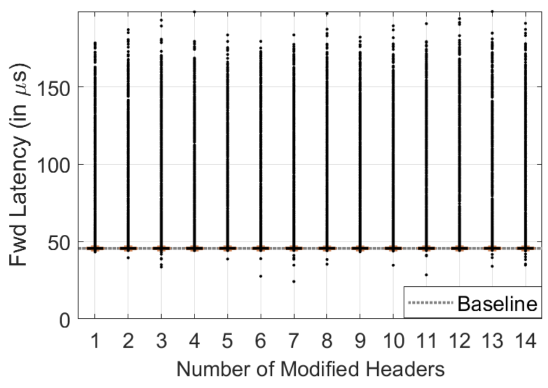
This section analyzes the results of modifying headers where the forwarding latency of different P4 pipelines with an increasing number of modified headers is measured. Recall that from the previous experiment, we found out that the number of fields modified within a header does not affect the forwarding latency as the P4 targets modify the full header when any of its fields are modified. Figures 3.5a, 3.5b, and 3.5c show the box plots of measured latency in μs for a packet size equal to the MTU when the number of modified headers increases on NetFPGA-SUME, Agilio CX SmartNIC, and the t4p4s software switch, respectively. Moreover, the latency of **Base_14** pipeline, where 14 headers are parsed, is plotted for different P4 targets. From these figures, we can observe that while the median of the forwarding latency increases almost linearly as a function of the increasing number of modified headers in the case of Agilio CX SmartNIC, it stays constant in the cases of NetFPGA-SUME and the t4p4s software switch, where the measured latency is recorded to be 5.8 and 45 μs, respectively. The latter is because the t4p4s software switch rewrites the full header stack of a packet when a single header is modified [114]. The NetFPGA-SUME follows a similar approach because keeping track of which headers have been modified out of those defined in the packet's header stack requires additional logic on the card. This additional logic requires additional processing latency and thus makes it more practical to always modify the complete header stack when any of its constituting headers is modified [43]. Unlike the latter two P4 targets, Agilio CX SmartNIC marks the modified headers with *dirty* flag, and consequently only rewrites the memory space



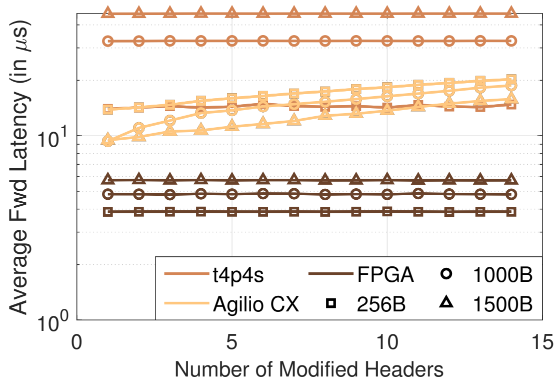
(a) NetFPGA-SUME



(b) Agilio CX SmartNIC



(c) t4p4s switch

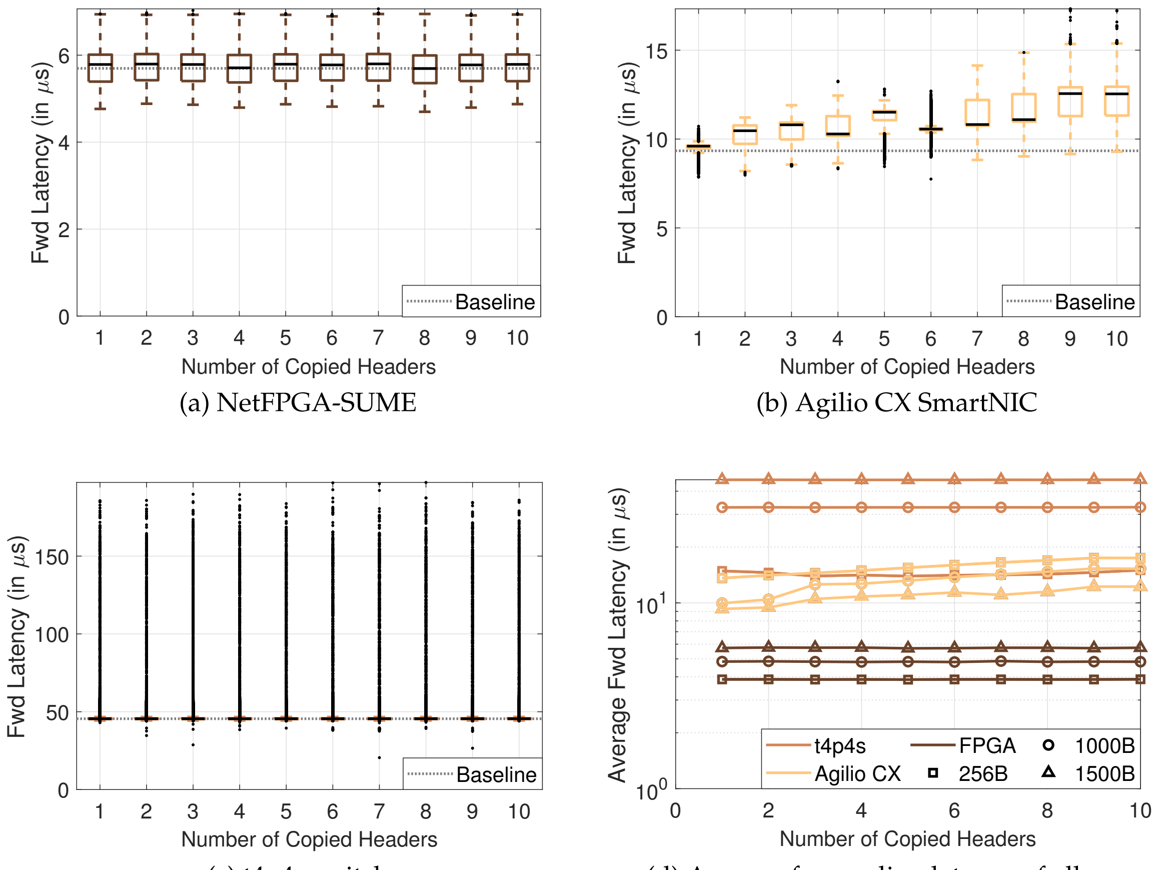


(d) Average forwarding latency of all cases

Figure 3.5.: Measured forwarding latency as a function of the number of modified headers. Packet size equals 1500 Bytes unless otherwise specified.

corresponding to the modified (marked) headers.

Fig. 3.5d shows the average measured latency in logarithmic scale and in μs for different packet sizes as a function of the number of modified headers for the three benchmarked P4 targets. This plot shows a similar dependency as that observed in the other subplots between packet forwarding latency and the number of modified headers. Also, the effect of packet size on packet forwarding latency is similar to that observed in Fig. 3.3d. Furthermore, we can observe that the average forwarding latency measured on NetFPGA-SUME is always less than that measured on the other two P4 devices. Also, the Agilio CX SmartNIC has lower forwarding latency compared to the t4p4s software switch except for the case of small packet sizes. Finally, it is worth noting that the impact of IP checksum execution is also tested when modifying the IP header, where results showed a negligible variation in the packet forwarding latency on the three P4 targets.

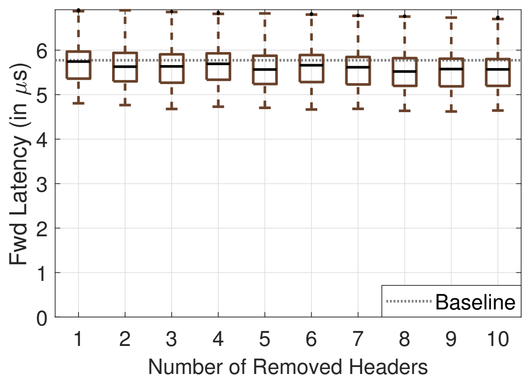


(a) NetFPGA-SUME (b) Agilio CX SmartNIC (c) t4p4s switch (d) Average forwarding latency of all cases

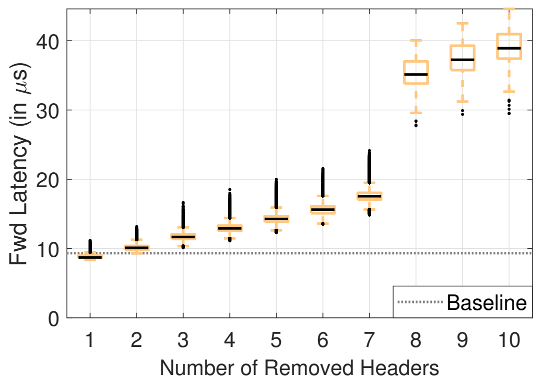
Figure 3.6.: Measured forwarding latency as a function of the number of copied headers. Packet size equals 1500 Bytes unless otherwise specified.

Headers Copying

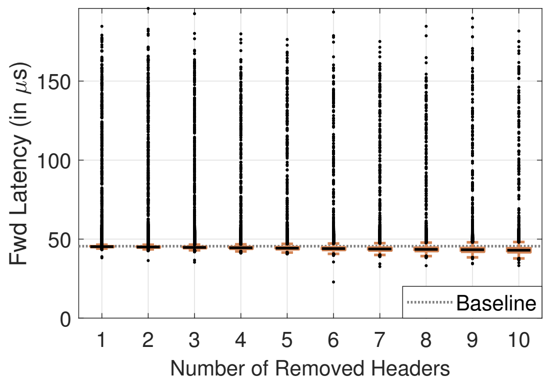
The results when copying a different number of headers within a P4 pipeline are analyzed in this subsection. The pipelines of this experiment are built on top of **Base_14** pipeline where 14 headers are parsed and whose median latency results are presented in the first three subplots of Fig. 3.6 as baselines. In general, the box plots of the packet forwarding latency plotted in these subplots for different P4 devices and for MTU-sized packets, resemble a similar behavior to that recorded in the previous experiment when the number of modified headers is varied between 1 and 10. The same is observed when inspecting Fig. 3.6d, which shows the variation of average forwarding latency on the three devices and for three different packet sizes as a function of the number of copied headers in μs and in the logarithmic scale. The reason behind the similarity between the results of this experiment and the previous one is that the copy header operation is an extreme case of modifying header operation, i.e., in a copy header operation we modify all the fields of the destination header. Also given the findings of **Header's Fields Modification** experiment, where we identified that the



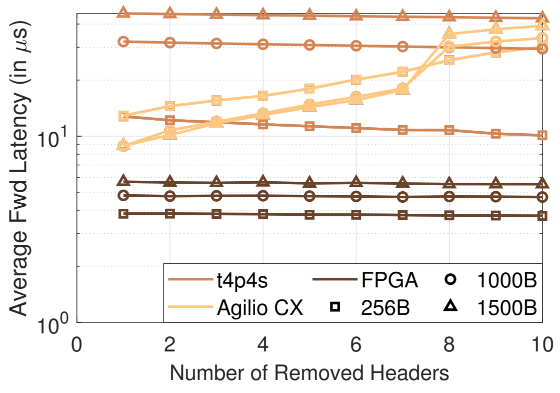
(a) NetFPGA-SUME



(b) Agilio CX SmartNIC



(c) t4p4s switch



(d) Average forwarding latency of all cases

Figure 3.7.: Measured forwarding latency as a function of the number of removed headers. Packet size equals 1500 Bytes unless otherwise specified.

number of fields modified within a header is irrelevant to the packet forwarding latency, we can conclude that the copy header experiment and modify header experiment have a similar forwarding latency because they both result in similar packet processing to be executed in the P4 targets. There is a slight difference in the measured latency of these two experiments when the number of headers is less than 4 because the first four modified headers in these two experiments are different as explained earlier in the description of these two experiments.

Headers Removal

The box plots corresponding to the packet forwarding latency as a function of the number of removed headers are plotted in the first three subplots of Fig. 3.7 in μs and for MTU-sized packets. Additionally, the forwarding latency of **Base_14** pipeline is plotted in these subplots.

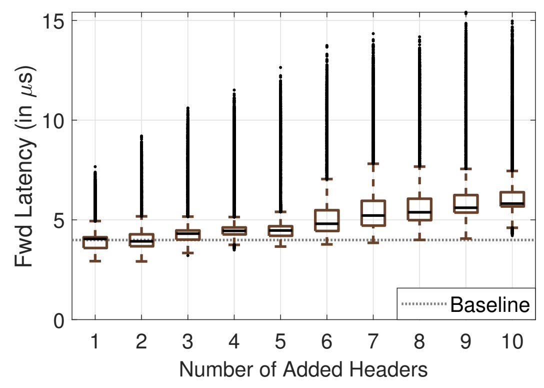
In the case of NetFPGA-SUME and t4p4s software switch, the median of the measured forwarding latency slightly decreases by 0.1 and 2 μs when the number of removed headers increases up to 10, as plotted in Fig. 3.7a and 3.7c, respectively. This decrease is probably due to the decreased size of the packet's header stack to be emitted. The distribution of collected packet latency for both devices is similar to that observed in the previous experiments. The results corresponding to Agilio CX SmartNIC, depicted in Fig. 3.7b, show that the forwarding latency linearly increases from 8.7 μs to 17.5 μs when the number of remove header operations increases from 1 to 7. Afterward, the latency sharply increases to 35.1 μs when one more remove header operation is applied. Then, the latency increases again linearly up to 39 μs when the number of removed headers increases from 8 to 10. This sharp increase in latency after removing 7 headers is related to implementation specifics of the Agilio CX SmartNIC, where an infrastructural process is involved at the deparsing stage causing this high processing latency. This involved process is dependent on the size of the removed headers. If the size of removed headers is larger than a threshold, the execution time of this process becomes larger as it triggers moving the whole payload of the packet from one memory space to another. The movement of the payload also makes the execution time of this process depends on the size of the payload. The collected results show that this behavior takes place when more than 7 headers each of size 16 Bytes, i.e., $7 \cdot 16B = 112\,B$, need to be removed from the packet.

Figure 3.7d shows the average forwarding latency of the three devices for different packet sizes in μs and in logarithmic scale as a function of the number of removed headers. The previously described relation between packet size and forwarding latency for all three devices still holds, except for Agilio CX SmartNIC after 7 headers need to be removed where in this case the forwarding latency becomes larger for larger packets because of the implementation specifics described before. Also similar to previous results, NetFPGA-SUME still outperforms Agilio CX SmartNIC, which outperforms the t4p4s software switch except for two cases: *(1)* when the packet size is equal to 256 Bytes; and *(2)* when the packet size is equal to 1000 Bytes and more than 7 remove headers operations are executed.

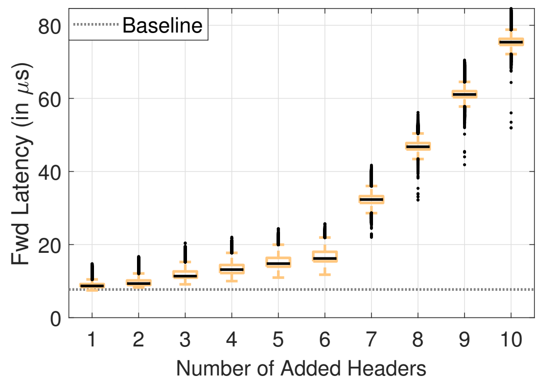
Headers Addition

The first three subplots of Fig. 3.8 depict the box plots of measured forwarding latency of packets of size 1300 Bytes in μs when the number of add header operations increases from 1 to 10. The median of the latency corresponding to the **Base_4** pipeline, for MTU-sized packets, are also plotted in these figures to approximate the baseline case.

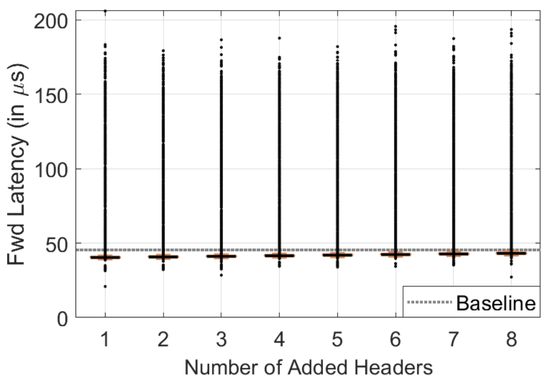
The results corresponding to NetFPGA-SUME, depicted in Fig. 3.8a, show that the median of the latency increases linearly from 4 to 5.8 μs when the number of added headers increases from 1 to 10. This increase is due to the grown header stack, because of added headers, to be emitted.



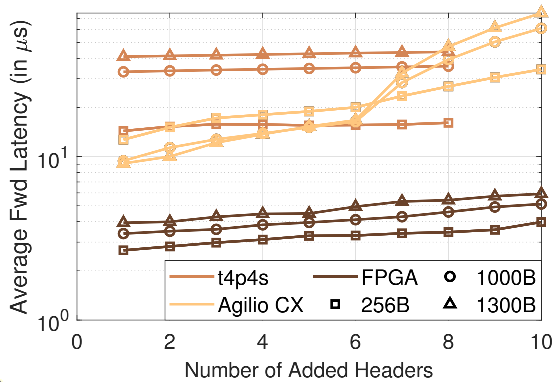
(a) NetFPGA-SUME



(b) Agilio CX SmartNIC



(c) t4p4s switch



(d) Average forwarding latency of all cases

Figure 3.8.: Measured forwarding latency as a function of the number of added headers. Packet size equals 1500 Bytes unless otherwise specified.

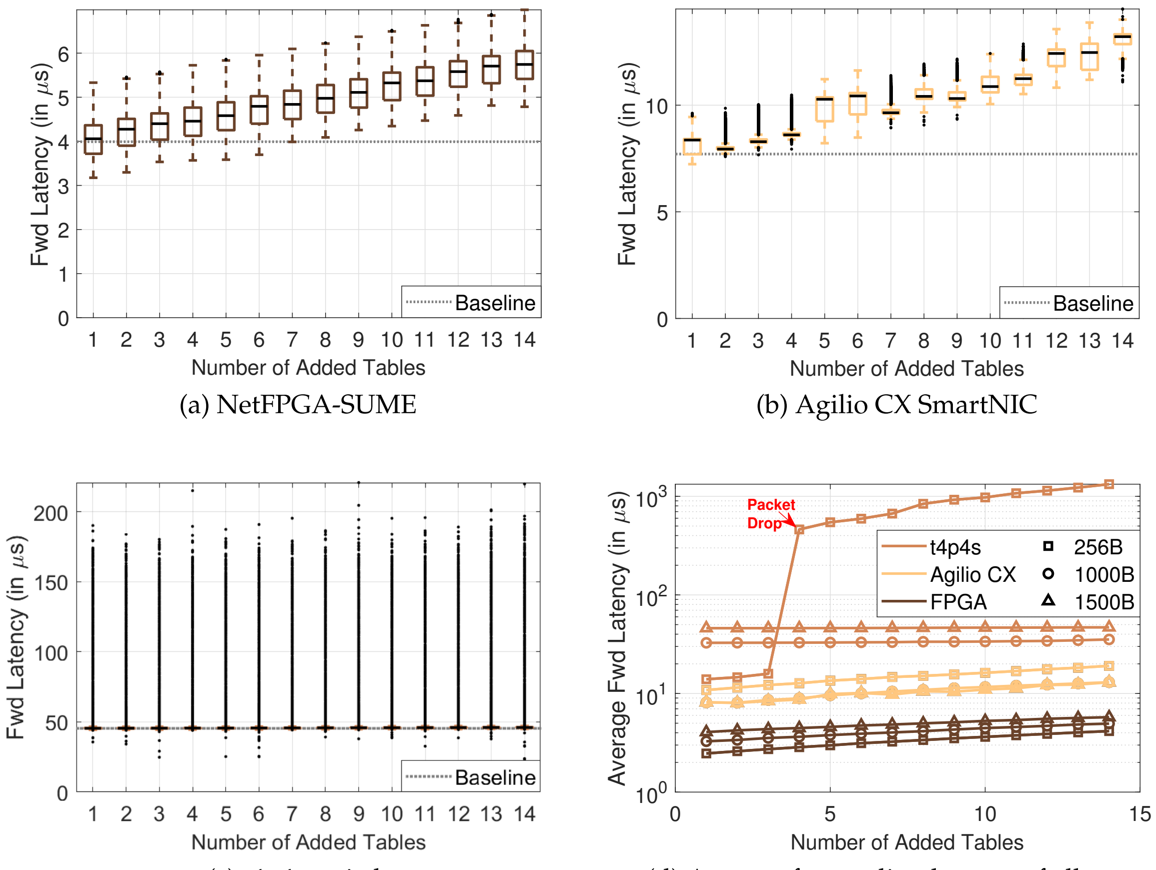
Similarly, the latency results corresponding to Agilio CX SmartNIC, depicted in Fig. 3.8b, show a linear increase from 8.7 up to 16.2 μs when the number of added headers increases to 6. Afterward, the latency increases more sharply reaching up to 75.4 μs while the number of added headers increases from 6 to 10. The reason for this sharp increase in latency is similar to that clarified in the previous subsection, where the involvement of an infrastructural process for moving the payload of the packet after a certain threshold of added Bytes to the packet results in a large execution time. Based on Fig. 3.8b, this behavior begins after adding more than 6 headers of size 16 Bytes, i.e., $6 \cdot 16\,B = 96\,B$, to the header stack of a packet. Looking at the results corresponding to the t4p4s switch plotted in Fig. 3.8c, we can observe that the latency also slightly increases from 40.3 to 43.1 μs as the number of added headers increases from 1 to 8. This increase is also related to the grown header stack to be emitted. The maximum number of headers that could be added to the t4p4s software switch is equal to 8 headers due to constraints imposed by the implementation of the DPDK framework. This framework reserves a space in front of every processed packet for adding headers. This space is called *RTE_PKTMBUF_HEADROOM* and it is preset to a maximum value equal to 128 Bytes [103]. In this experiment, the size of each added header is equal to 16 Bytes, making the maximum number of headers that can be added to a packet limited to 8 headers, $128\,B/16\,B = 8$. The packets' distributions on different targets are similar to that observed in the previous experiments, except for NetFPGA-SUME, where we observe an increasing occurrence of outliers when more headers are added in the loaded P4 pipeline.

The average forwarding latency of the three devices for different packet sizes in μs and in logarithmic scale as a function of the number of added headers is shown in Fig. 3.8d. The effect of packet size on forwarding latency is the same as described in previous experiments for all three devices, except for Agilio CX SmartNIC after 6 headers are added where in this case the forwarding latency becomes larger for larger packets due to the specific implementation details of the device described before. In all cases, NetFPGA-SUME outperforms the other two targets for all packet sizes. The Agilio CX SmartNIC has a lower forwarding latency compared to the t4p4s software switch except for two cases: *(1.)* when the packet size is equal to 256 Bytes with more than one add header operations; and *(2.)* when the packet size is equal to 1000 and 1500 Bytes with the number of add header operations equal to 8.

Tables Addition

The first three subplots of Fig. 3.9 show the box plots of measured forwarding latency in μs for MTU-sized packets when the number of instantiated tables in a P4 pipeline increases. These figures also show the median of the measured latency of the baseline pipeline defined for this experiment, i.e., **Base_4** pipeline. Figures 3.9a, 3.9b, and 3.9c shows a linear increase of the median of the measured latency by 1.7, 4.8, and 0.6 μs in the cases of NetFPGA-SUME, Agilio CX SmartNIC, and the t4p4s switch, respectively when the number of added tables increases from 1 to 14. This increase in latency is due to the extra processing taking place in the P4 targets because of the increasing lookup and matching operations accompanied by the added tables.

The average measured latency corresponding to the three targets for different packet sizes and as a function of the number of added tables is plotted in Fig. 3.9d in μs and in logarithmic scale. Generally, NetFPGA-SUME always has a lower forwarding latency compared to Agilio CX SmartNIC, which has a lower forwarding latency compared to the t4p4s software switch. Also, the previously identified relationship between the size of packets and measured latency is still valid for all targets, except for the t4p4s software switch after three added tables for small-sized packets. In this case, we observe that t4p4s starts dropping some packets and the packet forwarding latency grows sharply. The latter takes place on t4p4s because the number of lookup operations to be executed increases because of the increased number of tables to be processed and the increased incoming packet rate when the packet size is smaller. When executing a high number of lookup operations, some hardware constraints such as exceeding



(a) NetFPGA-SUME (b) Agilio CX SmartNIC (c) t4p4s switch (d) Average forwarding latency of all cases

Figure 3.9.: Measured forwarding latency as a function of the number of added tables. Packet size equals 1500 Bytes unless otherwise specified.

the cache memory size, available CPU cycles, and memory transfer bandwidth are violated on the hosting machine [103]. For small-sized packets, our evaluation showed that t4p4s can process 14 tables without dropping packets only when the incoming traffic rate never exceeds 4.5 Gbps.

3.2.3. Summary

In this section, we evaluated three state-of-the-art P4 targets to quantify the contribution of different P4 constructs to the overall forwarding latency on these investigated P4 targets. The evaluation showed that NetFPGA-SUME has a lower forwarding latency compared to Agilio CX SmartNIC, which has a lower forwarding latency compared to the t4p4s software switch. Generally, we observed that distinct P4 constructs have a different impact on the processing latency of different P4 targets. Also, we could observe that in most cases when a P4 construct influences the processing latency of a P4 device, there exists a linear relationship between the number of applied P4 constructs and the average forwarding latency on that P4 device. This observation is crucial for deriving a model that quantifies the latency of executing arbitrary P4 programs on different P4 devices as will be illustrated in Section 4.2.

3.3. Evaluating Flow Scalability

After investigating the impact of the complexity of the loaded P4 data plane on the forwarding latency in Section 3.2, in this section, we focus on evaluating the impact of the flow scalability on the packet forwarding latency. The flow scalability analysis focuses on determining the maximum number of rules that can be supported by a P4 device. Assuming that different flows hit distinct rules, this analysis quantifies the maximum number of flows that can be handled by the investigated P4 devices. As an example, such an analysis can reveal the maximum number of users, based on the maximum number of supported rules, that can be defined in a P4 device running a P4 program that implements access control functionality. Moreover, we evaluate in this analysis the impact of the increasing number of incoming flows on the packet forwarding latency of different P4 devices. This metric is important for assessing the behavior of the investigated P4 device when the incoming number of distinct flows scales up. Accordingly, it can reveal the limitations of these P4 devices if they are to be used in production environments with high traffic loads.

In this evaluation, we focus on the MAU (or P4 table) in a P4 pipeline. We identify the maximum number of rules that can be installed in a P4 table and the impact of scaling up the number of distinct incoming flows that match these installed rules on the packet forwarding latency. The contributions presented in this section are based on our publication [3]. In Subsection 3.3.1, we describe the designed experiments, while in 3.3.1 we plot and analyze the collected results. In Subsection 3.3.3, we summarize the findings of this section.

3.3.1. Testbed and Experiment Design

In this subsection, we describe the experimental setup used for conducting this experiment as well as the different parameters used in the evaluation.

Testbed Setup

The measurement setup used in this experiment is similar to that used in the previous experiment studied in Section 3.2, which is shown in Fig. 3.2. The flow scalability analysis is conducted for the same three P4 devices: Agilio CX SmartNIC, NetFPGA-SUME, and the t4p4s software switch. In this experiment, we always load the L3Fwd P4 program into the P4 device, while installing a different set of rules in different evaluated scenarios. Note that in this experiment we do not vary the P4 program as in the previous experiment, rather, we only vary the rules to be installed into the P4 tables. Traffic is generated by MoonGen [31] and sent over a 10 Gbps link to the investigated P4 target. After getting processed in the P4 target, this traffic is sent back to MoonGen over another 10 Gbps link for measuring and reporting the latency results.

In all evaluated cases, MoonGen is configured to generate traffic at the line rate of the investigated P4 devices, i.e., 10 Gbps for hardware devices and 9.7 Gbps for the t4p4s software switch. The packet size is varied to take values equal to 256, 1000, and 1500 Bytes to test its impact on the forwarding latency. The latency measurements of 100 thousand packets are collected by MoonGen in each evaluated case to generate the statistics report.

Evaluated Cases

As the purpose of this experiment is to test the impact of scaling up the number of rules installed in a P4 device and the number of incoming flows to it, we fix the loaded P4 program in the P4 device to always perform L3Fwd functionality. The program is made up of a single table that matches the IPv4 destination address header field and accordingly performs forwarding. The rules installed in this table define the different proactively configured routes. The matching type of the P4 table is also varied and studied in this experiment, where both exact matching and wildcard matching (ternary or LPM) are examined.

Four different cases are studied in this experiment, summarized in Table 3.2, by varying the number of installed rules in the P4 routing table and the number of incoming flows into the P4 target. The first case R_1F_1 is the baseline case where a single routing rule is installed into the P4 table while MoonGen is configured to generate a single flow of packets with IPv4 destination address matching the installed rule. In the second case $R_{max}F_1$, the number of rules installed in the P4 table is increased to reach the maximum possible number, while Moongen is configured to keep sending traffic corresponding to a single flow (hitting a single rule in the P4 table). In the third case $R_{max}F_{1k}$, we keep the P4 table filled with the maximum number of rules that could be installed, but we increase the generated flows coming into the P4 target to reach 1000 flows that hit different installed routing rules. In the last case $R_{max}F_{max}$, we increase the number of generated flows to match the maximum number of installed rules in the P4 target.

Among the attributes of P4 tables in the P4 language is the "Table Size". This attribute specifies the maximum number of rules that can be added to a P4 table, and accordingly the memory space to be reserved for storing these rules. This table size is limited to some value for each P4 device. While this limit represents the *theoretical* maximum

Table 3.2.: Applied parameters with corresponding IDs for the different cases evaluated in the flow scalability experiment.

Case ID	Number of Installed Rules	Number of Processed Flows
R_1F_1	1	1
$R_{max}F_1$	Maximum	1
$R_{max}F_{1k}$	Maximum	1000
$R_{max}F_{max}$	Maximum	Maximum

number of rules that can be added to a P4 table, we found that the P4 devices do not always operate properly when reaching that limit. Accordingly, we searched for the *operable* upper limit on each P4 device by trial and error. We start by trying to add the theoretical maximum number of rules to the P4 table of each P4 device and checking if the device operates successfully. If this was not the case, we cut the number of added rules to half and try again until finding the operable maximum number of rules that could be supported by a P4 device. This limit is identified for each P4 device when performing exact and wildcard matching. In the wildcard matching case, the LPM is used for all devices except for NetFPGA-SUME which only supports ternary matching.

Starting with NetFPGA-SUME, this P4 device uses the Xilinx P4-SDNet tool for compiling P4 programs. Its documentation specifies the theoretical maximum number of rules for different match types [115]. When trying to set the table size to a value larger than this limit, a compilation error is reported. Although the compilation process was successful for this theoretical table size, the operable limit was less than that. For the exact matching case, when using the theoretical limit of 512k rules, the synthesis of this program fails as it could not allocate sufficient memory resources, specifically RAM, to support storing this large number of rules. For 256k and 128k rules, the program's synthesis passes successfully, however, the control plane application fails in interacting with the generated design when trying to add the routing rules. The reason for this could be the high utilization of the RAM resources on the FPGA as this reached around 90%. The next trial of 64k rules works successfully, and thus it is selected as the operable limit for exact matching on the NetFPGA-SUME case. In the case of wildcard matching, we found that the operable limit is the same as the documented 4k rules theoretical limit.

In the case of Agilio CX SmartNIC, although the documentation says that the theoretical maximum number of rules can be up to 64k rules [101], the P4 program compiles even when the table size is set to take values as large as 50 million. However, a loading error takes place when trying to install more than 48k rules to the SmartNIC in both exact and wildcard matching cases. So we select 48k rules to be the operable limit in the evaluation while 64k is the theoretical limit. It is also observed that the maximum number of supported rules depends on the size of the rule entry, i.e., the number and size of matching fields and action parameters. It seems that there is limited memory space for storing the rules in the SmartNIC and this space fills up based on the product of the number of table entries with the size of each entry.

Table 3.3.: A summary of the theoretical and operable limit on the maximum number of rules used in the flow scalability evaluation for the three investigated P4 targets. The symbol (=) denotes exact matching, while (*) symbol denotes wildcard matching.

P4 Target	Match Type	Theoretical Max. Rules	Operable Max. Rules
NetFPGA	=	512k	64k
	*	4k	4k
Agilio CX	= and *	64k	48k
t4p4s	= and *	1k	1k

In the case of the t4p4s software switch, we found that the limit on the maximum number of rules that could be installed into a P4 table is hardcoded to be 1024 rules in the source code of this software switch in the "HASH_ENTRIES" and "LPM_MAX_RULES" constants in "dpdk_tables.h" [114]. This makes the theoretical and operable limit for this software switch equal to 1024 rules. Note that it is observed that although t4p4s allows setting the table size in the P4 program to larger values without compilation or execution errors, it only applies the last 1024 rules when performing the packet forwarding. Moreover, we note that while the hardcoded limit on the number of rules could be changed to larger values in the source code of t4p4s, we decide to stick to evaluating the default implementation available in the open-sourced repository.

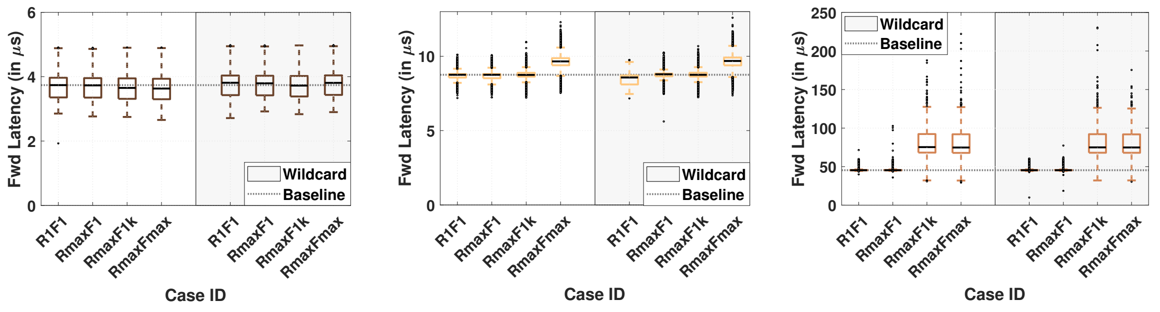
Table 3.3 summarizes the theoretical and operable limit on the maximum number of rules that could be used on different P4 targets. The operable maximum number of rules is used in the flow scalability analysis to configure the maximum size of the P4 routing table and to generate a corresponding number of distinct flows.

3.3.2. Evaluation

The results of the flow scalability experiment described in Subsection 3.3.1 are plotted and interpreted in this Subsection.

Fig. 3.10 shows the box plots of the measured forwarding latency on the three investigated P4 targets in μs for the different examined cases when the packet size is equal to MTU. Results corresponding to the wildcard matching case are shaded in grey.

The median of the forwarding latency for the baseline case R_1F_1 is found to be equal to 3.74, 8.7, and 45.4 μs on NetFPGA-SUME, Agilio CX SmartNIC, and t4p4s software switch, respectively. When the number of added rules increases to the maximum operable value of each device in case $R_{max}F_1$, the measured forwarding latency stays the same. However, when the number of incoming flows increases in cases $R_{max}F_{1k}$ and $R_{max}F_{max}$, we observe that while the forwarding latency stays invariant in the case of NetFPGA-SUME, it slightly increases in the case of Agilio CX SmartNIC by 1 μs



(a) NetFPGA-SUME (b) Agilio CX SmartNIC (c) t4p4s switch

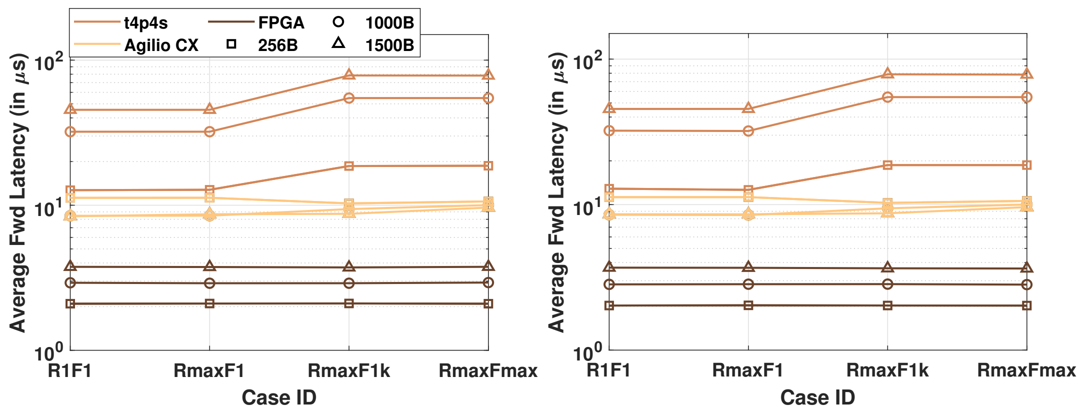
Figure 3.10.: Measured forwarding latency of different flow scalability cases for MTU-sized packets.

and sharply increases in the case of t4p4s software switch by around 30 μs. The results corresponding to the wildcard matching case show the same impact of flow scalability on the packet forwarding latency for all three investigated P4 targets.

These results reveal that NetFPGA-SUME is well designed for accommodating scaled traffic without compromising on the high forwarding performance. Agilio CX shows a similar high performance while dealing with a scaled-up number of flows. The slight increase in latency could be due to the increased lookup time when accessing per-flow cached information. Unlike the latter two optimized hardware P4 targets, the t4p4s software switch showed a weak scaling behavior when the number of incoming flows increased. This is because t4p4s is a software switch running on a general-purpose server, which is not optimized for packet processing purposes, and thus can hit different memory bottlenecks when executing extensive lookup operations. Note that the two cases $R_{max}F_{1k}$ and $R_{max}F_{max}$ are equivalent for t4p4s, because the operable maximum number of rules, and thus generated flows, is equal to 1k in this case as highlighted in Table 3.3.

The average forwarding latency, in μs and logarithmic scale, when running the previously defined flow scalability cases on different P4 targets with packet sizes equal to 256, 1000, and 1500 Bytes is shown in Fig. 3.11a and 3.11b for exact and wildcard matching cases, respectively. These figures again show that NetFPGA-SUME and Agilio CX SmartNIC have a stable performance when processing a scaled number of flows, unlike the case of the t4p4s software switch.

The impact of packet size on shifting the forwarding latency of different P4 targets analyzed in Subsection 3.2.2 still holds here. It is notable that the forwarding latency slightly decreases when the number of incoming flows scales up in the case of Agilio CX SmartNIC for 256 Bytes-sized packets. This unexpected behavior could be due to some optimization techniques implemented on this card, which is activated when the number of lookup operations increases largely, as in this case when the number of distinct incoming flows and the packet rate (for small sized-packets) increase.



(a) Exact Matching (b) Wild Matching

Figure 3.11.: Average forwarding latency of different flow scalability cases for different targets, packet sizes, and types of matching.

3.3.3. Summary

In this section, we investigated the effect of scaling up the number of incoming flows on the forwarding latency of three state-of-the-art P4 targets. In general, the results revealed that while hardware-based P4 targets such as NetFPGA-SUME and Agilio CX SmartNIC could handle traffic with a high number of distinct flows without compromising the forwarding performance, this is not the case for the software DPDK-based t4p4s switch.

3.4. Evaluating Responsiveness of Control Agent

In this experiment, we focus on evaluating the responsiveness of P4 devices to control plane commands. P4 devices interact with SDN controllers through the Control Agent. This Control Agent lies in the P4 device on top of the packet forwarding pipeline. on one hand, it communicates with the SDN controller and decodes its messages. On the other hand, it changes the state of the packet forwarding pipeline accordingly by adding table rules, etc.

In this study, named rule update responsiveness, we target finding the delay needed by the packet processing pipeline to respond to control plane commands. Identifying this delay is critical, as it reveals the period over which the control and data planes are in an inconsistent state, which can accordingly result in vague forwarding behavior. The contributions presented in this section are based on our publication [3]. In Subsection 3.4.1 we describe the designed experiments, while in Subsection 3.4.2 we plot and analyze the collected results. In Subsection 3.4.3, we summarize the findings of this section.

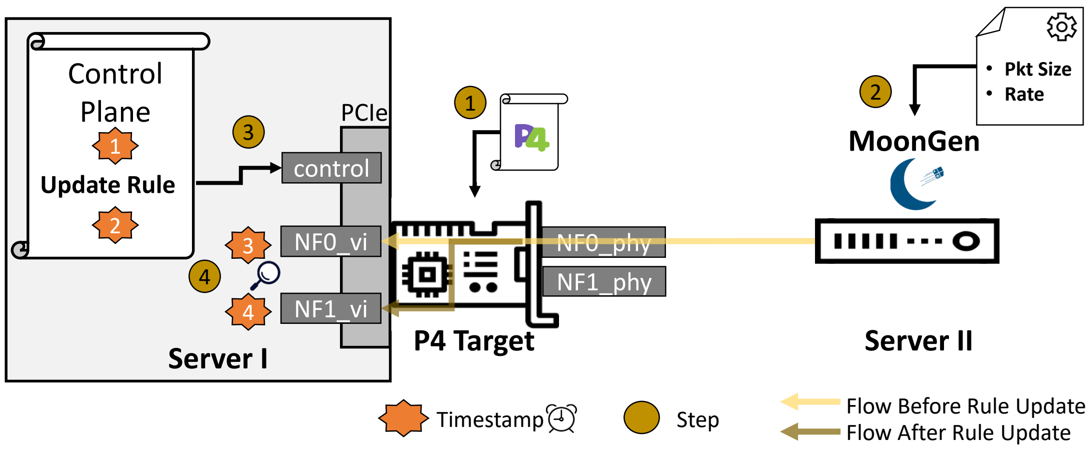


Figure 3.12.: A testbed built for conducting the rule update responsiveness analysis. Server I hosts the P4 target and the control plane, while Server II runs MoonGen packet generator. Multiple timestamps are collected at different stages of the experiment to calculate the delays related to the responsiveness of the P4 devices to control plane update rules.

3.4.1. Testbed and Experiment Design

In the following, we describe the testbed used for conducting this experiment, as well as the different scenarios considered in this evaluation.

The two Nokia NDCS16RM AirFrame Compute Nodes described in Subsection 3.2.1 are used to build the setup shown in Fig. 3.12. Only the two hardware P4 targets, i.e., NetFPGA-SUME and Agilio CX SmartNIC, are evaluated in this experiment. The t4p4s software switch is excluded because its current implementation of the Control Agent is still experimental and does not fully allow adding rules to the switch at runtime. The two hardware P4 targets are plugged into the Peripheral Component Interconnect Express (PCIe) bus of Server I, which also runs the Control Plane software. These two hardware P4 targets have physical interfaces to connect with other machines, and virtual interfaces exposed to the operating system of the hosting machine. We connect MoonGen to one of the physical interfaces, i.e., *NF0_phy*, of the P4 target under investigation. The experiment is conducted through the following steps:

1. The L3Fwd P4 program is loaded to the investigated P4 target, along with a routing rule that forwards packets coming from one physical interface, i.e *NF0_phy*, to another virtual interface, i.e., *NF0_vir*.

2. MoonGen starts sending traffic over the link connected to the physical interface *NF0_phy* of the investigated P4 target. The rate and packet size of this generated traffic is configurable.

3. The control plane program running on Server I generates and sends an updated

forwarding rule to the P4 target over the control channel. The updated rule changes the egress port corresponding to the packets received on *NF0_phy* to let them leave on virtual interface 1, i.e., *NF1_vir*, instead of *NF0_vir*. Two timestamps **T1** and **T2** are captured directly before and after sending the update rule in the control plane application.

4. The traffic leaving on the two virtual interfaces *NF0_vir* and *NF1_vir* is monitored using Tcpdump [116]. A timestamp **T3** is taken for the last packet received on interface *NF0_vir* before the update rule changes the egress port. Another timestamp **T4** is taken for the first packet received on interface *NF1_vir* after the update rule takes effect in changing the forwarding path.

The collected timestamps will be used to measure relevant metrics for analyzing the responsiveness of P4 targets to control plane update rules. Note that to avoid time synchronization issues, we ensure that all timestamps are obtained from the same server. Each test case is run 20 times to make confidence in the measured timestamps **T1**, **T2**, **T3**, **T4**. The default tools provided for each P4 target are used to run the control plane and to issue the update rules. In the case of NetFPGA-SUME, a provided python API that includes different relevant libraries is used for pushing the control plane update rule to the P4 data path, while in the case of Agilio CX SmartNIC, the executable program "rtecli" is used for this purpose. Note that in the case of NetFPGA-SUME, we could take one more timestamp just before the last line of code responsible for pushing the control plane update rule to quantify the pre-processing delay in the tool. This was possible because the source code of the NetFPGA-SUME's python API was unrestricted, unlike the case of Agilio CX which provides the "rtecli" tool as an executable program.

Experimental Scenarios

In this experiment, we vary different parameters to study the impact of each on the responsiveness of P4 devices to control plane commands. In each evaluated case, we measure the following two metrics:

- $Response\_Time = T4 - T1$: This is the period from issuing the control plane command until this command takes effect on the data plane forwarding behavior. Note that during this time period, the control and data planes have inconsistent states, which may result in out-of-date forwarding behavior.

- $Update\_Rate = (T2 - T1)^{-1}$: This value defines the maximum rate at which a control plane can issue and send update rules to the data plane. It is calculated based on the execution time required for issuing a single update rule command, i.e., $(T2 - T1)$.

Three different P4 pipelines, each requiring a different type of update rules, are investi-

Table 3.4.: A summary of the variables and metrics considered in the rule update responsiveness experiments.

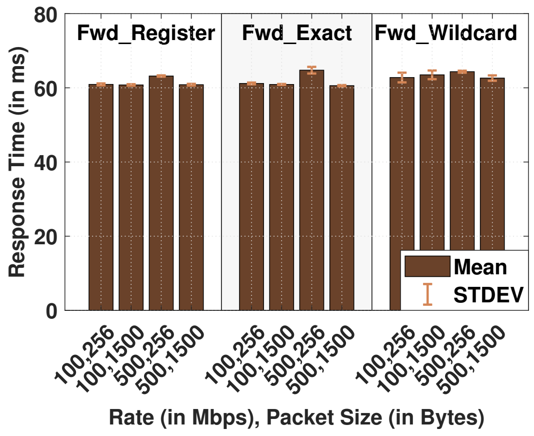
Varied Parameter	**Value**
P4 Targets	NetFPGA-SUME, Agilio CX SmartNIC
P4 Pipelines	Fwd_Exact, Fwd_Wildcard, Fwd_Register
Packet Size (in Bytes)	256, 1000, 1500
Rate (in Mbps)	100, 250, 500/ 6000
Evaluated Metrics	Update_Rate, Response_Time

gated in this experiment. The three pipelines perform packet forwarding functionality. The first pipeline, named Fwd_Exact, forwards packets based on an **exact** match-action table, where the corresponding update rule modifies the egress port from *NF0_vir* to *NF1_vir*. The second pipeline, named Fwd_Wildcard, performs similar functionality but uses a wildcard match-action table and type of update rule. The third pipeline, named Fwd_Register, forwards packets to an egress port whose value is read from a stateful register, where this stored value changes according to the received update rule.

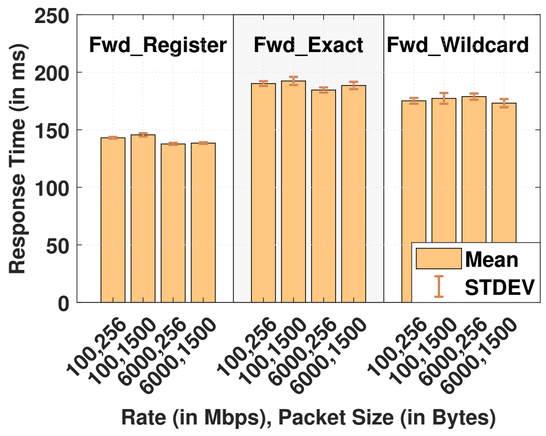
We also varied the configurations of generated traffic throughout the experiment. The size of generated packets is set to take small, average, and large values equal to 256, 1000, and 1500 Bytes, respectively. Additionally, we varied the rate of incoming traffic to the P4 targets. While the physical interfaces can handle up to 10 Gbps traffic rate, this rate is limited to lower values when packets are forwarded to the virtual interfaces through the PCIe bus. We found that the maximum supported throughput that can be forwarded to virtual interfaces in the case of NetFPGA-SUME is limited to 500 Mbps. On the other hand, Agilio CX SmartNIC could forward up to 6000 Mbps traffic of small-sized packets. Therefore, we select these two rates as the maximum rates to be evaluated for each device, besides 250, and 100 Mbps as average and low-loaded cases, respectively. A summary of all varied parameters and evaluated metrics in this experiment is provided in Table 3.4.

3.4.2. Evaluation

In this subsection, we present the results corresponding to the rule update responsiveness experiment. Figures 3.13a and 3.13a show the average response time, in μs, measured on NetFPGA-SUME and Agilio CX SmartNIC, respectively. The results are plotted when different combinations of traffic load, i.e., rate and packet size, are generated, and when different types of update rules are issued by the control plane. The results collected on NetFPGA-SUME show that the response time stays constant in a range between 61 and 65 ms disregarding the variation of traffic load or type of update rule. On the contrary, the Agilio CX SmartNIC results show that the type of update rule impacts the data plane's response time, while the traffic characteristics do not. The recorded response time is higher than that measured on NetFPGA-SUME,



(a) NetFPGA-SUME



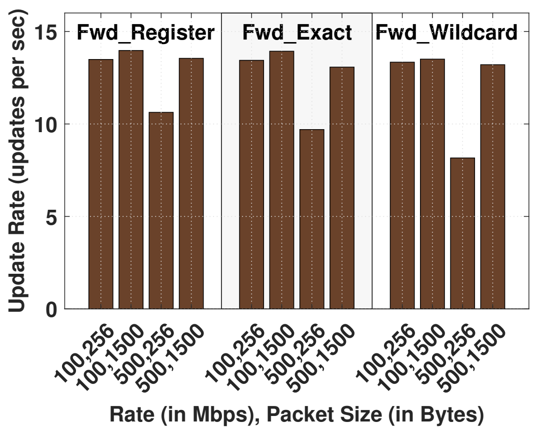
(b) Agilio CX SmartNIC

Figure 3.13.: Measured average response time corresponding to different traffic characteristics and types of control plane rules.

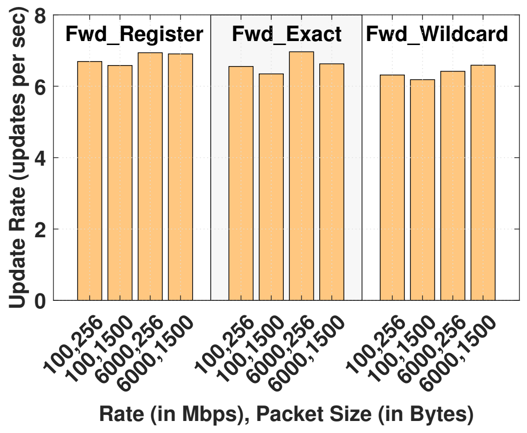
where it varies between 137 to 145 ms, 185 to 192 ms, and 173 to 179 ms when the type of the issued control plane rule is register update, exact matching, and wildcard matching, respectively.

In all the evaluated cases, we observe that the traffic characteristics have a minimal impact on the measured response time not exceeding 10 ms. The same is recorded when setting packet size and traffic rate to average values, i.e., 1000 Bytes and 256 Mbps, respectively. For this reason, we skip plotting the combinations corresponding to these cases. The standard deviation of the 20 runs of each evaluated scenario is always less than 3 ms and 10 ms in the case of NetFPGA-SUME and Agilio CX SmartNIC, respectively. It should be highlighted that the pre-processing of control plane commands that takes place in the control plane tools contributes to a major part of the measured delay in the response time. This pre-processing delay is found to be reaching up to 60 ms when measured on NetFPGA-SUME.

The evaluated average update rate measured on the two devices for different traffic characteristics and types of control plane update rules is presented in Fig. 3.14. The update rate in the case of NetFPGA-SUME, shown in Fig. 3.14a, varies between 8 and 14 updates per second. This rate is invariant when the type of update rule changes. However, it varies when the incoming traffic is at a high packet rate, i.e., when the bitrate is high at 500 Mbps and the packet size is small at 256 Bytes. In this case, the update rate drops to 8 updates per second because of a probable processing bottleneck taking place during the interaction of the NetFPGA-SUME with the PCIe bus of the hosting machine. On the other hand, the update rate recorded on Agilio CX SmartNIC, depicted in Fig. 3.14b, shows invariant results when traffic characteristics or type of update rules change. This update rate is smaller than that recorded on NetFPGA-SUME, where it varies between 6.2 and 7 updates per second.



(a) NetFPGA-SUME



(b) Agilio CX SmartNIC

Figure 3.14.: Measured average update rate corresponding to different traffic characteristics and types of control plane rules.

Note that we removed some of the measured data points when they are identified as outliers with a very large distance from the median value of the other collected measurement results. These removed data points are considered measurement noise and they are less than 2.5 % of the total number of collected measurement results.

3.4.3. Summary

In this section, we investigated the responsiveness of two state-of-the-art P4 targets to control plane update rules. Our evaluation revealed that the response time to control plane commands is in the millisecond range, which is three orders of magnitude larger than the data plane's forwarding latency. This response time varies for different P4 targets, where we observed that NetFPGA-SUME reacts faster than Agilio CX SmartNIC to control plane update rules. A big part of the response time is spent in the preprocessing of the control plane commands that take place in the given devices' toolchains.

3.5. Evaluating P4Runtime Controllers

The controller is a keen component in the SDN architecture. It is the layer that guarantees smooth communication between the control applications via the northbound interface and the data planes via the southbound interface. It is involved in many procedures for managing and controlling switches in the forwarding plane. Therefore, it is critical to evaluate this controller and its interaction with the data plane to understand its performance and limitations as these will be inherited by the overall SDN system. Additionally, it is crucial to push the controller to edge cases by increasing the size of the network to be controlled or by generating a high traffic load to be handled to understand the existing bottlenecks in the controller's implementation and to capture any unexpected behaviors.

To control P4-based data planes, the controller needs to support P4RT framework as a southbound protocol. ONOS controller, illustrated in Subsection 2.2.2, is one of the few controllers that currently have a stable implementation of P4RT framework. Although there are plenty of works in literature that evaluate the performance of OF-based SDN controllers, there is none yet that studies P4RT-based ones. In this section, we fill this gap by proposing a new benchmarking tool, called P4Runtime Controller Probe (P4RCProbe), for evaluating the performance of P4RT-based SDN controllers. The contributions presented in this section are based on our publication [6]. The design and implementation of this tool are described in Subsection 3.5.1. In Subsection 3.5.2, we use this tool to evaluate the performance of the ONOS controller when running in P4RT mode and compare the results to that collected when using ONOS in OF mode, where the latter serves as a baseline case. Finally, in Subsection 3.5.3, we elaborate on the procedure applied using P4RCProbe to identify bottlenecks in ONOS-P4RT implementation, and we describe the code patch proposed for mitigating these bottlenecks, which resulted in 17% overall improvement in the packet rate successfully handled by ONOS. In Subsection 3.5.4, we summarize the findings of this section.

3.5.1. P4RCProbe: Design and Implementation

In this subsection, we elaborate on the design and implementation of P4RCProbe, a novel benchmarking tool for P4RT-based SDN controllers.

As the literature contains many open-source benchmarking tools for OF controllers, we choose to build on top of one of these tools to utilize its mature development stage. We extend the OFCProbe tool [20], which was originally designed to benchmark OF controllers, to support benchmarking P4RT controllers. OFCProbe's modular design allows for the easy addition/integration of P4RT-enabler modules. In Fig. 3.15, we depict the original architecture of OFCProbe highlighting the modified or added modules to support benchmarking P4RT controllers.

The tool basically instantiates virtual (dummy) switches that connect to the SDN controller. When benchmarking starts, the tool triggers these switches to send Packet-In messages to the controller and receive back Packet-Out messages from it, while recording statistics related to this communication such as packet rate, Round-trip Time (RTT), etc. The Communication Layer of the tool initiates and manages the connection channels between the virtual switches and the SDN controller. On the other hand, the Traffic Generation Layer takes care of managing the events related to the life

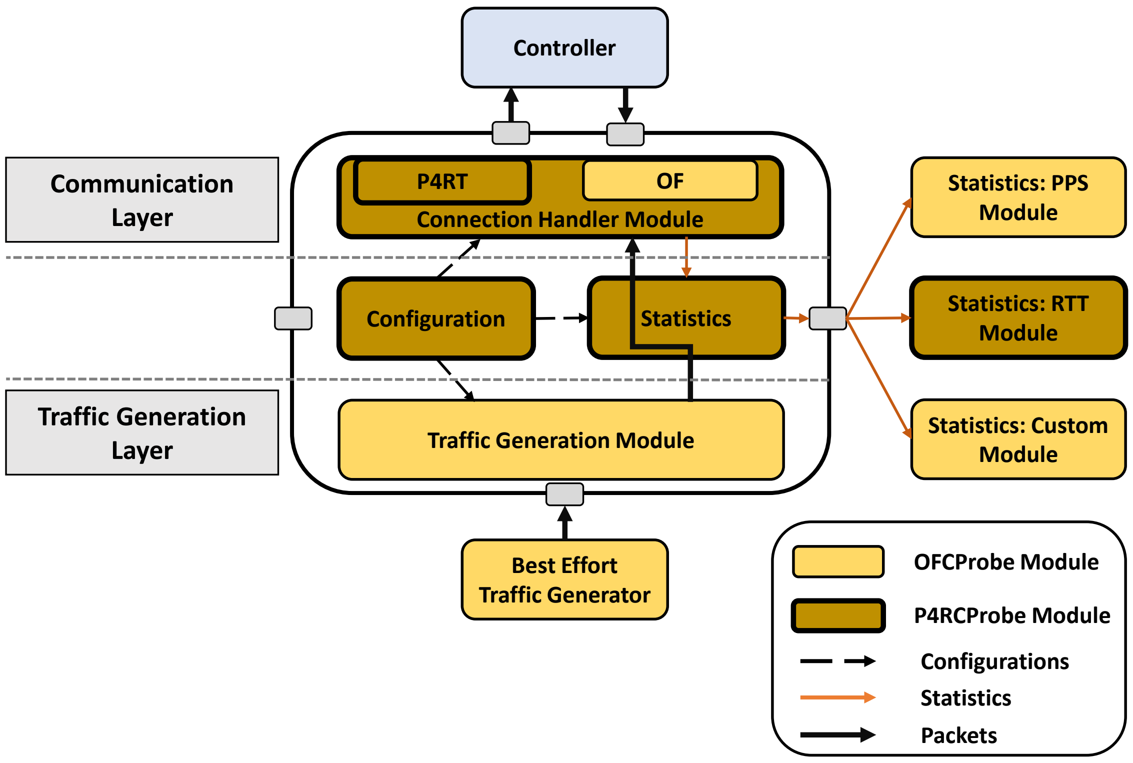


Figure 3.15.: Architecture of P4RCProbe, where modified/ added modules with respect to OFCProbe are highlighted.

cycle of packets to be sent by the virtual switches such as packets' generation, queuing, scheduling, handling, etc. The implementation details of P4RCProbe, especially those related to the modified or added modules, are described in the following.

Connection Handler Module

Unlike OF controllers that use OF protocol to communicate with data planes, P4RT-based controllers use P4RT framework for this purpose. P4RT framework is based on Remote Procedure Call (RPC)s where gRPC and Protocol Buffers are used. To support the two types of southbound communications, we implement a parent connection handler module class, which can be inherited by both OF and P4RT connection handlers. The P4RT connection handler can instantiate a gRPC server stub based on the P4Runtime protocol buffer file "P4Runtime.proto" [94] for each emulated switch. This gRPC server stub waits for the connection to be initiated by the gRPC client stub that runs in the P4RT-based controller under investigation. For each virtual switch-controller communication channel, the following five RPCs are established and managed:

- **Write RPC**: This is a unidirectional RPC used by the controller to update/ write one or more P4 entities on the P4 target.

- **Read RPC**: This is a unidirectional RPC used by the controller to read one or more P4 entities from the P4 target.
- **SetForwardingPipelineConfig RPC**: This is a unidirectional RPC used by the controller to push the P4 pipeline's configuration file to the P4 target.
- **GetForwardingPipelineConfig RPC**: This is a unidirectional RPC used by the controller to retrieve the configuration file of the currently running P4 forwarding-pipeline on the P4 target. Note that the"Cookie field" is one of the fields to be filled in the messages communicated in this RPC. This field is used to uniquely identify a P4 forwarding pipeline and is calculated as the hash function of the configuration file of that particular P4 forwarding pipeline. The tool fills this field with the value extracted from the same field of the SetForwardingPipelineConfig message to emulate for the controller that all the instantiated switches run a particular P4 forwarding pipeline.
- **StreamChannel RPC**: This is a bidirectional stream initiated by the controller and connects to the P4 target. Besides initiating a connection through client arbitration, checking the session liveliness of a switch, and streaming notifications from the switch, this stream channel is mainly used for sending/ receiving Packet-In/ Packet-Out messages between the switch and the controller. After the beginning of the benchmarking procedure, P4RCProbe starts sending Packet-In messages over this stream channel with Transmission Control Protocol (TCP)-Synchronize (SYN) set as a payload. The metadata information of the packets such as the ingress port and padding is set to be compatible with the predefined Packet-In headers in the used P4 pipeline at the data plane. The tool decodes the Packet-Out responses received over this channel to update collected statistics.

If P4RCProbe is configured to emulate multiple virtual switches, it instantiates a gRPC server stub for each switch as depicted in Fig. 3.16. These stubs operate independently and communicate with the same client stub running in the controller.

Configuration Module

The tool takes as input a configuration file with the different parameters that can be used to configure the modules of the tool to customize the benchmarking procedure according to the desired evaluation scenario. These parameters include the type of the controller to be benchmarked (whether OF or P4RT), the number of virtual switches to be emulated, the packet rate to be generated and sent by each emulated switch, the statistics to be tracked, etc.

If the runtime protocol is set to P4RT, then the controller needs to know the forwarding

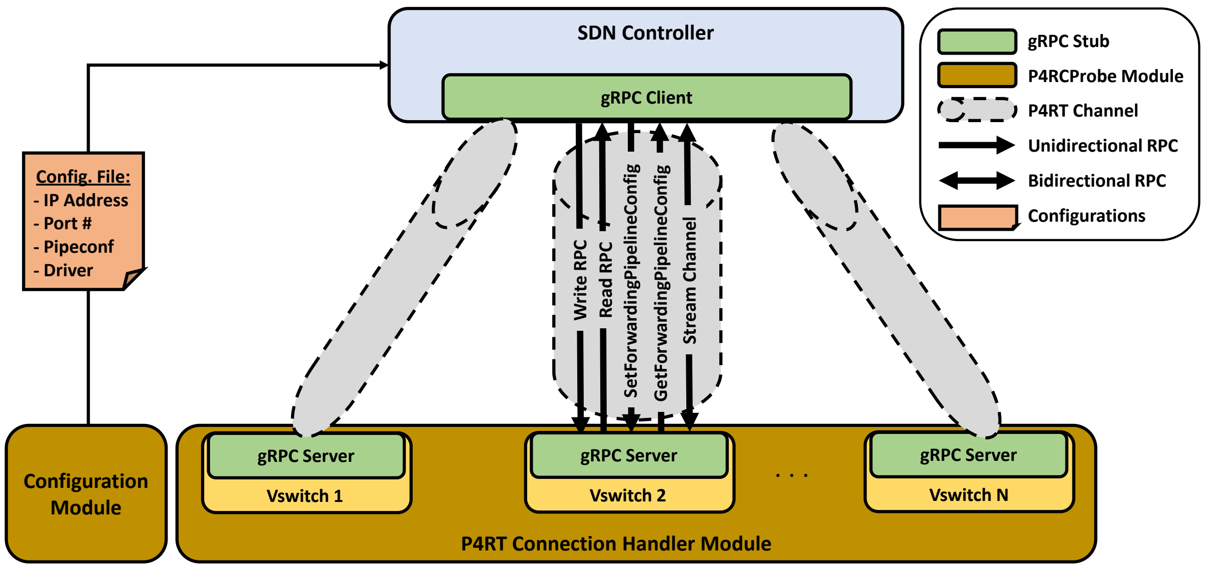


Figure 3.16.: Communication channels established between P4RCProbe and the P4RT-based controller under investigation.

pipeline configured on the P4 target's data plane. Our benchmarking tool P4RCProbe automatically creates and sends a configuration file to the controller with all the necessary fields for initializing the connection. This file includes information related to the selected forwarding pipeline configuration file, the driver of the device at the data plane (e.g. BMv2), IDs, IP addresses, and ports of the different emulated virtual switches. It is the role of the controller to initiate the connection with the virtual switches emulated by P4RCProbe since the gRPC client stub runs on the controller in P4RT framework. Thus it is important to pass this configuration file to the controller before starting the benchmarking procedure.

Statistics Module

The RTT statistics module is modified to be compatible with P4RT mode. Initially, this module takes two timestamps: one when the Packet-In message leaves P4RCProbe and another one when the corresponding Packet-Out response is received, where RTT is calculated as the difference between these two timestamps. While mapping Packet-Out responses to their Packet-In requests can be easily done in OF mode using the "Transaction ID" field defined in the OF protocol, this is not possible in P4RT mode. Instead, for RTT calculation, we need to include an artificial ID in the payload of the randomly generated TCP SYN Packet-In messages to track their Packet-Out responses.

The interactions and communication channels between P4RCProbe and the SDN controller are shown in Fig. 3.16. The source code of P4RCProbe is made publicly available in [1].

3.5.2. Evaluation

In this subsection, P4RCProbe is used to benchmark the performance of the ONOS 2.5.0 release controller [109] when running in P4RT mode. The performance of ONOS when running in OF mode is also evaluated to serve as a baseline when compared to the performance of the P4RT-based mode.

The evaluation is conducted using a testbed made up of two connected machines each running Ubuntu 18.04, has a CPU with 4 cores (Intel i5-4670 at 3.40GHz), and is equipped with two 10 Gbps Ethernet ports. The benchmarking tool P4RCProbe runs on one machine while the ONOS controller under test runs on the other machine. The open-source pipeconf project "P4tutorial", published on the ONOS GitHub page, is used as the P4 data plane in this evaluation. This project contains a P4 program that describes basic forwarding and tunneling functions. It also contains the configuration files and the interpreter file that enables ONOS to understand the specific constructs of the given "P4tutorial" program. This P4 program, like OF switches, is compatible with the "Reactive Forwarding" control application available in ONOS, which is developed to interact with the data plane to control packet forwarding. Using the same control application, i.e., Reactive Forwarding, is important to guarantee a fair comparison of the performance of ONOS when running in P4RT vs OF mode. To plot confidence intervals, we repeat each measurement 5 times, wherein ONOS service is restarted between consecutive runs.

We start by evaluating the case when a single switch connects to ONOS to understand the performance limitations of this case. Then, we extend the evaluation scenario to the cases when multiple switches connect to ONOS.

Single Switch

In this first evaluation scenario, we target understanding the performance of ONOS when handling incoming packet streams with increasing packet rates. For this purpose, we configure P4RCProbe to emulate a single P4RT switch that connects to ONOS when running in P4RT mode. Then, we run different measurements when this switch starts sending Packet-In messages to ONOS while gradually varying the configured packet rate. The same evaluation is conducted when ONOS runs in OF mode.

Fig. 3.17 shows the results of the sent and received packet rate when a single switch communicates with the ONOS controller. In Fig. 3.17a, we plot the average recorded rate of sent Packet-In messages to ONOS running in OF mode and the average rate

[1]https://github.com/tum-lkn/P4RCProbe

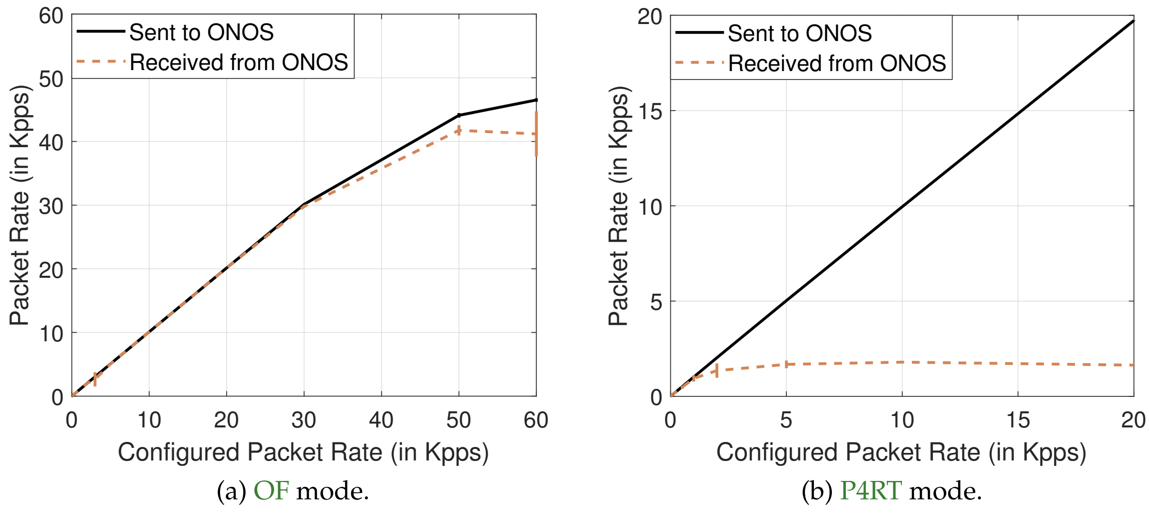


Figure 3.17.: Packet rate sent to and received from ONOS when controlling a single switch.

of Packet-out messages received from ONOS when varying the configured sending rate in P4RCProbe. It is observed that ONOS in OF mode can process and respond to around 30K pps rate of incoming Packet-In messages, after which he starts dropping some incoming packets before saturating at 42K pps. On the other hand, the results corresponding to ONOS running in P4RT mode, depicted in Fig. 3.17b, show that only around 1.5K pps can be handled successfully.

The core utilization of ONOS when running in OF and P4RT modes as a function of an increased incoming packet rate from a single switch is shown in Fig. 3.18. When the configured packet rate is low, it is observed that ONOS running in OF mode consumes less computing resources compared to the case when running in P4RT mode. As the configured packet rate increases, the ONOS processing in OF mode consumes more core resources, while the resources required by ONOS when running in P4RT mode saturates at around 250 % core utilization as soon as the configured packet rate reaches around 1.5K pps. Recalling that ONOS is running on a machine with 4 cores, this makes the maximum achievable core utilization equal to 400 %. This represents the physical hardware processing limit in the testbed, which has never been reached in this evaluation.

As the purpose of this experiment is to find the limits of ONOS when handling an increased rate of incoming packets, we disable the RTT statistics module in P4RCProbe to avoid overloading it with storing timestamps for packets that are dropped at ONOS and thus will never receive a response.

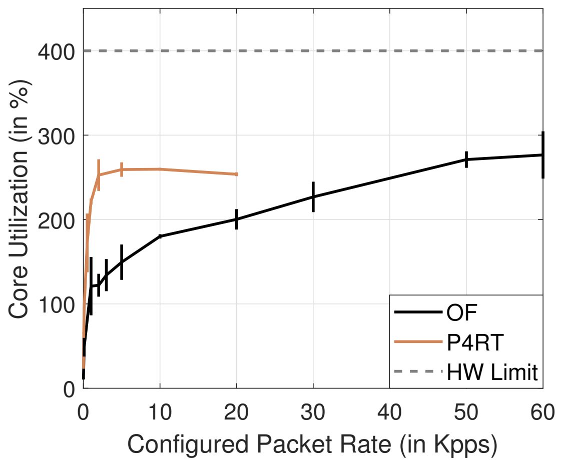


Figure 3.18.: Core utilization of ONOS when controlling a single switch running in OF versus P4RT mode.

Multiple Switch

In this subsection, P4RCProbe is configured to emulate an increasing number of switches to be controlled by ONOS. The evaluation is also conducted when running ONOS in OF and P4RT modes. Based on the results of the previous subsection, the packet rate to be sent by each switch is configured such that the sum of the rates of all switches is around the previously identified 1.5K pps limit.

In Fig. 3.19, we show the overall packet rate sent to and received from ONOS when running in OF versus P4RT mode as a function of the number of emulated switches in the network for different configured packet rates to be sent per switch: 10, 50, and 100 pps. Fig. 3.19a shows that the received packet rate curve is overlapping with the sent packet rate curve for all cases. This means that ONOS running in OF mode can handle all incoming traffic without packet drop for all number of switches to be controlled and per switch packet rate configurations. Looking to Fig. 3.19b, ONOS in P4RT mode can handle packets from all connected switches only when the configured per switch sending packet rate is equal to 10 pps. When the packet rate per switch increases to 50 pps and 100 pps, the received packet rate saturates after 10 and 1 switch connecting, respectively, which means that ONOS can not handle received packets after these stages. We can observe that the performance of ONOS in P4RT mode when controlling multiple switches is bottlenecked when processing a rate less than 1000 pps (20 switches x 50 pps-per-switch), which is lower than the previously identified limit of 1.5k pps in the single switch evaluation. This is interpreted to be due to the extra processing required by ONOS for handling the connections from the multiple connected switches.

The recorded average RTT in ms and in logarithmic scale of packets communicated with ONOS when running in OF and P4RT modes is shown in Fig. 3.20a. The results are

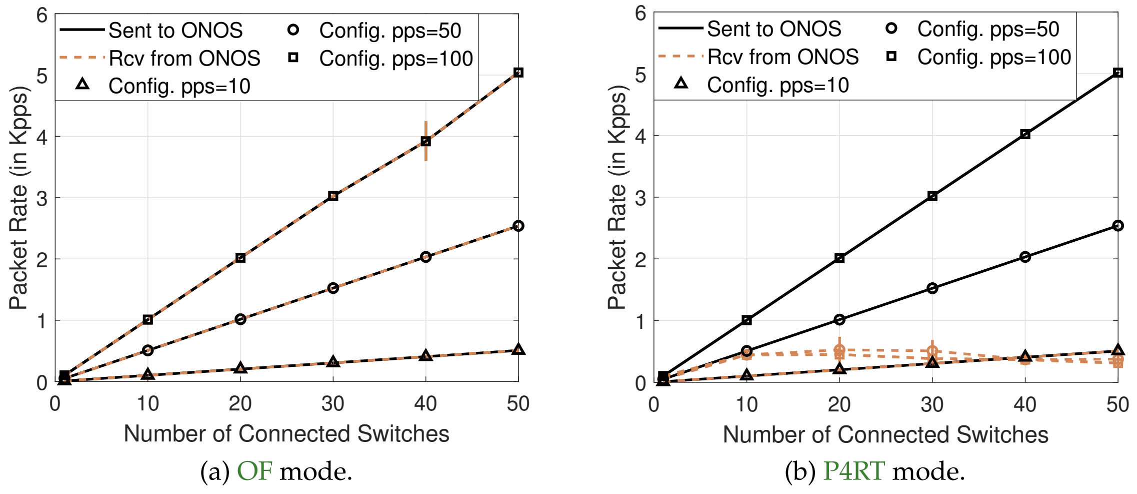
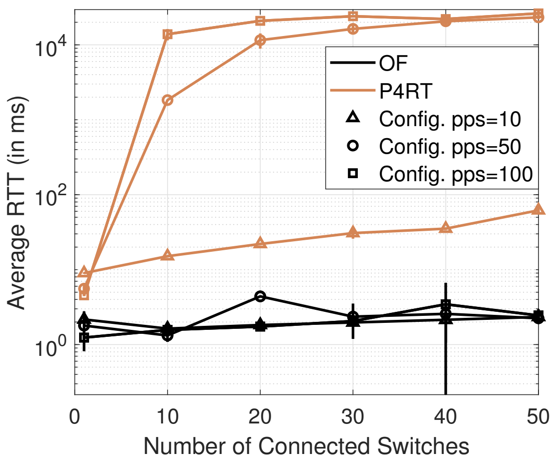


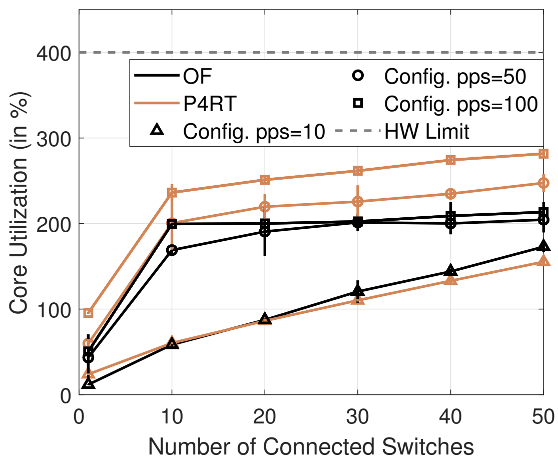
Figure 3.19.: Packet rate sent to and received from ONOS when controlling multiple switches.

shown when the number of emulated switches increases and for different configured per-switch packet rates. The average RTT recorded when ONOS runs in OF mode varies between 1.5 and 4.3 ms with a slightly increasing trend when the number of switches increases. The impact of configured per switch packet rate on the RTT is minimal because the processing load in this evaluation is relatively low for ONOS when running in OF mode, given that it could steadily process up to 30k pps in the single switch evaluation. On the other hand, the P4RT mode results show high average RTT values compared to OF mode. In the stable case when the traffic load is low at 10 pps per switch, where ONOS could handle all incoming packets successfully, the average recorded RTT increases from 8.9 to 61 ms as the number of emulated switches increases to 50. When configured per switch packet rate is set to 50 pps, the average RTT increases to 1.8 seconds in the case of 10 switches, and it reaches around 20 seconds when 50 switches are emulated. When the configured per switch packet rate is set to 100 pps, the average RTT directly reaches 20 seconds when more than 1 switch is emulated.

The utilization of core resources by ONOS when running in OF and P4RT modes as a function of an increasing number of connected switches and for different per switch packet rate configurations is shown in Fig. 3.20b. When the configured packet rate is equal to 10 pps, both P4RT and OF modes require similar core resources, which rise from 10 to 170 % as the number of switches increases to 50. When the configured per switch packet rate increases to 50 and 100 pps, the consumed core resources in both modes increase. In P4RT mode, ONOS requires more core resources compared to OF reaching around 200 and 280 % core utilization at 50 switches for 50 and 100 pps loads, respectively. On the other hand, OF mode requires around 200 % for both 50 and 100 pps loads. Similar to the single switch case, since the core utilization never reaches 400 %, then no hardware limit is reached in this evaluation.



(a) RTT Results

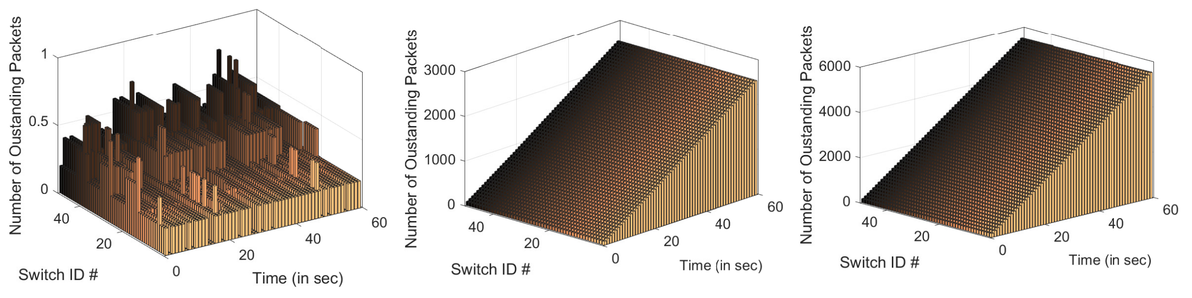


(b) Core Utilization Results

Figure 3.20.: RTT and core utilization results of the multiple switches evaluation.

P4RCProbe can evaluate another interesting metric which is the number of outstanding packets. An outstanding packet is defined as the packet which is sent to ONOS as a Packet-In message but no response was received for it over a given time interval. The number of outstanding packets when ONOS runs in P4RT mode and controls 50 switches for 60 seconds is shown in Fig. 3.21. Fig. 3.21a shows the case when the generated traffic load per switch is low at 10 pps. In this case, the number of recorded outstanding packets is always less than one since ONOS can handle all the incoming low volume of traffic as revealed from Fig. 3.19. The outstanding packets results become more interesting when ONOS can not handle all the incoming packets as shown in Figs. 3.21b and 3.21c when packet rates are set equal to 50 and 100 pps per switch, respectively. In these two cases, the number of outstanding packets increases linearly over time reaching around 2550 packets after 60 seconds for each switch when the packet rate per switch is set equal to 50 pps, and around double that value (5500 packets) when the per switch rate is set equal to 100 pps. The results of these outstanding packets reveal that there is a consistent buffering behavior taking place in ONOS when running in P4RT mode, where packets have to wait before they get their turn to be served. This inferred behavior of ONOS is further supported when we observed that ONOS could respond back to all received packets but after some time (around tens of seconds) without any packet drop.

On the positive side, when looking at the variation of the number of outstanding packets across the switch ID axis, it can be observed that there is a consistent accumulation of outstanding packets across different switches, meaning that ONOS deals with all these switches fairly disregarding the order they are connected in. The results corresponding to OF mode are not plotted since the number of outstanding packets is always less than one since ONOS in OF mode can handle all the incoming packets at these traffic loads as we observed in 3.19a.



(a) Configured packet rate set to 10 pps. (b) Configured packet rate set to 50 pps. (c) Configured packet rate set to 100 pps.

Figure 3.21.: The number of outstanding packets recorded at ONOS when running in P4RT mode for different per switch packet rate configurations.

3.5.3. P4RCProbe in Practice

In this subsection, we showcase how the P4RCProbe tool can be used in practice for identifying bottlenecks in the implementation of SDN controllers. After identifying one performance bottleneck in ONOS when running P4RT mode, we propose a possible enhanced design for solving this bottleneck.

Methodology: Our approach goes as follows. We create an artificial loopback after different processing blocks in ONOS as depicted in Fig. 3.22. Paths P1, P2, and P3 are created by forwarding packets back to P4RCProbe directly after the gRPC Client module, the P4Runtime protocol decoding module, and the reactive forwarding application, respectively. P4RCProbe is used to send 10K pps to be processed at ONOS over each created path, and to measure the received packet rate, i.e the rate of packets successfully processed through the investigated path in ONOS without packet drop. Each measurement case is evaluated five times to calculate the average and standard deviation of the received packet rate and core utilization. The measurement results showed that while the received rate over Path 1 after the gRPC client processing stays at around 10K pps, this rate directly drops down to around 1.8K pps when packets traverse Path 2 which includes P4Runtime protocol module processing. These results clearly indicate that there exists a processing bottleneck in this module. The received packet rate over Path 3, wherein the complete ONOS processing takes place, is measured to be equal to around 1.5K pps.

Problem Analysis: Next, we analyze the method which implements the P4Runtime protocol processing in ONOS source code to identify the reason behind this bottleneck. The implementation of this method retrieves/gets the pipeline configuration object

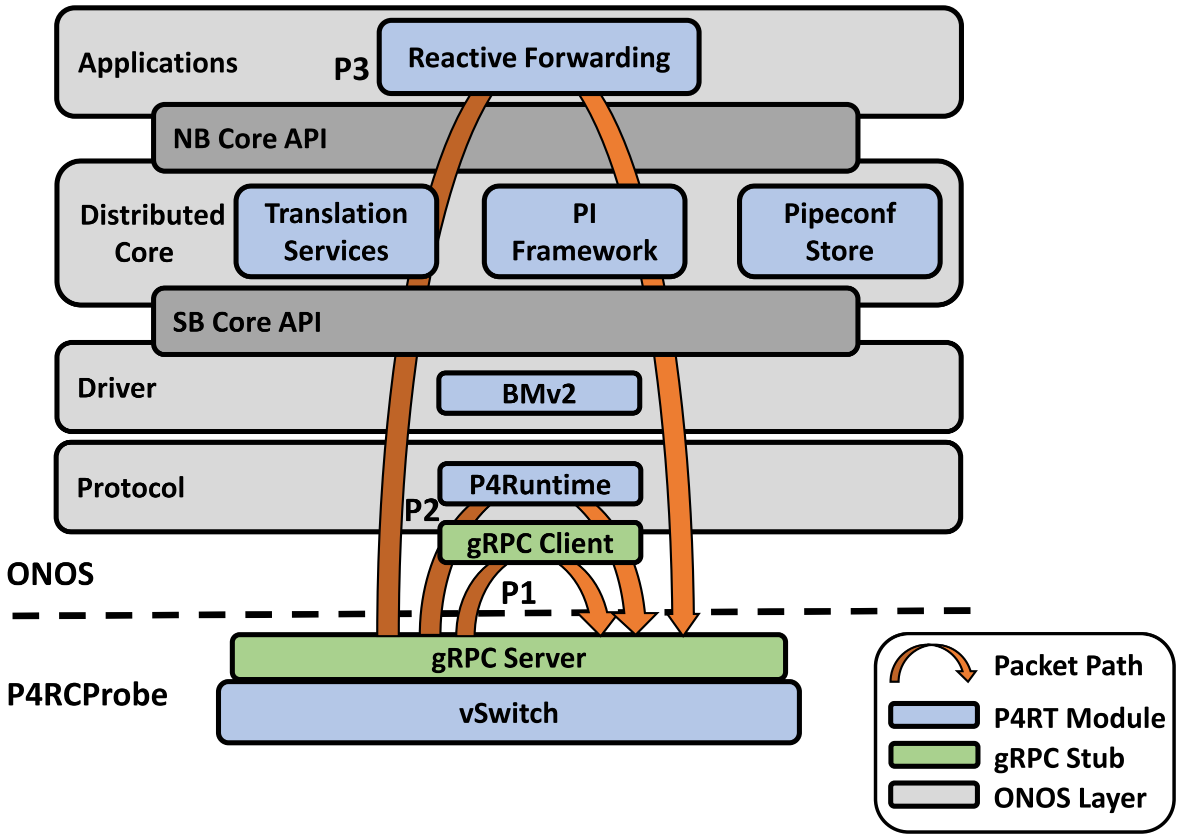


Figure 3.22.: The method applied using P4RCProbe for identifying performance bottlenecks in the implementation of the ONOS controller when running in P4RT mode.

"PipeConf" for every received packet at ONOS. This PipeConf object includes information, some in JSON files, that describes the loaded P4 data plane on the controlled switch. Retrieving or getting this PipeConf object which includes a method for reading input streams from stored files is processing-intensive. This is proven by trying to skip the lines of code corresponding to getting this Pipeconf object, where we observed that in this case the received packet rate increases again to 10K pps. Recalling that ONOS in P4RT mode deals with P4 data planes that can be reconfigured at any time, this implementation of ONOS where the PipeConf object is retrieved for every packet is meant to ensure that each received packet gets processed according to the latest running P4 pipeline on the data plane.

Design Enhancement: One possible design enhancement is suggested and implemented to mitigate this bottleneck. The enhancement is based on the idea that the PipeConf object should not be read for every incoming packet, rather it should be only read when the running P4 data plane changes. When the client gRPC stub is created, the PipeConf object of the initially loaded P4 data plane is retrieved. Then, only if

Table 3.5.: A Summary of the received packet rate and the core utilization measured over different processing paths before and after applying the design enhancement code patch to ONOS.

Metric	Path / Code	P1	P2	P3
Rcv. Packet Rate (in pps)	Original	10030 ± 10	1836 ± 41	1471 ± 10
	Enhanced	10026 ± 8	10050 ± 7	1723 ± 30
Core Utilization (in %)	Original	61 ± 0.2	167 ± 0.7	215 ± 1.6
	Enhanced	60 ± 0.1	66 ± 0.2	206 ± 1.1

the data plane on the P4 device changes, a signal is sent to read the PipeConf object of the updated data plane. The change in the P4 data plane can be detected by the GetForwardingPipelineConfig RPC, which always checks and reads the latest running P4 pipeline. Accordingly, we save ONOS from performing the processing-intensive call for getting the PipeConf object for every packet, where this call will be executed only when a change in the running P4 data plane is identified.

The application of this design enhancement into the source code of ONOS lead to increasing the packet rate that could be handled by the P4Runtime protocol block on Path 2 again to around 10K pps, indicating that this implementation bottleneck is resolved. A summary of the recorded received packet rates and core utilization (averages and standard deviations) for different packet processing paths before and after implementing the design enhancement is provided in Table 3.5. From this table, we can observe that the core utilization results are in line with the packet rate variation. This core utilization decreases from 167 % to 66 % over Path 2 after applying the enhanced code patch, which indicates that the processing load at this stage is relaxed.

It is also observed that over Path 3 with the complete ONOS processing, the packet rate drops again to 1723 pps even with the design enhancement implemented. However, this packet rate is still more than the rate recorded on this path before implementing the design enhancement, which was equal to 1471 pps. This means that although the enhanced design mitigated the processing bottleneck in the P4Runtime protocol block, there are still other bottlenecks at later stages in the ONOS processing pipeline. Nevertheless, solving this implementation bottleneck in the P4Runtime protocol block could improve the overall processing rate of ONOS in P4RT mode by around 17 %. As the purpose of this exercise is to show how our proposed benchmarking tool P4RCProbe can be used in practice for identifying bottlenecks in the design of SDN controllers, we stop at this stage in this analysis since the same approach can be used to solve the other processing bottlenecks in ONOS controller or in any other P4RT-based SDN controller.

3.5.4. Summary

In this section, we evaluate the performance of the ONOS controller when controlling P4 targets using P4RT framework. As we did not find in the literature any benchmarking tool for P4RT-based controllers, we had to first fill this gap by implementing a novel benchmarking tool for this purpose, named P4RCProbe. Then, we conduct a comprehensive evaluation for the ONOS controller when running in P4RT, and compared that to a baseline case when ONOS runs in OF mode. The evaluation revealed that although ONOS in P4RT mode handles switches fairly, it can handle less packet rate and takes longer RTT compared to OF mode. This weaker performance of ONOS when running in P4RT mode presents the cost of controlling reconfigurable data planes as in the case of P4 targets. Finally, we showcased how the tool can be used to identify bottlenecks in the implementation of P4RT based controllers, and we proposed a design enhancement to the ONOS controller to mitigate an identified bottleneck achieving an overall improvement of 17% in the packet processing rate of ONOS. The RTT measurement results collected for the ONOS controller in this section are used as input for characterizing the service time of the SDN controller in a P4-based system in Chapter 4, where we target deriving an analytical model for the packet forwarding latency in P4-based systems.

3.6. Summary

In this chapter, we conducted a comprehensive performance study of the different components that build a P4-based system. While Section 3.5 filled a gap in the literature about evaluating the performance of P4RT-based control of P4 packet processors, Sections 3.2, 3.3, and 3.4 focus on evaluating the performance of P4 data planes. Our measurements revealed the capabilities and limitations of different investigated P4 devices towards advancing the common knowledge on the performance of these programmable packet processors. Table 3.6 compares the performance of different P4 devices according to different relevant criteria. The criteria that were evaluated in this chapter are highlighted in green, while the results of other criteria are surveyed from [32] and [30]. The general pattern recognized from this table is in line with the hardware-software trade-off between flexibility and performance. The summary confirms that when the P4 device is more of a hardware type, its performance prevails compared to software-based devices in terms of throughput, forwarding latency, flow scalability, responsiveness to control plane commands, and jitter. On the other hand, this high performance of hardware-based devices comes at the cost of reduced flexibility in defining new functionalities such as P4 externs, restricted availability of computing resources, and higher price. Note that the price of CPU-based devices is considered the cheapest since P4 software switches can share the same CPU computational resources with other software applications.

Table 3.6.: A comparative summary of different P4 targets that belong to different processing platforms based on various relevant criteria [1, 2, 30, 32]; (.) is used for entries where information is not available. Criteria evaluated in this chapter are highlighted in green.

P4 Device / Criteria	t4p4s *CPU*	Agilio CX *NPU*	NetFPGA-SUME *FPGA*	Tofino *ASIC*
Throughput	+	++	+++	++++
Forwarding Latency	+	++	+++	++++
Flow Scalability	+	++	+++	.
Rule Update Responsiveness	.	++	+++	.
Jitter	+	++	+++	++++
Resources Availability	++++	+++	++	+
Flexibility	++++	+++	++	+
Price	++++	+++	++	+

The measurement results collected in this chapter create the main building block for deriving analytical models for the performance of P4-based systems in the following chapter. Moreover, this performance understanding enables the performance-aware management of P4-based cloud environments in Chapter 5.

4. Modeling the Performance of P4 Programmable Packet Processors

A P4-based system is made up of a data plane and a controller, where packets can be sent in case the forwarding decision is not installed at the data plane. In the previous chapter, we conducted extensive measurements to understand the performance of these two components separately. However, to assess the suitability of P4-based systems for a certain deployment use case scenario, it is important to understand the performance of the complete system, including the different interaction levels between its components. Fig. 4.1 shows a P4-based system with a P4 programmable data plane and P4Runtime-compatible controller. Different factors influence the performance of a P4-based system. These factors include:

- The complexity of the loaded P4 program at the data plane as the processing latency varies based on the amount of packet processing operations to be executed there.
- The type of the P4 device since different P4 devices have different forwarding delays as observed from the measurement results presented in Chapter 3.
- The degree of involvement of the controller. If packets are to be sent to the controller, additional latency should be counted.
- The forwarding delay at the controller. This delay should be added to the overall packets' delay in case the controller is involved in the packet processing.
- The incoming traffic load. It is expected that if the incoming traffic load approaches the line rate of a device, queueing will take place in the device and the packet's waiting time in the queue should be accounted for.

While measurements give exact knowledge about the performance of P4 devices or controllers under test, it is not feasible to do the same for P4-based systems while varying all the previously listed influential parameters since this results in countless combinations. Therefore, in this chapter, we make use of the measurements conducted in Chapter 3 to propose analytical models that can predict the real performance of P4-based systems more generically.

In this direction, we first propose in Section 4.2 a model for predicting the packet forwarding latency at the data plane as a function of the loaded P4 program. This

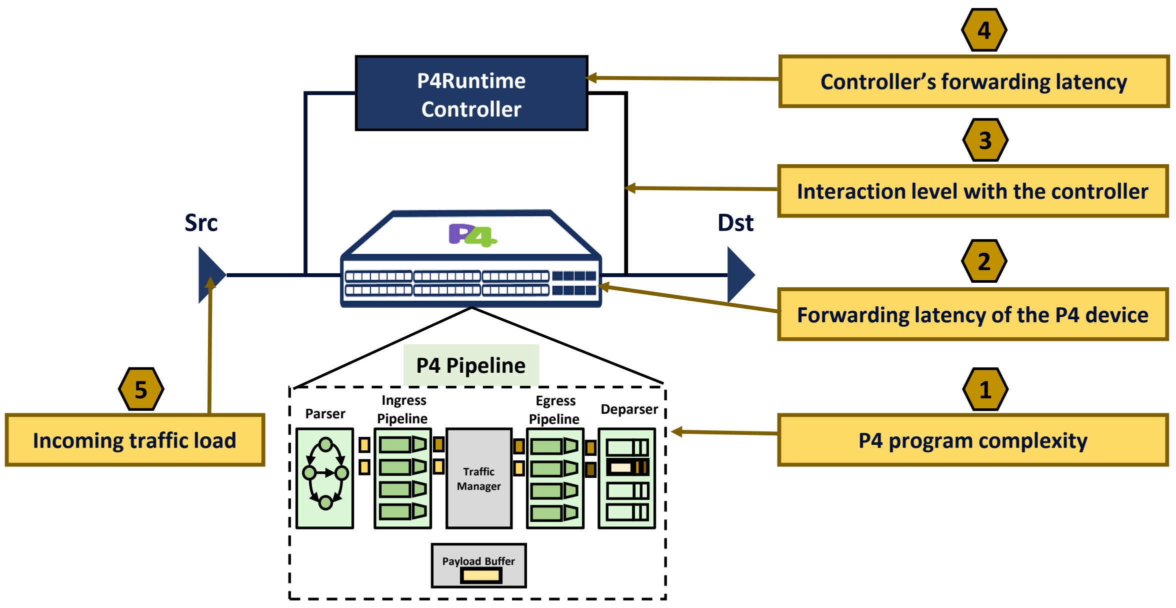


Figure 4.1.: A P4-based system with a programmable data plane and controller with all performance influential factors highlighted.

model is parametrized according to the used P4 target. The model is then verified using measurements. The contributions of this section are based on our previously published work [1].

In Section 4.3, we propose two queueing models for P4-based systems that can predict the average sojourn time in the system based on the previously listed influential factors which can be set to match any deployment configuration in question. The two models make use of the model proposed in Section 4.2 for parametrizing the service time at the data plane based on the used P4 target and the complexity of the loaded P4 program. The models abstract the behavior of the system as a queue at the data plane with a feedback queue system at the controller. The first model assumes memoryless service times (i.e., exponentially distributed) at the data plane and the controller aiming for a simplified version for quick sojourn time calculations. The second model relaxes this assumption, where the service process is assumed to be generic and dependent on the real measured service time at the data plane and controller. The purpose of the second model is to better capture the real performance of P4-based systems. The contributions of this chapter are partially based on our previous publication [5].

In Section 4.4, we conduct a parameter study where we vary a wide range of parameters to verify the accuracy of the two proposed models in Section 4.3 through simulations. Based on this evaluation, we derive a performance constraint for ensuring a stable operation in a P4-based system, where this constraint can guide the dimensioning of the permissible traffic that can be successfully handled by the system. Also, the contributions of this chapter are partially based on our previous publication [5].

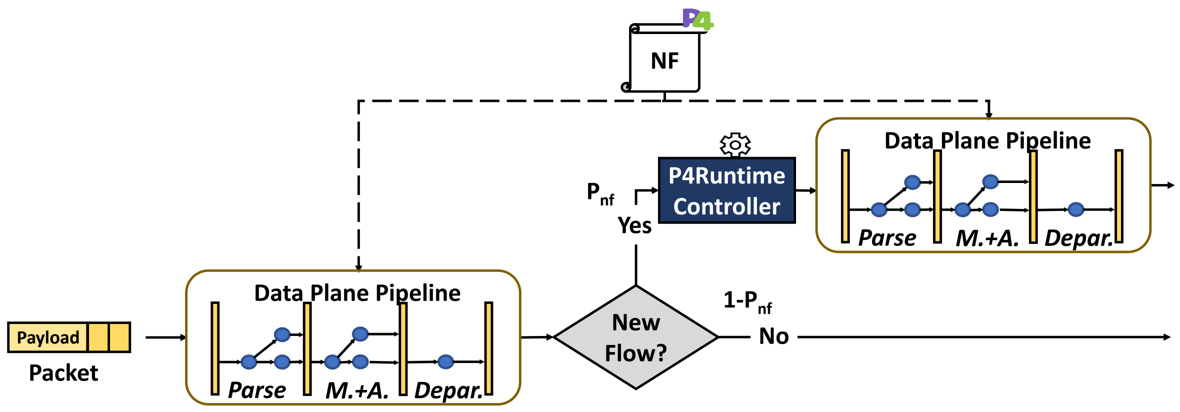


Figure 4.2.: Packet's life cycle in a P4-based system with a programmable data plane and a controller.

The rest of this chapter is organized as follows. A background overview of the packet's journey in P4-based systems is provided in Section 4.1 along with a summary of related works that deal with modeling the performance of P4-based devices. In Section 4.2, we start with modeling the forwarding latency at the data plane based on the loaded P4 program and the used P4 device. Two queueing models are proposed in Section 4.3 for abstracting the performance of complete P4-based systems. These two models are evaluated and analyzed in Section 4.4 towards deriving constraints related to dimensioning the permissible traffic that can be handled in P4-based systems. Finally, Section 4.5 summarizes the contribution of this chapter.

4.1. Background and Related Work

In this section, we first elaborate on the relevant background about the packet's life cycle in P4-based systems in Subsection 4.1.1, and then we describe related works in Subsection 4.1.2.

4.1.1. Packet's Life Cycle in a P4-based System

A P4 program describes the device's data plane/pipeline through which packets will be routed. At the parser stage, headers are extracted from the packet first. The packet is then routed through a control flow of MAUs where extracted headers are manipulated. The MAU is the fundamental unit for packet processing in P4, and it consists of a table with matching keys as well as a list of possible custom actions that can be invoked upon matching. Finally, the packet passes through the deparser stage, where the modified header stack is re-added to the packet before it exits via the designated egress port.

Fig. 4.2 depicts the life cycle of a packet passing through a P4-based system. When a packet arrives at the switch from an ingress port, it is processed based on the P4 data path loaded into the data plane pipeline. If a packet matches any of the rules stored in the MAUs, it is processed following that rule. Otherwise, the packet is forwarded to the controller, which makes the packet forwarding decision based on the applications running in the control plane. After the controller's processing, the packet is returned to the data plane, along with the forwarding rules to be stored in the device's MAUs. Following packets matching the installed rule will only be processed at the data plane side.

The analysis in this chapter focuses on the key performance metric of packet forwarding latency (sojourn time) throughout the system. The sojourn time of the system can be calculated using Eq. (4.1). It is equal to the sum of the sojourn times of the two previously described paths: The first path is the data plane path, with sojourn time denoted as E_d, without the involvement of the controller. The second path is taken with probability P_{nf} when the received packet belongs to a new flow, which is the case where the packet is forwarded to the controller. In this case, the sojourn time is equal to the controller sojourn time, denoted as E_c, together with the data plane sojourn time. Hence, we can write:

$$E_{sys} = E_d + P_{nf} * (E_d + E_c). \tag{4.1}$$

4.1.2. Related Work

Modeling the performance of packet processors is critical for network planning, prediction, and problem mitigation. Since there are no related works in the literature that model the performance of complete P4-based systems, in the following, we first review the performance modeling works of the predecessor technology, i.e., the OF architecture. Then, we discuss the works that target modeling the performance of P4-based devices.

Jarschel et al. [29] investigate an OF-based architecture in which the switch and control plane are both modeled as M/M/1 queues. Mahmood et al. then map the previous model to a general Jackson model in [55] and use it for modeling OF networks as Jackson networks in [59]. Goto et al. [58] refines the model presented in [29] for a single node by including processing priorities for packets going to the switch. Ansell et al. [60] introduce a control plane application for monitoring networks and predicting their behavior based on queueing theory. OpenFlow Operations Per Second (OFLOPS) benchmarking suite, proposed by Rotsos et al. [34], breaks down the processing components in OF switches and models these components. They focus on investigating the behavior of the switch when different OF rules are inserted. Their findings indicate that the performance of OF switches is heavily dependent on their implementation.

While the literature is replete with papers analyzing and modeling the performance of OF-based switches, few works focus on modeling the performance of devices with programmable data planes.

Dang et al. proposed an approach for benchmarking individual components of P4 targets in [30], wherein they compare the latency performance of three software and emulated P4 devices, namely the BMv2, PISCES, and P4FPGA emulator when running legacy P4-14 constructs. Scholz et al. [32] select to measure and model only the relevant performance metrics of a P4 device based on its type. The resource utilization is selected in the case of the ASIC-based Tofino switches, while the latency and throughput metrics are chosen to model the performance of the t4p4s software switch. Lukács et al. propose a probabilistic model to calculate the expected execution cost for a given control flow graph in [61]. The worst case, as well as the expected cost when executing the program, can be calculated based on the P4 source code and information about the execution environment, i.e., the target platform and implementation. As a result, this cost is calculated as the sum of the costs of all possible execution paths multiplied by their execution probability.

None of the previous works consider developing a comprehensive analytical model for the performance of P4 devices. A network calculus-based model is proposed in [4] for analyzing the worst-case data plane performance of OF and P4-based switches. While network calculus-based models focus on worst-case analysis, in this chapter, we consider stochastic models based on the queueing theory that can provide information about the mean of various network metrics.

To the best of our knowledge, the literature does not provide a model yet for estimating the packet forwarding latency on a P4 device as a function of the running P4 program. Moreover, the performance of a complete P4-based system wherein the controller's impact is taken into consideration is not analyzed in the literature. Therefore, in this chapter, we fill in these research gaps.

4.2. Modeling Data Plane's Service Time

In this section, we propose a method for predicting the packet's forwarding latency when running arbitrary P4 programs on any P4 target. The complexity of the P4 program affects the processing latency on a P4 target and thus affects its overall forwarding latency. It is of paramount importance to understand the relation between the complexity of the P4 program and the processing latency or more generally the forwarding latency on its hosting P4 target as this presents a pivot building block for deriving more generic models for the performance of P4-based systems.

The proposed model uses the measurement results collected in Section 3.2 to quantify the delay in processing different atomic P4 operations or constructs when executed on different P4 targets. These previously collected measurements revealed in most cases a linear relationship between the measured forwarding latency and the number of applied P4 constructs. These findings will guide the modeling approach in this section. The model deals with the different atomic P4 constructs as a basis of a space spanned by all the possible P4 programs that are made up of a combination of the basis atomic P4 constructs. Accordingly, we predict in this section the forwarding latency of running an arbitrary P4 program on a P4 target as the linear combination of the latency cost for executing the P4 constructs that constitute the given P4 program. Note that the proposed method is applicable and tested when the exact processing path of the P4 program is known, with traffic of one flow, and with deterministic processing assumed. The contributions presented in this section are based on our publication [1].

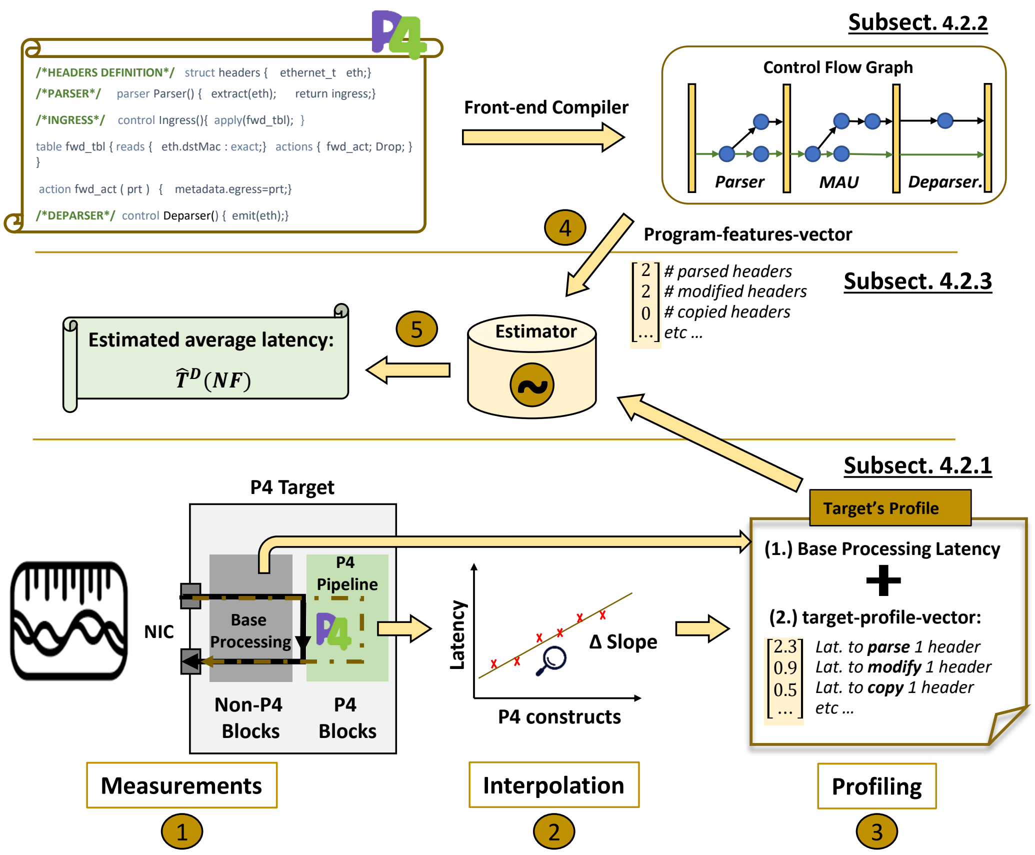
In Subsection 4.2.1, we start by describing the created profiles for different P4 targets. In Subsection 4.2.2, we describe the relevant information that should be extracted from a P4 program to perform the latency prediction. Subsection 4.2.3 elaborates on how the estimated average forwarding latency on a P4 target, using the derived targets' profile, can be calculated as a function of the constituting P4 constructs in the loaded P4 program. The overall workflow of the proposed estimation method is illustrated in Fig. 4.3. Finally, we validate the proposed method using measurements by testing it on three different realistic network functions in Subsection 4.2.4.

4.2.1. Profiling P4 Targets

In this subsection, we create performance profiles for different P4 targets based on the measurement results collected in Section 3.2. These profiles include all the performance information about a P4 target needed for predicting the packet forwarding latency on that target as a function of the loaded P4 program.

The forwarding latency of a P4 target is divided into two main components as shown in Fig. 4.3. The first one is the *base processing delay*, which includes the transmission delays and the processing delay of all non-P4 programmable blocks in a P4 target. The second component is the processing delay because of processing the P4 data path. Unlike the base processing delay which is fixed for a given P4 target, the P4 processing part varies based on the complexity of the loaded P4 program. This P4 program is decomposed using the methodology applied in Section 3.2, where we could quantify the latency cost of processing different atomic P4 constructs. Accordingly, the performance profile of a P4 target includes these two components as shown in Fig. 4.3.

The base processing latency denoted as T^{D}_{Base}, is defined as the end-to-end latency for executing a minimal P4 data path that parses a single header and applies a single table for forwarding packets on a P4 device D. The forwarding latency of this minimal P4 data path for different P4 devices can be derived from Fig. 3.3d by checking the forwarding latency when a single header is parsed for any selected packet size.



```
/*HEADERS DEFINITION*/   struct headers {   ethernet_t   eth;}
/*PARSER*/    parser Parser() {   extract(eth);     return ingress;}
/*INGRESS*/   control Ingress(){  apply(fwd_tbl);  }
table fwd_tbl { reads {   eth.dstMac : exact;}   actions {  fwd_act; Drop; }
}
 action fwd_act ( prt )   {    metadata.egress=prt;}
/*DEPARSER*/  control Deparser() {  emit(eth);}
```

Figure 4.3.: The general workflow for estimating the average forwarding latency when running arbitrary P4 programs on any P4 target. The extracted information related to the performance of P4 targets is described in Subsection 4.2.1. The extracted information from the given P4 program is described in Subsection 4.2.2.

In addition to this base processing latency, the profile includes the latency cost for executing different atomic P4 constructs on the profiled P4 target. This information is retrieved by using linear interpolation to extract the slopes of the curves that capture the increase in the packet forwarding latency as a function of the number of parsed headers, number of modified headers, number of copied headers, number of removed headers, number of added headers, and number of added tables from Figures 3.3d, 3.5d, 3.6d, 3.7d, 3.8d, and 3.9d, respectively. For each considered P4 operation, the slope of the fitted curve indicates the extra processing latency needed in μs to execute one instance of that operation. For example, if the interpolated slope of the fitted parsing header curve is equal to 2, then the latency cost of every parse header operation executed on top of the baseline pipeline is equal to 2 μs. Note that for adding and removing header cases on the Agilio CX SmartNIC, we perform piece-wise linear interpolation to capture the special behavior of the card when a big number of headers are added or removed. When performing linear interpolation for fitting the plots corresponding to all the P4 targets and in all P4 constructs cases, the root mean square error is evaluated and found to always be less than $0.4\,\mu$s. The row vector **target-profile-vector**, denoted by a^D, is defined as the vector that contains all the interpolated slopes corresponding to the different P4 constructs for a specific P4 target D. For example, it contains the slope of the curve corresponding to parsing headers, i.e., the latency cost for parsing a single header, besides the slopes corresponding to the other P4 operations/constructs. These slopes are denoted by $\Delta^D_{P4Construct}$ and ordered in a^D as shown in Eq. (4.2):

$$a^D = \left[\Delta^D_{ParseHdr}, \Delta^D_{ModifyHdr}, \Delta^D_{CopyHdr}, \Delta^D_{RemoveHdr}, \Delta^D_{AddHdr}, \Delta^D_{AddTable}\right]. \tag{4.2}$$

In the remainder of the thesis, we consider the results corresponding to a packet size equal to MTU. This packet size is selected because traffic streaming utilizes the maximum available payload of a packet. The proposed method can be easily applied to other packet sizes by simply substituting the corresponding values from Section 3.2. This is applicable as we observed in Section 3.2 that the packet size has only the effect of shifting the curves without significantly changing its slopes. For MTU-sized packets, the base processing latency is found to be equal to **45.9 μs, 3.54 μs, and 7.45 μs** in the cases of t4p4s, NetFPGA-SUME and Agilio CX SmartNIC, respectively, while the derived target-profile-vectors corresponding to these targets are shown in Eqs. (4.3), (4.4), and (4.5), respectively.

$$a^{t4p4s} = \begin{bmatrix} 0 & 0 & 0 & -0.29 & 0.4 & 0.08 \end{bmatrix}, \tag{4.3}$$

$$a^{NetFPGA} = \begin{bmatrix} 0.17 & 0 & 0 & -0.02 & 0.23 & 0.13 \end{bmatrix}, \tag{4.4}$$

$$a^{Agilio} = \begin{bmatrix} 0.11 & 0.5 & 0.28 & 1.43 & 1.59 & 0.37 \end{bmatrix}. \tag{4.5}$$

Note that in the case of Agilio CX SmartNIC, the following two corrections should be applied to capture its special behavior in some situations and thus accurately characterize its performance. First, when the number of added headers is greater than 6, the interpolated slope corresponding to adding header operations shown in Eq. (4.5) should be updated with the value 14.4 instead of 1.59 to capture the steeper slope observed after adding 6 headers, as described in Subsection 3.2.2. The second change is applied by adding a correction factor of $17.68\,\mu$s to the estimated average latency when the number of removed headers exceeds 7, as we previously observed in Subsection 3.2.2.

4.2.2. Analyzing P4 Programs

In this subsection, we describe the handling of the given P4 program toward estimating the forwarding latency when loading it to a P4 target.

First, we compile the given P4 program using the standard front-end compiler to get the Intermediate Representation (IR) of the program's logic that can be drawn as a Control Flow Graph (CFG). The CFG is a directed acyclic graph that depicts the packet's processing life cycle based on the P4 program starting when a packet is received on an ingress port and ending when it is pushed to an egress port or dropped. The arrows shown in Fig. 4.3 indicate the relative order of operations in CFG. Note that a P4 program can describe multiple NFs each with distinct packet processing paths using some conditions as shown in the CFG. Accordingly, the analysis of the program is applied to one selected path at a time. For example, the green lower path with two parse header operations and two packet header operations is highlighted as the selected path in Fig. 4.3.

Then, we extract from the CFG a vector named **program-features-vector** and denoted by $x(NF)$. This vector contains the number of occurrences of the different atomic P4 constructs in the given P4 program for a selected path describing a specific network functionality NF. It is defined:

$x(NF) =$ [#parsed headers-1, #modified headers, #copied headers, #removed headers, #added headers, #tables-1].

Note that the minimal baseline program already contains a single parse header operation and a single table as described in Subsection 3.2.1. So we decrement the number of parsed headers and the number of tables by one in the program-features-vector to avoid counting the occurrence of these constructs twice as they are already included in the baseline processing latency component.

4.2.3. Latency Estimation

In this subsection, we elaborate on the equation that estimates the packet forwarding latency using the predefined target-profile-vector and the program-features-vector.

We recall that each designed experiment in Subsection 3.2.1 for the different evaluated P4 constructs built on top of the same baseline pipeline that parses some number of headers and modifies the egress port according to the ingress port in a single table to forward the packet. Additionally, we made sure in each of these experiments that the analyzed P4 construct is the only parameter varying in the evaluated P4 programs. Accordingly, the estimated average forwarding latency of a given NF written as a P4 program when running on a P4 target D is equal to the summation of two terms:

(1) The measured base processing latency on a P4 target D, denoted as T^D_{Base}, which includes the transmission delays, the propagation delays, the packet processing delays before and after entering the P4 pipeline, and the packet processing delay of the minimal P4 operations (parsing a single header and executing a single table) for forwarding packets on a P4 target; *(2)* The sum of the extra latency cost for executing the P4 constructs that constitute the selected P4 program. This is equal to the **target-profile-vector** a^D that contains the latency for executing a single instance of different P4 constructs weighted with the number of occurrences of each of these P4 constructs collected in the **program-features-vector** $x(NF)$.

$$\hat{T}^D(NF) = T^D_{Base} + a^D \cdot x^T(NF). \tag{4.6}$$

The formula for evaluating the estimated average packet forwarding latency in μs, $\hat{T}^D(NF)$, when running network function NF on a selected P4 target D is shown in Eq. (4.6).

4.2.4. Evaluation

In this subsection, we evaluate the proposed model by comparing the forwarding latency it predicts to the actual forwarding latency derived from measurements. The proposed estimation method is applied for three network functions written in P4 which have increasing complexity. Then, the estimated latency is compared to the measured latency when the three tested network functions are loaded to each of the investigated P4 targets. The three selected network functions are the following:

L3Fwd: This function describes the layer 3 forwarding or IPv4 forwarding. The P4 data path of this network function is made up of the following operations: *(i)* parsing Ethernet and IPv4 headers, *(ii)* matching on IPv4 destination address in a single table, and *(iii)* modifying the source and destination Media Access Control (MAC) addresses of the Ethernet header and the time to live Time to Live (TTL) field of the IPv4 header if there is a match. The program-features-vector corresponding to this network function $x(L3Fwd)$ recognizes two parse header operations, two modify header operations, and one table. After decrementing the number of parsed headers and added tables operations each by one as described in Subsection 4.2.2, we get $x(L3Fwd) = \begin{bmatrix}1 & 2 & 0 & 0 & 0 & 0\end{bmatrix}$.

L3Fwd + UDP-based Firewall: This function describes the IPv4 forwarding functionality in addition to a simple stateless UDP-based firewall functionality. The P4 data path corresponding to this network function is similar to that described for L3Fwd but it also includes parsing one more header which is the UDP header. Moreover, the data path of this network function requires one more table for matching on the UDP destination port to drop/filter undesired packets. The program-features-vector

Table 4.1.: Evaluation results of the proposed method for different network functions and P4 targets.

NF	P4 target	Meas. Avg. Latency	Est. Avg. Latency
L3Fwd	t4p4s	45.9 μs	45.9 μs
	Agilio CX	8.2 μs	8.6 μs
	NetFPGA-SUME	3.7 μs	3.7 μs
L3Fwd + Firewall	t4p4s	45.9 μs	46 μs
	Agilio CX	8.9 μs	9.0 μs
	NetFPGA-SUME	4.0 μs	4.0 μs
VxLAN_Decap	t4p4s	45.9 μs	44.8μs
	Agilio CX	15.2 μs	15.5μs
	NetFPGA-SUME	5.2 μs	5.3μs

corresponding to this network function $x(L3Fwd + Firewall)$ recognizes three parse header operations, two modify header operations, and two tables. Accordingly, the program-features-vector of this NF $x(L3Fwd + Firewall)$ equals to $\begin{bmatrix}2 & 2 & 0 & 0 & 0 & 1\end{bmatrix}$.

VxLAN Decapsulation: This function describes Virtual Extensible LAN (VxLAN) decapsulation functionality. Note that for this network function, we made sure that the header stack up to the PTP header is maintained before implementing the operations related to the VxLAN decapsulation. This is important for allowing the used packet generator MoonGen [31] to perform latency measurements using PTP mechanisms. The evaluated P4 pipeline of this NF parses the following header stack: Eth, IPv4, UDP, PTP, ETH, IPv4, UDP, VxLAN, Eth, IPv4, UDP. It contains two tables: One table to forward packets based on ingress port matching, and another table that matches on the VxLAN Network Identifier (VNI) header field to perform the VxLAN Decapsulation action. The VxLAN Decapsulation action copies the inner Ethernet, IPv4, and UDP headers into the outer headers. Then, it removes these inner Ethernet, IPv4, and UDP headers along with the VxLAN header. This NF requires parsing 11 headers, copying 3 headers, removing 4 headers, and matching in 2 tables. The program-features-vector $x(VxLAN\_Decap)$ corresponding to this network function is derived to be equal to $\begin{bmatrix}10 & 0 & 3 & 4 & 0 & 1\end{bmatrix}$.

After extracting the program-features-vector, and using the previously derived target-profile-vectors of the three investigated P4 targets, the estimated average latency when running any of these network functions on any profiled P4 target can be calculated using Eq. (4.6). For example, the average forwarding latency of running L3Fwd network function on Agilio CX SmartNIC can be calculated as: $\hat{T}^{Agilio}(L3Fwd) = T^{Agilio}_{Base} + a^{Agilio} \cdot x^T(L3Fwd) = 7.45 + [0.11, 0.5, 0.28, 1.43, 1.59, 0.37] \cdot [1, 2, 0, 0, 0, 0]^T = 8.56\,\mu$s.

The average forwarding latency when running each of the three network functions described above on the three state-of-the-art P4 targets is measured. These measurement results serve as the ground truth for assessing the accuracy of the latency estimation calculated based on Eq. (4.6). Table 4.1 summarizes the measured and estimated average forwarding latency in μs of the three evaluated network functions on the three investigated P4 targets. It can be observed that the accuracy of the forwarding latency estimations is always greater than 95% when compared to the measured latency. This high accuracy validates the correctness and effectiveness of the proposed estimation method.

When looking into the results corresponding to the different P4 targets, we can observe that the prediction accuracy of the forwarding delay on the NetFPGA-SUME card is the highest, followed by that of the t4p4s software switch, followed by that of the Agilio CX SmartNIC. This accuracy varies based on the degree of dependency of the P4 target on the loaded P4 pipeline. This dependency can be identified by inspecting the target-profile-vector of the investigated P4 targets from Subsection 4.2.1. For example, Agilio CX SmartNIC has the lowest prediction accuracy because it is performance is the most dependent one among other P4 targets on the complexity of the loaded P4 program.

Note that the described model in this thesis is slightly different from that proposed in our published work [1]. The only difference is that we decoupled the processing latency of parsing headers from the base processing latency as described in Subsection 4.2.1. This proposed methodology is generic enough to be applied to other P4 targets. A profile for any emerging P4 target can be derived by extracting the **target-profile-vector** as described in Subsection 4.2.1. This profile with all relevant performance information can be published by researchers or any third-party institution. Then, the P4 compiler can easily be extended to analyze and decompose the given P4 program to extract the **programs-feature-vector** to calculate the estimated average forwarding latency when running arbitrary P4 programs on a specific P4 target with a given performance profile, as described in Subsections 4.2.2 and 4.2.3.

The proposed methodology is generic and can be applied to other P4 targets. Moreover, different performance metrics such as the deadline latency and the jitter of a device can be estimated by tailoring this method where the maximum and the variance of the measured packet latency are considered instead of the average forwarding latency.

4.3. Modeling P4-based Systems

In the following, we first describe in Subsection 4.3.1 the requirements targeted by the performance model to be proposed. Then, in Subsections 4.3.2 and 4.3.3, we propose two queueing models to capture the performance of P4-based systems. The first model is meant to be simple with memoryless service times (i.e., exponentially distributed), while the second model relaxes this assumption considering general service times whose distribution is determined based on measurement data to better capture the real behavior of the modeled system. Finally, Subsection 4.3.4 clarifies the underlying assumptions taken when deriving the two models.

4.3.1. Model Requirements

The proposed model for P4-based systems should satisfy the following requirements:

- The model should take into account the variability of the packet processing latency at the P4 data plane. This variability is based on the complexity of the configured P4 program/pipeline as discussed in Section 4.2.
- The model should be generic to cover the performance of different P4 devices, which may have different service processes.
- The model should take into account that the SDN controller may be involved in processing packets that have no matching rule at the data plane; typically, this is the case for the first packet of every newly observed flow.
- The model should account for the variable volume of incoming traffic loads, which mimics the situation in production environments.
- The model should stay relatively simple to permit a quick first-hand understanding of the system's performance and limitations.

4.3.2. Simple Model with Exponentially-Distributed Service Times

In the following, a simple queueing model for P4-based systems is proposed. The proposed model is a feedback-oriented queueing system, similar to the approach followed in [29]. The forwarding behavior of the P4 data plane is abstracted as an *M/M/1* queueing system, while the forwarding of the controller is abstracted as a feedback queueing system of the same *M/M/1* type. The overall proposed model of the system where each influential factor is associated with a system parameter is shown in Fig. 4.4. The external arrival process to the switch is assumed to follow a Poisson process with an average packet rate equal to λ_{ext}. A Poisson arrival process is selected since it is a convenient mathematical representation of the input traffic in many communication systems. Moreover, since we are dealing with time averages, the Poisson Arrivals See Time Averages (PASTA) property of the Poisson process allows us to assume the interarrival times are memoryless [54].

In the following, we will describe each of the data plane and controller models.

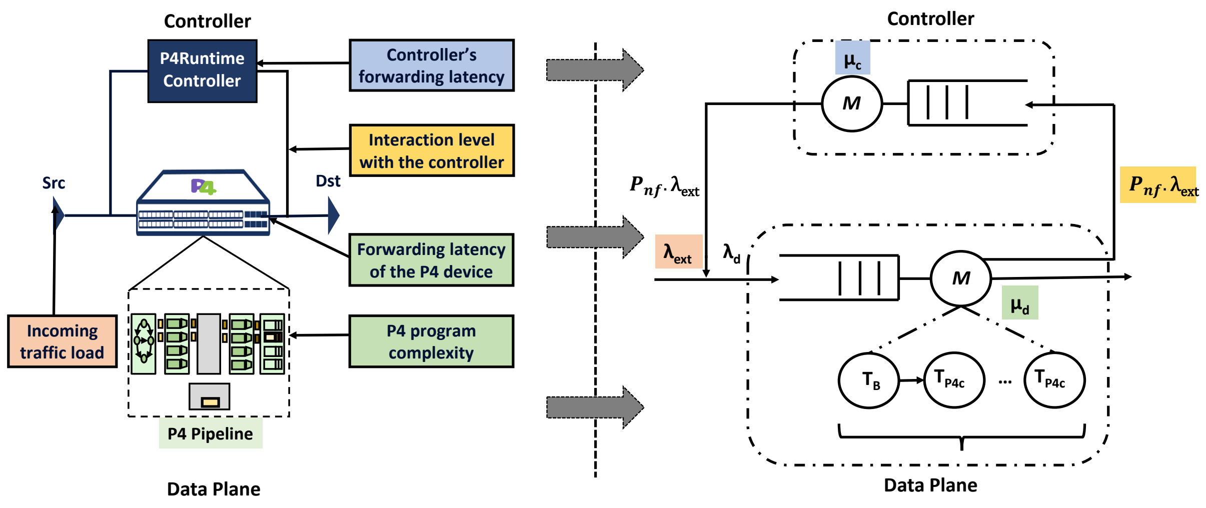

Controller
P4Runtime Controller
Controller's forwarding latency
Interaction level with the controller
Src
Dst
Forwarding latency of the P4 device
Incoming traffic load
P4 program complexity
P4 Pipeline
Data Plane
Controller
μc
M
$P_{nf} \cdot \lambda_{ext}$
$P_{nf} \cdot \lambda_{ext}$
λext
λd
M
μd
T_B
T_{P4c}
...
T_{P4c}
Data Plane

Figure 4.4.: A simple model for P4-based system abstracting a programmable data plane and a controller.

Data Plane

The service time on the P4 data plane is assumed to be exponentially distributed. The average service time on a P4 device is not constant as the processing latency on the device varies based on the complexity of the loaded P4 program. When the loaded program is more complex, the processing delay increases, and accordingly the forwarding latency or the service time at the data plane becomes longer. This creates a challenge in capturing the variation in the performance of the data plane since it can arbitrarily change based on the complexity of the loaded P4-based network function. However, the previous work presented in Section 4.2 already addresses this issue, where it provides a method for estimating the packet forwarding latency on a P4 device as a function of the loaded P4 program. According to these findings, we model the data plane of a P4 device as a queueing system made up of a queue followed by a series of sequential servers. The first server abstracts the base processing T_B that takes place in a P4 target, and the following servers abstract the processing of the different P4 constructs T_{P4c} that constitute the running P4 program. To simplify the model, we assume that all these described service processes, when combined, form a service process with exponentially-distributed service times with an average service rate equal to $\mu_d^D(NF)$ evaluated as shown in Eq. (4.7).

$$\mu_d^D(NF) = (T_{Base}^D + a^D \cdot x^T(NF))^{-1} \quad \forall NF, \forall D, \tag{4.7}$$

where $\mu_d^D(NF)$ is the data plane service rate when running network function NF on a P4 device D. It is equal to the inverse of the forwarding latency, which is calculated as the summation of two terms: The first term T_{Base}^D is the base processing delay on a selected P4 device. The second term, calculated as the dot product of the target-profile-vector a^D and the program-feature-vector $x^T(NF)$, is the delay required for processing the P4 constructs that constitute a network function NF on a given P4 device D.

The size of the queue in the data plane model is assumed to be infinite. The arrival process into the data plane has a rate λ_d, which can be evaluated as shown in Eq. (4.8).

$$\lambda_d = \lambda_{ext} + P_{nf} * \lambda_{ext}, \tag{4.8}$$

It is equal to the combination of the externally arriving packets, with a rate equal to λ_{ext}, and the loop-backed packets from the controller [55].

Controller

The controller is modeled as a feedback queueing system on top of the data plane model. To keep the model simple, we assume that service time at the controller is exponentially distributed. Note that, in this model, we assume that the interarrival process to the controller and the data plane follows a Poisson process with exponentially distributed service times. The latter may not fully hold since the interarrival process to the data plane is made up of the combination of two dependent Poisson processes (i.e., external arrivals and feedback arrivals). Recalling that the first packet corresponding to a new flow will be forwarded to the controller only once after being processed at the data plane, the arrival process incoming to the controller will have an average arrival rate equal to $P_{nf} * \lambda_{ext}$, where P_{nf} is equal to the probability that an observed packet corresponds to a new flow. Packets with a rate of $P_{nf} * \lambda_{ext}$ are fed back to the data plane from the controller as shown in Fig. 4.4.

Overall System's Model

The model of the P4 system under consideration resembles a Jackson network [54]. Therefore, the average sojourn time at the data plane and the controller E_d and E_c can be evaluated based on the equations of *M/M/1* queueing systems. By substituting these components in the system's average sojourn time shown in Eq. (4.1), we get the packet's sojourn time evaluated according to this simple model as shown in Eq. (4.9):

$$E_{sys} = \frac{1}{\mu_d^D(NF) - \lambda_d} + P_{nf} * (\frac{1}{\mu_d^D(NF) - \lambda_d} + \frac{1}{\mu_c - \lambda_c}), \tag{4.9}$$

where $\mu_d^D(NF)$ and λ_d can be evaluated as illustrated in Eqs. (4.7) and (4.8), respectively, while μ_c and λ_c refer to the controller's service and arrival rates, respectively.

4.3.3. Refined Model with General Service Times

The next step is to derive a refined model that better captures the real performance of P4 programmable devices. This is done by adopting the same feedback-oriented queueing system described in the simple model case, but with the assumption related to the exponentially-distributed service times at the data plane of P4 devices and at the controller relaxed. Alternatively, in this model, we go beyond the first moment in characterizing the service time of the data plane and controller. More specifically, we assume a general service process at the data plane and controller, whose distribution is determined based on measurements conducted on these components. The data plane is abstracted as an *M/G/1* queueing system, while the controller is abstracted as a feedback queueing system of the type *M/G/1*. Note that we assume that the incoming traffic to the controller and that forwarded back to the data plane from the controller have exponential interarrival time even though this traffic is generated from generic service processes. The impact of this assumption on the model accuracy is put to test in the evaluation section in 4.4.3.

In the following, we first elaborate on the data plane's model and its fitting procedure, then we illustrate the controller model, and finally, we discuss the system's model with the data plane and controller components involved.

Data Plane

In this enhanced model, the distribution of the data plane's service time is not assumed to be exponential, rather it is derived based on the measured service time of different P4 devices. The service time of the data plane varies based on the selected P4 device and based on the complexity of the loaded P4 program for each device.

In our approach, we start by fitting a service curve that best fits the measured service time when a baseline P4 program, which describes the minimal required processing for forwarding packets, is loaded to different P4 devices. This way we can find the service time distribution that best fits the realistic performance of the different devices.

The Matlab Statistics and Machine Learning toolbox [119] can be used to find the best fitting distribution for the service time of the modeled P4 device. The Akaike Information Criterion (AIC) metric can also be used to assess the goodness of fit of different distributions. Besides the goodness of fit, the best-fitting distribution should also have a closed-form expression. The latter is important especially since the analytical equations of the model eventually depend on the selected distribution, and the complexity of these equations should not grow arbitrarily large.

Note that different P4 devices are distinguished by having different standard deviation values that characterize the sparseness observed in measured forwarding delay.

After selecting the best fitting distribution for the baseline data path, the next step is to capture the effect of processing more complex P4 programs on each device. In Section 4.2, we showcased that the latency for processing a P4 program could be calculated as the summation of the base processing latency of the device and the processing latency of each P4 construct constituting the loaded P4 program. Accordingly, these components are considered in the data plane path of the model as shown in Fig. 4.4. The average service time is equal to the summation of the baseline processing latency, extracted from the fitted Erlang distribution, and the average processing time of all the constituting P4 constructs creating the loaded P4 data path. The latency cost of the constituting P4 constructs on each P4 device can be extracted based on the target-profile-vector derived in Subsection 4.2.1.

The overall latency of the data plane component in this refined model can be calculated using the Pollaczek-Khinchin formula for M/G/1 queue systems as follows [69]:

$$E_d^G = \frac{1}{\mu_d^D(NF)} + \frac{\lambda_d * (\frac{1}{\mu_d^D(NF)^2} + \sigma_d^2)}{2 * (1 - \rho_d)}, \tag{4.10}$$

where $\mu_d^D(NF)$ and λ_d are equal to the data plane's average service rate and arrival rate and can be calculated as illustrated in Eqs. (4.7) and (4.8), respectively. The parameter σ_d stands for the standard deviation of the service time for different P4 devices and is constant for a given P4 device. Finally, ρ_d stands for the utilization of the data plane queue.

Controller

Similar to the approach followed in data plane modeling, we target enhancing the model adopted for the control plane. This is achieved by relaxing the assumption related to the exponentially-distributed service times at the controller. So instead of assuming that the controller's service time is exponentially-distributed, we adopt a more accurate distribution that fits the real service time of the controller, which is identified based on measurements.

The de-facto control runtime for P4 devices is the P4Runtime framework [92]. This framework enables controlling P4 data planes that can be reconfigured at any time. Our novel P4RCProbe tool, described in Section 3.5, can be used to benchmark the performance of P4Runtime-based controllers. The tool can be configured to emulate a single switch sending packets according to Poisson distribution with a configurable rate. It also records the RTT of each packet sent to the controller. The distribution of these collected RTT values is used as a ground truth empirical distribution of the controller's service time.

Next, we search for the theoretical distribution that best fits the measurement-based empirical distribution. Again, the AIC metric is used to assess the goodness of fit of different candidate distributions, and then the best fitting distribution is used to model the service time of the controller.

The overall latency of the control plane component can be calculated according to the Pollaczek-Khinchin formula for M/G/1 queue systems as follows [69]:

$$E_c^G = \frac{1}{\mu_c} + \frac{\lambda_c * (\frac{1}{\mu_c^2} + \sigma_c^2)}{2 * (1 - \rho_c)}, \tag{4.11}$$

where μ_c and σ_c stand for the service rate and standard deviation of the service distribution, respectively, and they are constant for a given P4 controller. The controller's utilization is denoted by ρ_c, while the controller's interarrival rate is denoted by λ_c and is calculated as $P_{nf} * \lambda_{ext}$.

Overall System's Model

After evaluating the average sojourn time at the data plane and the controller E_d^G and E_c^G based on Eqs. (4.10) and (4.11), respectively, the sojourn time of the system can be calculated by substituting these values in the system's average sojourn time Eq. shown in (4.1). The average packet's sojourn time in the system according to this refined model is shown in Eq. (4.12):

$$E_{sys}^G = \frac{1}{\mu_d^D(NF)} + \frac{\lambda_d * (\frac{1}{\mu_d^D(NF)^2} + \sigma_d^2)}{2 * (1 - \rho_d)} + P_{nf} * [\frac{1}{\mu_d^D(NF)} + \frac{\lambda_d * (\frac{1}{\mu_d^D(NF)^2} + \sigma_d^2)}{2 * (1 - \rho_d)} + \frac{1}{\mu_c} + \frac{\lambda_c * (\frac{1}{\mu_c^2} + \sigma_c^2)}{2 * (1 - \rho_c)}], \tag{4.12}$$

where $\mu_d^D(NF)$ and λ_d can be evaluated as illustrated in Eqs. (4.7) and (4.8), respectively, while μ_c and λ_c refer to the controller's service and arrival rates, respectively. The variables ρ_d and ρ_c correspond to the data plane and controller utilization, while σ_d and σ_c correspond to the standard deviation of the data plane's and controller's service time which can be derived from measurements. Finally, P_{nf} stands again for the level of interaction with the controller.

4.3.4. Assumptions

In the following, we highlight some of the assumptions that were made while deriving the models described in the previous subsection. In the two models, the external arrival process coming to the switch is assumed to follow a Poisson distribution with a single queue assumed to abstract the incoming traffic from different ports of the switch. Additionally, it is assumed that the first packet of every newly observed flow is forwarded to the controller, which is usually the case for TCP traffic. Moreover, the analysis and evaluation conducted for the two models focus on the average results of the system rather than on actual distributions. Finally, the interarrival process into the controller and back from the controller to the data plane are both assumed to be exponentially distributed in the two proposed models. The service times for the data plane and controller are assumed to be exponentially distributed in the first simple model, while this assumption is relaxed in the second refined model where these service times are assumed to follow a general distribution derived based on measurement results.

4.4. Evaluation of P4-based System's Models

In this section, we test the derived models under various scenarios and validate the results with simulations. We first elaborate on the parameters and evaluation scenarios adopted for conducting this evaluation in Subsection 4.4.1. Then, we present the evaluation results corresponding to the simple exponentially-distributed service time model and the refined model with generally-distributed service time in Subsections 4.4.2 and 4.4.3, respectively. Finally, in Subsection 4.4.4, we provide a constraint for properly dimensioning the use of systems with P4 devices based on the results of the previous two subsections.

4.4.1. Evaluation Scenarios and Parameters

In the following, we explain the different varied influential parameters of the system. Then, we elaborate on the conducted evaluation scenarios.

Data Plane's Parameters:

In this evaluation, we reuse the following three NFs described in Subsection 4.2.4. These NFs are L3Fwd, L3Fwd + UDP-based Firewall, and VxLAN Decapsulation. The packet forwarding latency when running these three NFs on t4p4s software switch, Agilio CX SmartNIC, and NetFPGA-SUME is summarized in Table 4.1. These results are used to derive the service rate of the data plane $\mu_d^D(NF)$ for different P4 targets and different NFs by taking the inverse of the forwarding latency values presented in Table 4.1 according to Eq. (4.7).

For the refined model with generally-distributed service times, we need to find the best fitting distribution as explained in Subsection 4.3.3. We refer to the previous measurements presented in Subsection 3.2.2, wherein the forwarding latency when running baseline P4 programs on NetFPGA-SUME, Agilio CX SmartNIC, and t4p4s software switch are plotted, as the ground truth presentation of the device's service process. It is clearly observed from these plots that the distribution and the range of the service curves corresponding to different devices differ a lot even when these P4 devices run the same P4 program.

To this end, we decided to model the service time of the data plane corresponding to different devices with an **Erlang** distribution. The Erlang distribution is found to be a plausible option since it is one of the best-fitting distributions with a closed-form expression and relatively manageable expression complexity. Moreover, it is selected because the measurement service time of different devices is found to have a coefficient of variation of less than one, which is best mimicked by the Erlang distribution.

The goodness of fit of the Erlang distribution is better than that found when fitting an exponential distribution to the service time. This means that the refined *M/G/1* model better mimics the service process of P4 devices' data plane and thus the model better captures the real performance of these devices. Note that we do not remove outliers from the measured data to keep the derivations as authentic and realistic as possible. The Root Mean Square Error (RMSE) between the Cumulative Density Function (CDF) of fitted Erlang distribution and the Empirical CDF of measured data is found to be equal to 0.5, 0.3, and 5.6 in the cases of NetFPGA-SUME, Agilio CX SmartNIC, and t4p4s, respectively.

The average service rate is also derived from Table 4.1, where the average service time corresponding to the different combinations of running different NFs on different P4 targets is shown. The standard deviation of the service times of different P4 targets are calculated based on the measurement data and is found to be equal to 0.39, 0.21, and 4.55 μs in the cases of NetFPGA-SUME, Agilio CX SmartNIC, and t4p4s, respectively. Note that this standard deviation is needed for calculating the average sojourn time in the refined model with generally-distributed service times.

Controller's Parameters:

Currently, the ONOS controller is one of the few SDN controllers that has a stable P4Runtime implementation. For this reason, we select ONOS as a representative P4-compatible SDN controller in this modeling exercise.

The service distribution that best mimics the behavior of the controller is derived based on measurements. The testbed for conducting this evaluation is similar to that described in Subsection 3.5.2, where the P4RCProbe tool is used. The tool runs on one server while the controller runs on another server. The tool is then configured to emulate a single switch that sends 100,000 packets to the ONOS controller according to Poisson distribution with a rate equal to 5 packets per second.

The tool records the RTT of each packet sent to ONOS. The distribution of the collected RTT values is used as a ground truth empirical distribution of the controller's service time. The average service time is found to be approximately equal to 8 ms, while the standard deviation is equal to 1.1 ms.

The selected values corresponding to the controller's service process are inspired by the collected measurement results. In this evaluation, we consider a 10 ms average controller service time as the worst-case scenario. We expect that the continuous enhancement in the implementation of these controllers will reduce the service time in the future. Accordingly, we study the following three different controller's service times: 31 μs, 240 μs, and 10 ms to accommodate for a wide range of controller's performance, where the standard deviation is always set to 10% of the average controller's service time to be in line with the revealed measurement results. The standard deviation is relevant for calculating the sojourn time in the refined model case with generally-distributed service times.

The Erlang distribution is selected as the general distribution that best mimics the service process at the controller in the refined model case based on the same arguments provided in the previous data plane modeling part.

Involvement Level of the Controller:

The involvement level of the controller is varied by varying the probability of observing a new flow, P_{nf}. It is varied from zero where all packet processing takes place on the data plane to one where the controller is involved in handling every packet arriving at the switch. The latter can be useful in a scenario when a new southbound protocol is tested, wherein every packet arriving at the switch needs to be forwarded to the controller, such as the case in [56]. Moreover, values of P_{nf} equal to 0.2 and 0.5 are taken as intermediate values, while $P_{nf} = 0.04$ is evaluated since it reflects the probability of observing a new flow in a normal production network handling end-user traffic based on [57].

Evaluation Scenario

The proposed theoretical models in Subsections 4.3.2 and 4.3.3 are evaluated using simulations. We chose simulations over measurements to validate the models because simulation results can be obtained faster while still allowing us to widely vary the model's parameters. To that end, we used MATLAB [119] to create a packet-based simulation that reflects the chosen model.

The simulation ensures that a packet only visits the controller once to adhere to the realistic behavior of the packet's life cycle. The controller and data plane service times are set to be exponential- and Erlang-distributed in the simulations in Subsections 4.4.2 and 4.4.3, respectively. The external arrival process, like in the analytical model, is set to follow a Poisson process. The Poisson assumptions on the arrival process to the controller, which is the departure process from the data plane, as well as the one on the arrival process from the controller to the data plane, which is the loopback traffic, are relaxed. In the simulations, we do not force these arrival processes to be Poisson, rather we keep the interaction between the controller and data plane systems free to reflect the real performance of the overall system.

Each simulation in this section with different parameter sets is run on approximately 1.3 million packets and repeated five times with different seeds. On top of the average simulation results, the 95 percent confidence interval is plotted.

4.4.2. Simple Model with Exponentially-Distributed Service Times

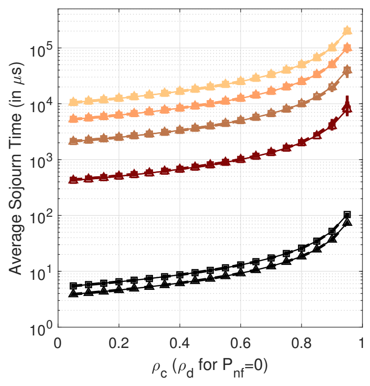
In this subsection, we present the results corresponding to the simple model presented in Subsection 4.3.2. The sojourn time is calculated using Eq. (4.9) based on the proposed model, and the simulation results are presented in the following when the controller's service time is set to 10 ms, 240 μs, and 31 μs.

Results of Slow Controller Case

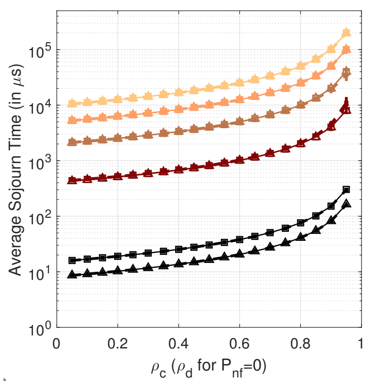
The analytical and simulation results of the average sojourn time in μs and in logarithmic scale as a function of controller load, ρ_c, for different P_{nf} values when controller service time is set to 10 ms are shown in Fig. 4.5. Figs. 4.5a, 4.5b, and 4.5c correspond to cases where L3Fwd and VxLAN Decapsulation P4 pipelines are evaluated on NetFPGA-SUME, Agilio CX SmartNIC, and t4p4s software switch, respectively, with data plane service times taken from Table 4.1. The results for the L3Fwd + UDP-based Firewall pipeline are not plotted because they look similar to the L3Fwd results as they both have very similar service times as can be inspected from Table 4.1. Note that when $P_{nf} = 0$, the x-axis varies depending on the data plane load since the controller's load is always zero.

All of these results show that the sojourn time increases almost linearly up to a load value of 0.8, whereafter the latency increases more sharply as the load approaches full utilization (i.e., equal to one), which is typical for all queueing systems. Furthermore, we can see that the derived analytical equations always accurately capture the performance of the simulated system under various loads.

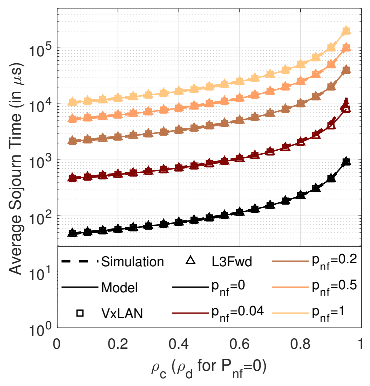
In the case of $P_{nf} = 0$, the system's performance is only influenced by the performance



(a) NetFPGA-SUME



(b) Agilio CX SmartNIC



(c) t4p4s switch

Figure 4.5.: Sojourn time of the system in the slow controller case.

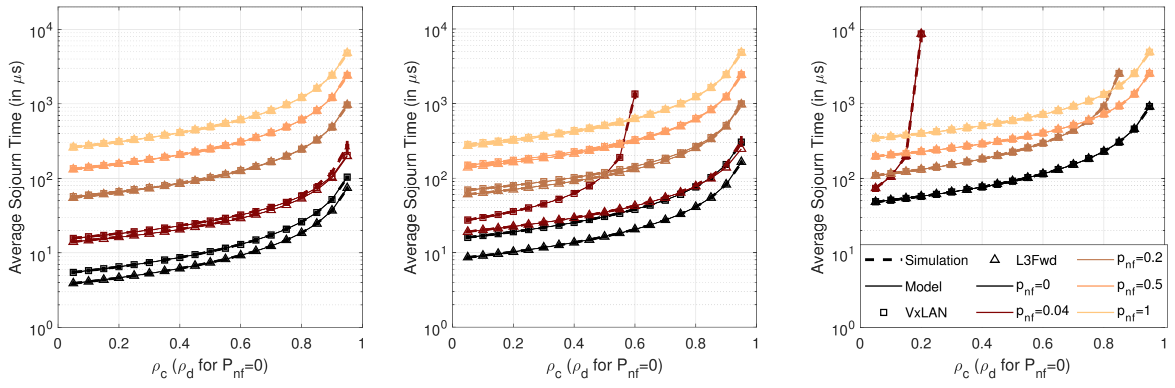
of the data plane. In this case, we can see that when the data plane's service time is larger, the curves are shifted upward. For example, when comparing the results in Fig. 4.5a corresponding to NetFPGA-SUME that has the smallest data plane's service time to the results presented in Fig. 4.5c for t4p4s software switch that has the largest service time. Furthermore, when the load is low, the sojourn time is nearly equal to the data plane's service time. For example, as shown in Fig. 4.5b, the sojourn time in cases of L3Fwd and VxLAN Decapsulation is equal to 8.6 and 16 μs, respectively when running on the Agilio CX SmartNIC with a load of 0.05.

When the P_{nf} values increase from zero to one, the sojourn time increases by 2 to 3 orders of magnitude. This significant increase in latency is due to the controller's involvement in packet processing, which has a much longer service time, i.e., 10ms, compared to the data plane's service time. Furthermore, the impact of loading different P4 pipelines becomes negligible as P_{nf} values increase, where the small difference in the data plane's service time is obscured by the controller's long service time.

Results of Average Controller Case

In the following, we examine the differences in system performance when the controller's service time is shorter, i.e., a faster controller compared to the previous case. Fig. 4.6 corresponds to Fig. 4.5 in displaying the analytical and simulation results of the average sojourn time for various cases but with the controller's service time set to 240 μs.

Because most of the previously examined observations remain valid, we will only look at the changes in performance that occur as a result of the controller's reduced service time. As the controller's service time decreases, the data plane performance becomes more important in determining the system's performance. When P_{nf} is low,



(a) NetFPGA-SUME (b) Agilio CX SmartNIC (c) t4p4s switch

Figure 4.6.: Sojourn time of the system in the average controller case.

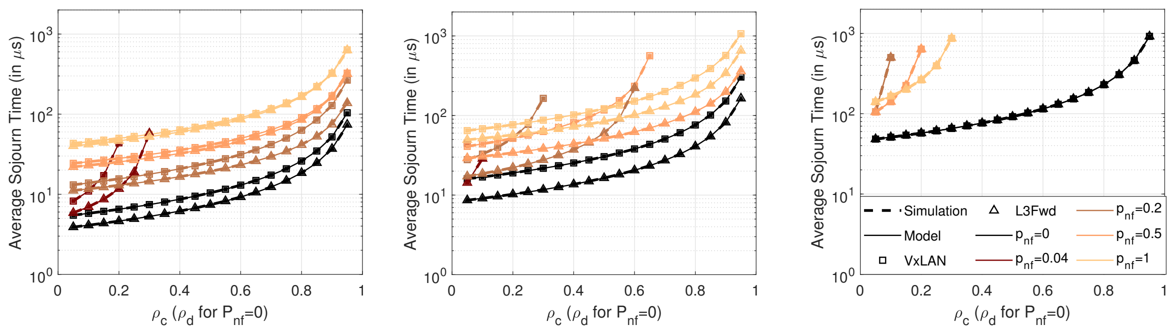
the performance curves corresponding to different P4 pipelines become more distinct. This is mostly visible in the case of Agilio CX SmartNIC in Fig. 4.6b since the difference in the pipeline's service time is larger on this device compared to the other devices.

The sojourn time increases by one to two orders of magnitude as P_{nf} increases. This smaller impact of P_{nf} on sojourn time is due to the controller requiring less service time to process packets corresponding to newly observed flows forwarded to it.

Unlike the previously observed patterns in the case of slow controller, we see a significant increase in latency in some of the curves in this case with average controller service time. This sharp increase is taking place in the following cases: *(i)* Agilio CX SmartNIC with VxLAN pipeline at $P_{nf} = 0.04$, *(ii)* t4p4s software switch with L3Fwd and VxLAN pipelines at $P_{nf} = 0.04$ and later at $P_{nf} = 0.2$. It is worth noting that in these cases, we skipped plotting the sojourn time after a certain point because the values became arbitrarily large. The reason for this sudden increase in latency is that the data plane's load/utilization in these cases is approaching one. More importantly, we can see that even when the system approaches these corner cases, the analytical model can still capture the system's performance. This instability will never happen with the controller queue because we explicitly vary its load between 0.05 and 0.95 in all tested cases. Subsection 4.4.4 provides a detailed analysis of this behavior.

Results of Fast Controller Case

Fig. 4.7 depicts the sojourn time of the various considered cases when the controller is fast, with an average service time of 31 μs. In this case, as in the previous case of the average controller, when latency caused by control plane processing is reduced, the data plane plays a larger role in determining the performance of the system. In general, we can see that the average sojourn time is reduced when compared to the previous



(a) NetFPGA-SUME (b) Agilio CX SmartNIC (c) t4p4s switch

Figure 4.7.: Sojourn time of the system in the fast controller case.

two cases. Moreover, the impact of loading different P4 programs with different service times at the data plane on the average sojourn time of the system is amplified.

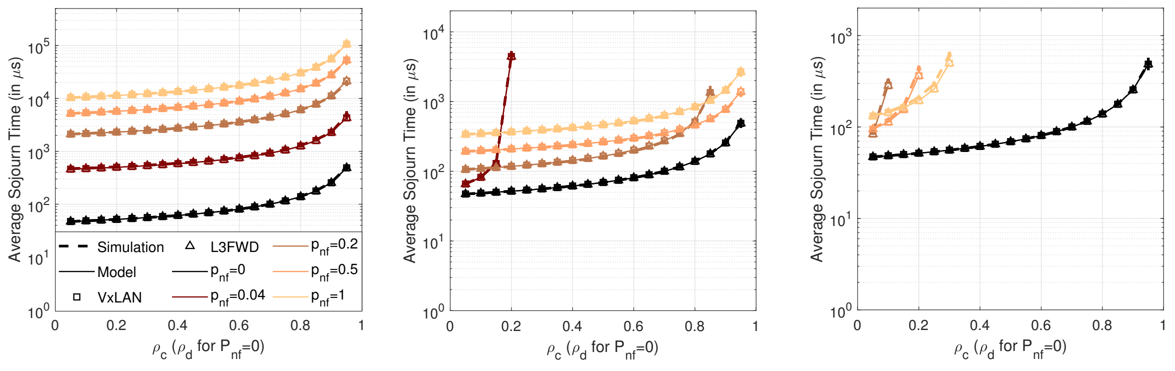
Furthermore, we can clearly see that the data plane utilization limit is frequently exceeded. This can be seen in the following situations: *(i)* The violation occurs in both pipelines in Agilio CX SmartNIC when $P_{nf} = 0.04$ and 0.2, but only in the VxLAN case when $P_{nf} = 0.5$; *(ii)* The violation occurs in both pipelines of NetFPGA-SUME when $P_{nf} = 0.04$; *(iii)* The violation occurs in all cases, except when $P_{nf} = 0$ where data plane's load is explicitly configured, in the t4p4s software switch. This violation occurs at various ρ_c values, and when the violation occurs early before $\rho_c = 0.05$, the curve is not shown at all, as can be observed in Fig. 4.7c when $P_{nf} = 0.04$.

It is worth noting that the confidence intervals in all cases are extremely narrow. In high-load cases where the system performance approaches instability, it is barely visible. Furthermore, the deviation of model results from simulations is evaluated in all cases and found to be less than 1.5% on average for all controller service rates.

4.4.3. Refined Model with General Service Processes

In this subsection, we analyze the results corresponding to the refined model presented in Subsection 4.3.3. The average sojourn time in the system is calculated using Eq. (4.12).

The simulation results of the system when using generally-distributed service times at the data plane and the controller revealed similar trends as those observed in the previous subsection when exponentially-distributed service times were adopted. For this reason, we will plot only the results corresponding to the t4p4s software switch with the three controller cases as an exemplary case in Fig. 4.8. The results of the refined model plotted in Figs. 4.8a, 4.8b, and 4.8c corresponding to the three different controller cases look very similar to the results of the simple model for t4p4s software



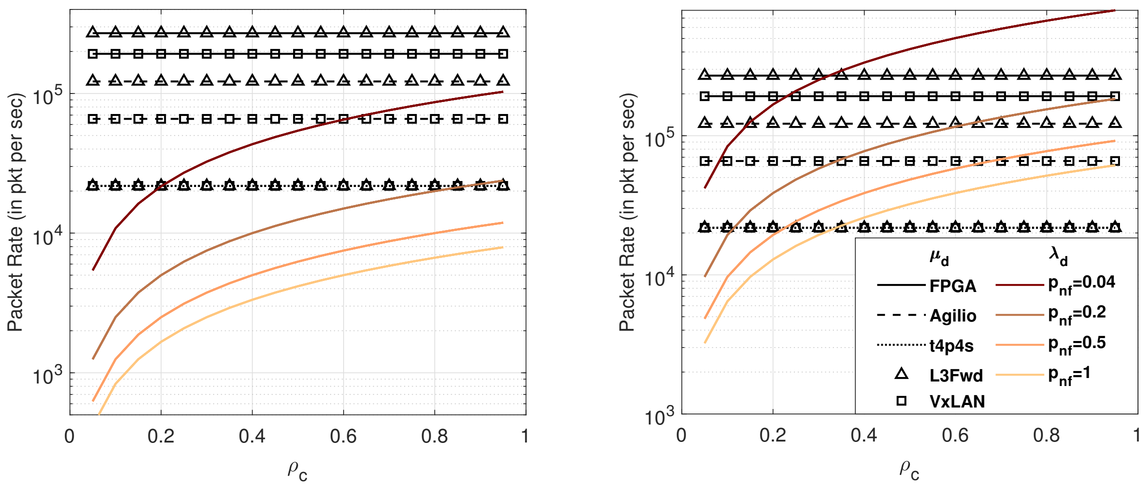
(a) Slow controller case. (b) Average controller case. (c) Fast controller case.

Figure 4.8.: Average sojourn time of the system with t4p4s software switch.

switch previously analyzed in Figs. 4.5c, 4.6c, and 4.7c, respectively. The impacts of the controller's load, P_{nf}, data plane's service time, and controller's service time on the average sojourn time of the system are still the same in the refined model case compared to the previously analyzed simple model case. Moreover, the sudden jumps in the sojourn time of the system due to the over-utilization of the data plane's queue are also visible in this case.

The main difference when adopting Erlang-distributed service times at the data plane and the controller in this refined model compared to the simple model with exponentially-distributed service times is that the average sojourn time in the system is relatively smaller. For example, looking at the curves of the fast controller case with the t4p4s software switch, while sojourn time reaches 1000 μs in the simple model case in Fig. 4.7c, we can observe that it is always less than that value in the refined model case in Fig. 4.8c. The reason for this decrease in sojourn time is that the variability of service time in the refined model case with Erlang-distributed service times is smaller than that in the simple model case with exponentially-distributed service times. This leads to smaller queueing delays in the data plane's and controller's queues and thus results in a smaller average system's sojourn time.

We can also observe that the theoretical results derived from the refined model very well match the system's performance revealed by simulations. The error in the model is found to be less than 12% in all cases when the system is not highly utilized (ρ_d and ρ_c are both less than 90%). When either of the two queues in the data plane or control plane approaches full utilization, the accuracy of the model decreases. In general, the reason for this decreased accuracy in the model compared to the simple model case is the memoryless assumption related to the arrival process to the controller and that to the data plane back from the controller. This assumption holds better in the simple model case with memoryless exponentially-distributed service times and departures compared to the refined model that adopts different service distributions at the data plane and controller.



(a) Controller's service time set to 240 μs. (b) Controller's service time set to 31 μs.

Figure 4.9.: Service rate and arrival rate at the data plane for different cases.

4.4.4. Derived Constraint for System Dimensioning

In this subsection, we derive a constraint for dimensioning input traffic to the system while avoiding packet drop. The utilization of a queue should always be less than one to keep it stable. In the feedback-oriented model under consideration, the system is not stable if any of the two queues at the data plane and the controller is not stable.

The system's utilization can be derived as follows. The system is empty, only if both the data plane and the controller queues are empty. The latter happens with probability equal to $(1 - \rho_d) * (1 - \rho_c)$. Accordingly, the system utilization can be expressed as $\rho_{sys} = 1 - (1 - \rho_d) * (1 - \rho_c)$.

During the previous evaluation, we noticed that the sojourn time increased arbitrarily large in some cases indicating that the system is over-utilized. In the controller's case, the load never exceeds one because we explicitly vary it between 0.05 and 0.95. However, this could be the case for the data plane's queue, wherein the packet arrival rate to the data plane surpasses the data plane's service rate. When we express the external arrival rate as a function of controller load as $\lambda_{ext} = (\mu_c * \rho_c)/P_{nf}$ and substitute it into Eq. (4.8) of the data plane's arrival rate, we get the following constraint

$$\mu_d^D(NF) > \frac{\mu_c * \rho_c * (1 + P_{nf})}{P_{nf}}. \tag{4.13}$$

Given the expected traffic characteristics for a use case scenario, this constraint can aid in the selection of the appropriate P4 target for running the intended network functionality based on its service rate or inversely its average forwarding latency.

Alternatively, the constraint can be used to control the maximum amount of incoming external traffic into a deployed P4 device with a specific service rate such that the traffic is handled without packet drops.

Fig. 4.9 depicts the arrival rate to the data plane (right-hand side term of 4.13) as a function of ρ_c for various P_{nf} values in logarithmic scale. In addition, the service rates corresponding to the various P4 pipelines and P4 devices are plotted in this figure, where these rates are calculated as the inverse of the previously provided service times. Figs. 4.9a and 4.9b correspond to the average and fast controller cases where service rates are set to (240 μs)$^{-1}$ and (31 μs)$^{-1}$, respectively. Note that the results corresponding to the slow controller with service time equal to 10 ms are not shown because data plane utilization is always stable in this case. Also, we skip plotting the arrival rate curves when $P_{nf} = 0$ because the data plane utilization is explicitly set to less than 1 in these cases. The data plane system, and thus the overall system, is stable as long as the service rate is greater than the arrival rate with different P_{nf} values.

Figs. 4.9a and 4.9b show that the constraint presented in Eq. (4.13) is violated at P_{nf} and ρ_c values that match the cases observed in Figs. 4.6 and 4.7, respectively, where the sojourn time increased arbitrarily and thus was not plotted. For example, let us examine the case where the controller service time is equal to 240 μs, as shown in Fig. 4.9a. It can be seen that the arrival rate for $P_{nf} = 0.5$ and 1 is never greater than any of the data plane service rates, meaning that the system is never overloaded. On the other hand, the system is overloaded when the arrival rate corresponding to $P_{nf} = 0.04$ crosses the service rate of the t4p4s switch for both pipelines after $\rho_c = 0.2$ and crosses the service rate corresponding to running the VxLAN pipeline on the Agilio CX SmartNIC after $\rho_c = 0.6$. Similarly, the arrival rate when $P_{nf} = 0.2$ crosses the service rate associated with the t4p4s switch after $\rho_c = 0.85$. The same analysis can be performed by cross-referencing the results presented for the fast controller case with Fig. 4.9b. Note that this analysis is conducted based on the simple model results, but it also holds the same in the refined model case.

4.5. Summary

In this chapter, we proposed performance models for P4-based systems. These models recognize the different factors that can influence the performance of the system and incorporate them as parameters that can be tuned based on the tested scenario.

The modeling exercise is broken into two stages. In Section 4.2, we focus mainly on modeling the challenging part related to the dependency between the data plane's forwarding latency and the complexity of the loaded P4 program. In this direction, we make use of the measurements collected in Section 3.2 to derive a generic method for predicting the packet forwarding latency when running arbitrary P4 programs on different P4 packet processors. The proposed method is based on a three-step approach.

First, we build performance profiles for different P4 packet processors that quantify the base forwarding latency and the latency to execute different P4 operations. Then, we analyze the given P4 program to extract the constituting P4 operations. Finally, we estimate the packet forwarding latency using the program's extracted information and the built P4 targets' performance profiles. The proposed method is validated by applying it to three realistic network functions running on three P4 targets, where the recorded estimation accuracy always exceeded 95%.

The method proposed in Section 4.2 served as the first building block for deriving more comprehensive analytical performance models for P4-based systems, where the estimated forwarding latency at the data plane characterizes the service process on that path. Section 4.3 uses the previous results to propose two queueing theory-based models that abstract the performance of P4-based systems as feedback-oriented queues where all the factors that influence the performance of this system are included in the models. The first model assumes exponentially-distributed service times to keep the model simple, while the second model relaxes this assumption and adopts generic service processes whose distribution is based on measurements to provide more realistic performance predictions.

Finally, Section 4.4 evaluates the two models by varying a wide set of parameters and analyzing their impact on the packet's sojourn time in the system. The model results are validated through simulations. In general, we observed that the results of the two models are very similar when varying any of the influential parameters, except that the refined model has lower sojourn time values compared to the model with exponentially-distributed service times. The accuracy of the two models compared to simulation results is found to be very high. This evaluation leads to deriving constraints for dimensioning the permissible input traffic that can be handled by the system without packet drops.

5. Performance-Aware Management of P4-based Cloud Environments

The optimal planning and management for the deployment of P4 programmable packet processors in cloud environments are studied in this chapter. When P4 programmability is integrated into cloud environments to process NF workloads, they create a new environment, which we call: **P4-enhanced NFV environment**.

P4 language is target or device-independent, meaning that the same P4 program can be used to configure different types of P4 programmable devices such as software switches, NPUs, FPGAs, or ASIC switches. When these different devices are used as a substrate for building cloud environments, they form a heterogeneous pool of processing resources. The measurements conducted in Chapter 3 revealed that there is a large difference in the forwarding performance of the different SOTA P4 targets. For example, when running the same P4 program on different P4 devices, the forwarding latency could vary by more than 12 folds as in the case when running the L3Fwd P4 program on the NetFPGA-SUME card versus running it on t4p4s software switch. On the other hand, these measurements also revealed that even when using the same P4 device, the processing latency could increase by up to 85% when the complexity of the P4 program increases as in the case when running L3Fwd P4 program versus VxLAN Decapsulation on the Agilio CX SmartNIC. The distinct performance and capabilities of the different available P4 programmable processing resources require smart management of these resources when selecting the processing platform that best hosts a certain workload with given QoS requirements for a certain type of functionality. The optimal management of these cloud environments enables enhancing the performance of the overall system in terms of forwarding performance, which is a valuable objective. For example, [120] quotes that a 1-millisecond latency advantage in trading applications can be worth 100 Million dollars a year to a major brokerage firm.

Thanks to the models derived in Chapter 4, we have a thorough understanding of the forwarding latency when running an arbitrary P4 program on an arbitrary P4 target. This **performance-awareness** enables us to manage the P4-enhanced NFV environments in a highly efficient way, wherein we can satisfy QoS requirements in terms of delay as we can predict a priori the forwarding latency of embedding any NF on any network substrate made up of P4 devices.

In this chapter, we consider two optimization problems related to managing P4-enhanced NFV environments, wherein each problem considers a different stage of management, and results in a different problem setup and formulation: (i.) The first problem is concerned with the offline optimal planning for selecting and building a network's substrate, and (ii.) the second problem is concerned with optimizing the runtime management of an already given built-up network.

The first problem called Performance-Aware P4 Virtual Network Functions Resource Allocation (PA-P4VNF-RA) deals with the planning of the infrastructure substrate of P4-enhanced NFV environments. The optimization problem looks for the optimal set of P4 packet processors that can handle a given processing workload and the placement solution of this workload into the selected hosting devices. The different requirements of the NFs workload and the distinct performance and capabilities of the available P4 devices in the cloud infrastructure are taken into consideration while formulating the problem. The objective function targets maximizing the performance in the system while minimizing the capital expenditure costs when selecting the optimal set of P4 packet processors that can handle a given processing workload. Note that the contributions related to this problem are based on our early publication [7].

The second optimization problem, called Performance-Aware P4 Service Function Chain Embedding (PA-P4SFC-E), targets finding the optimal embedding of SFCs into P4-enhanced NFV environments at runtime. The optimization problem formulation searches for the optimal placement and routing of SFCs into P4 packet processors. The functional and QoS requirements of these SFCs are fulfilled by leveraging the acquired knowledge from previous chapters related to the performance and capabilities of the different available P4 packet processors. Furthermore, a greedy solution is designed and implemented to solve this problem in a shorter time.

The rest of this chapter is organized as follows. In Section 5.1, we define the context and objective of the two studied optimization problems, as well as the distinction between these problems from state-of-the-art literature. In Sections 5.2 and 5.3, we define the mathematical formulation and present the evaluation of the two studied optimization problems PA-P4VNF-RA and PA-P4SFC-E, respectively. Finally, Section 5.4 concludes the chapter.

5.1. Problem Context and Related Work

In Subsection 5.1.1, we describe the scenario related to the two optimization problems addressed in this chapter. Then, we distinguish the scope of these problems from state-of-the-art works in the literature in Subsection 5.1.2.

5.1.1. Problem Context

The recent trend of deploying programmable packet processors in cloud environments improves packet processing capability without sacrificing the ability to adapt functions at runtime. However, managing network functions, particularly deciding where to instantiate a specific function, is a difficult task with numerous deciding factors. On one hand, the management scheme should take into consideration that the NFs to be managed in the environment have different functional and performance requirements. For example, a NF can be compatible with specific P4 architecture, or it can require certain acceleration functions, or it can demand some QoS levels in terms of delay or throughput, etc. On the other hand, the P4-enhanced NFV environment is made up of a heterogeneous pool of P4 packet processors with different capabilities and performance levels. For example, each P4 packet processor may support different P4 architectures, may include certain acceleration functions, or have different performance levels as analyzed in Chapter 3. To this end, we propose a management scheme that targets optimizing the deployment of network functions with distinct requirements into a P4-enhanced NFV environment made up of different P4 packet processors with distinct capabilities. The management scheme addresses the following two optimization problems:

PA-P4VNF-RA Problem

This problem stands for Performance-aware P4 Virtualized Network function Resource Allocation. In this problem, we are interested in the scenario where we want to plan for the infrastructure of the P4-enhanced NFV environment. **Given an expected workload of NFs that need to be processed in a cloud environment, the problem searches for the best set of P4 packet processors that can host the NF workload while maximizing the forwarding performance and minimizing the capital expenditure costs.**

Fig. 5.1 depicts the considered scenario. The workload to be processed by the environment is presented as a set of NFs written as P4 programs. If there is branching in the SFC definition, we consider each path of the acyclic graph as one sequential SFC. Each NF has different functional requirements such as compatible P4 architecture or required acceleration functions (expressed in terms of P4 externs), etc. Moreover, each NF describes a different processing pipeline written as a different P4 program with a different set of constituting P4 atomic constructs. We extract the atomic P4 constructs of each P4-based NF using the methodology described in Subsection 4.2.2. The model also checks for the possibility of NF sharing among the requested set of SFCs to be placed to avoid duplicate packet processing and save processing resources.

On the other hand, the problem recognizes that candidate P4 packet processors belong to different processing platforms, and thus they have different functional capabilities and limitations, as well as different performance levels. The optimization problem uses

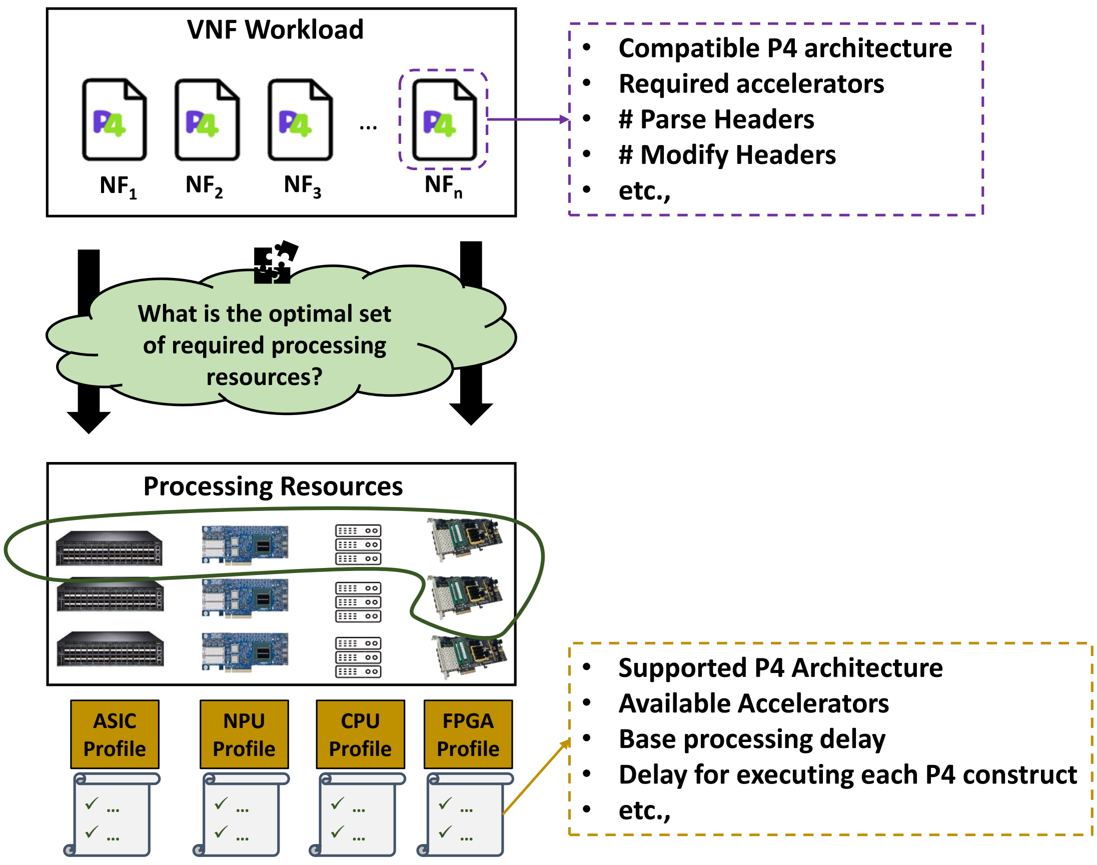


Figure 5.1.: Considered scenario in PA-P4VNF-RA problem.

the measurements conducted in Chapter 3 to identify the limitations and capabilities of hosting P4 devices to constraint the placement of NFs into P4 packet processors based on the requirements of NFs and the capabilities of the candidate P4 packet processors. These constraints ensure the compatibility between the NF and the hosting P4 device. The checked constraints include and are not limited to the following: compatibility of P4 architecture, availability of acceleration functions, sufficient throughput, availability of memory resources in terms of the total number of rules that can be placed, capacity in terms of the maximum number of atomic P4 constructs that can be supported on a device. Moreover, the problem uses the predeveloped performance models in Chapter 4 to make a priori calculations of the forwarding delay that may result from different placement options. This performance awareness enables finding the optimal placement solution that achieves the highest forwarding performance in the system or guarantees some QoS levels associated with the placement requests.

The multi-objective optimization function searches for the best set of hosting P4 packet processors and the optimal placement of NFs into this set of devices while minimizing both the forwarding latency in the system and the capital expenditure.

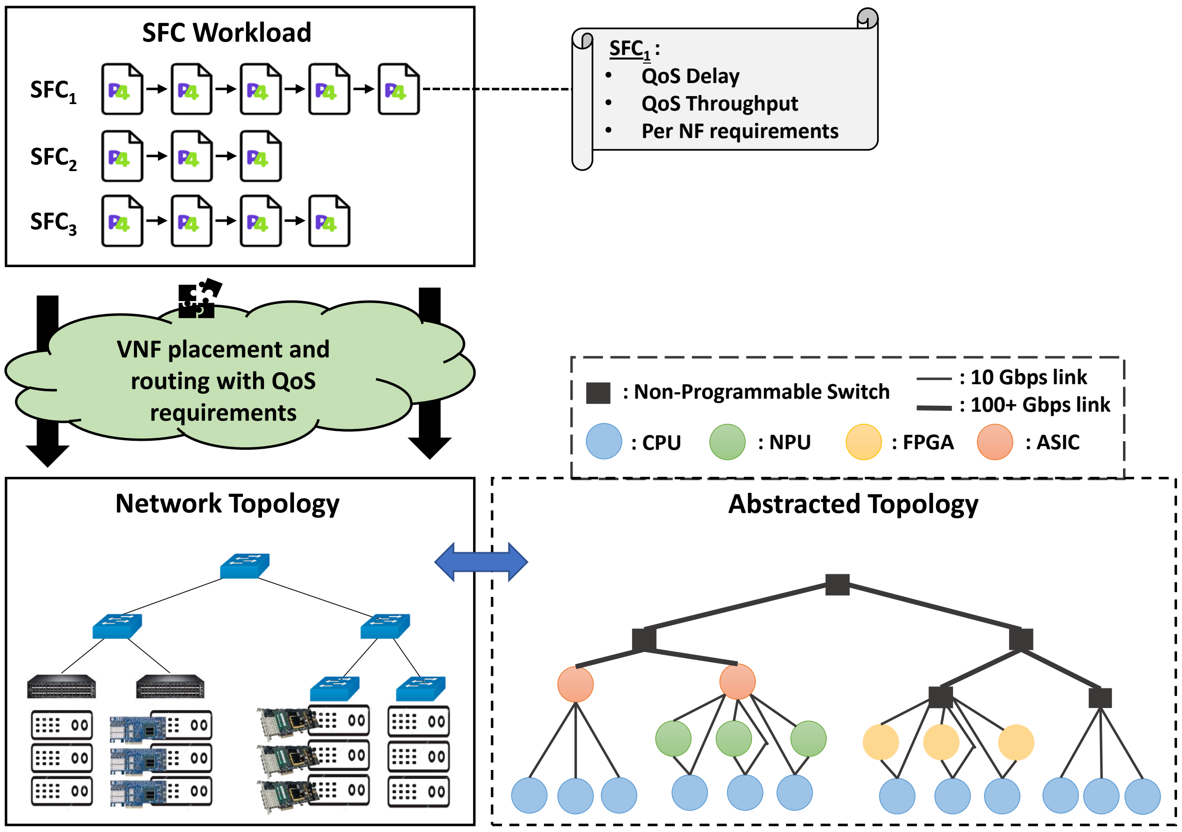


Figure 5.2.: Considered scenario in PA-P4SFC-E problem.

PA-P4SFC-E Problem

The second problem is more relevant for the runtime management of the P4-enhanced NFV environment. As its name states, performance-aware P4 SFC embedding, this problem deals with the embedding of SFCs into the P4 substrate. Fig. 5.2 illustrates the considered scenario.

Unlike the previous scenario, in this one, we consider that the infrastructure of the P4-enhanced NFV environment is already given. Although the formulated model can be solved for any topology, the commonly used "Fat Tree" topology is assumed in this scenario. The infrastructure is made up of racks of servers with CPU processors. Each server can be equipped with an NPU-based SmartNIC or an FPGA-based SmartNIC as depicted in Fig. 5.2. The servers and the two types of SmartNICs can support P4 programmability. The racks are connected via Top of Rack (ToR) switches. These ToR switches could be P4 programmable ASIC switches or traditional switches that only perform Layer 2 Forwarding (L2Fwd) functionality. Finally, the interconnection of the racks is achieved via traditional L2Fwd switches. The abstract representation of this network is shown in Fig. 5.2, where different P4 packet processors are presented as nodes with different colors. The links between the SmartNICs and the CPU-based servers abstract the PCIe bus connectivity between these devices when SmartNICs are plugged into the servers. On the other hand, the connections between the SmartNICs and the ToR switches present the case when the physical interfaces of the SmartNICs are connected to the ToR switches. For the sake of generality, the scenario recognizes the cases when traditional non-programmable switches are used in the infrastructure for interconnection. We assume that all P4 packet processors can process all their incoming traffic at line rate, so only the capacity of the connecting links (and not the devices) is considered in throughput constraints. It is also assumed that all P4 programmable devices in the network can serve as ingress or egress nodes for the traffic processed in the network. So each node is assumed to have another pair of ports that can serve as ingress and egress interfaces for traffic to enter or leave the network. The throughput constraint is also applied for these ingress and egress ports according to the throughput capacity of each P4 packet processor.

The workload in this scenario is made up of SFCs that need to be embedded into the P4 substrate. Each SFC is a connected sequence of P4-based NFs. Besides the requirements described in the previous scenario associated with each constituting NF, each SFC also has its QoS requirements related to the throughput and delay that need to be satisfied as shown in Fig. 5.2. The model also realizes that some SFCs are built of the same type of NFs. So, if the model decided to place two SFCs that have a common NF on the same device, it will place this common NF only once on the hosting device and let the two SFCs share this single placed function. NF sharing is favored by the model when applicable because when SFCs share an NF, there is no need to instantiate a duplicate NF on the same device as this would waste processing resources on the hosting device. Note that the model includes a constraint that checks the available processing resources on each device which are limited based on the device type.

The problem in this scenario is to search for the optimal embedding of a workload of SFCs into a given P4 infrastructure, while satisfying all the functional and QoS constraints associated with each SFC.

The component NFs of each SFC should be properly placed into P4 packet processors while recognizing the requirements of the NFs and the capabilities of the P4 devices as described in the previous scenario. Moreover, the routing between the NFs constituting an SFC should also be provisioned. Similar to the previous scenario, the predeveloped performance models are utilized in the formulated problem to perform a priori calculations of the delays associated with different embedding options until finding the optimal embedding that satisfies the required QoS levels per SFC. The objective function in this scenario targets minimizing the operational cost in the system by minimizing the total power consumed by the operating P4 devices.

5.1.2. Related Work

Since the introduction of NFV technology, many works were introduced to address the resource allocation and SFC embedding problems. More about these works can be found in these two surveys: [67] and [66]. For example, VNF Orchestration Problem (VNF-OP) [65] presents a comprehensive problem formulation for the placement and mapping of SFC requests. The authors used commodity servers to host a variety of NFs that were instantiated based on the flow requirements. However, this paper does not consider function accelerators or P4 programmability as needed in our considered scenario.

While most solutions in the literature, such as VNF-OP [65], deal with NF placement on commodity servers, Accelerator-aware VNF Placement and Chaining (VNF-AAPC) [64] considers also accelerators in the substrate when dealing with the placement problem. In the formulation of VNF-AAPC, the accelerator and server are considered as a single node. If a compatible accelerator is available, the server-accelerator node uses fewer resources (i.e., cores) to host an NF. Although VNF-AAPC recognizes accelerators in the problem formulation, it does not consider P4 programmability and the fact that different accelerators may execute different functions, which requires proper mapping between the requirements of the NF workload and the availability of acceleration functions on the hosting devices.

In its problem formulation, P4NFV [33] takes P4 programmability into account. The authors of this paper are interested in determining the best NF placement between a SmartNIC and its hosting server. However, they do not consider the placement problem at the level of a full network which may be made up of heterogeneous types of P4 programmable devices. Smartchain [62] is another work that targets finding the optimal placement of an SFC between the SmartNIC and the CPU of a device, similar to [33], but without utilizing the P4 programmability.

Flightplan [63] enables the placement of P4 programs on a network of devices by utilizing coarse segmentation of the P4 programs. The model presented in Flightplan divides a P4 program into several smaller P4 programs that can run on different P4 targets. It is stated that this would improve the overall network performance and resource utilization. Although Flightplan takes into consideration the performance of segmented P4 programs, the applicability and usefulness of this performance awareness are hindered because of the coarse granularity in segmenting P4 programs compared to the approach followed in our work, where we consider the performance awareness at the level of atomic P4 operations.

In this chapter, we investigate the optimal NF placement and routing in cloud environments comprised of various P4 devices. The planning aims to optimize the placement of NFs while minimizing the overall delay and cost. The applied method in this work takes into account the characteristics of cutting-edge P4 devices at the granularity of the execution of various P4 atomic constructs. Each NF is decomposed into a set of

Table 5.1.: Summary of the related works and their commonalities with the problems studied in this chapter.

Problem Scope	*VNF-OP* [65]	*VNF-AAPC* [64]	*P4NFV* [33]	*Flightplan* [63]	*Work in this Chapter*
VNF Placement	✓	✓	✓	✓	✓
SFC Embedding	✓	✓	✓	✓	✓
Placement into a Network of Devices	✓	✓	X	✓	✓
Quality of Service Requirements	✓	X	X	✓	✓
Availability of Acceleration Functions	X	✓	X	✓	✓
P4 Programmability	X	X	✓	✓	✓
Heterogenous P4 devices	X	X	X	✓	✓
Performance Awareness based on Atomic P4 constructs	X	X	X	X	✓

atomic P4 constructs in order to find the best-performing P4 device that can support running that NF. In contrast to the SFC embedding problem, which is widely studied in the literature, this problem is still not addressed when the embedding is performed into a heterogeneous pool of P4 packet processors with performance-awareness taken into consideration for satisfying QoS requirements. Table 5.1 represents a summary of the related works in comparison to the problems studied in this chapter.

5.2. PA-P4VNF-RA Problem

In this section, we describe the PA-P4VNF-RA problem formulation and its evaluation. The contributions presented in this section are based on our publication [7]. First, we model the infrastructure of the P4-enhanced NFV environment and the capabilities of the P4 packet processors in Subsection 5.2.1. The description of the workload in terms of NFs requirements is illustrated in Subsection 5.2.2. The formulation of the optimization problem under study, i.e., PA-P4VNF-RA, is provided in Subsection 5.2.3, where the decision variables, objective functions, and constraints are described. In Subsection 5.2.4, we illustrate the selection of system parameters. The selection of the parameters related to NFs and P4 devices is based on the evaluation conducted in Chapter 3 and other works surveyed from the literature. Finally, a detailed evaluation of the PA-P4VNF-RA problem is conducted in Subsection 5.2.5, wherein different scenarios are defined to vary the weights of the two objective functions, i.e., the cost and the performance.

5.2.1. Infrastructure Variables and P4 Devices' Characteristics

The target-independence feature of P4 allows the same P4 program to run on different types of P4 devices if a compatible P4 architecture is supported on these devices. The P4 architecture defines the programmable blocks in a P4 device, as well as the supported externs. These extern functions are additional methods supported by the P4 device that can be called from within a P4 program via a given API. For hardware P4 packet processors, externs can be used to access built-in acceleration functions such as an encryption function, while for software P4 packet processors, externs can be used to access software implementations of the referenced external function.

We define $\mathcal{D}$, $\mathcal{D}^P$, and $\mathcal{D}^N$ to be the sets of all devices, all P4 programmable devices, and all non-programmable switches in a cloud environment, respectively ($\mathcal{D} = \mathcal{D}^P \cup \mathcal{D}^N$). The set $\mathcal{D}^P$ may include multiple instances of devices belonging to the same type of processing platforms such as NetFPGA-SUME cards. In this case, these instances have the same device capabilities. The two sets $\mathcal{A}$, and $\mathcal{E}$ are defined to include all the supported P4 architectures, and the available P4 extern functions in a cloud environment, respectively.

Each P4 device $d \in \mathcal{D}^P$ has the following characteristics:

- δ_d^{BP} stands for the base processing delay of P4 device d, which includes the delay of the non-programmable blocks in the device.
- δ_d^c stands for the processing delay of a P4 construct c on P4 device d.
- δ_d^e stands for the processing delay of running a P4 extern function $e \in \mathcal{E}$ on P4 device d.
- ω_d stands for the processing resources of P4 device d whose definition depends on the type of the processing platform. For example, while the processing resources in the case of FPGA-based devices are expressed in terms of the available Look-up-tables, these resources are expressed in terms of the number of available stages in the case of ASIC devices. To have a common definition, we quantify the processing resources of a P4 device as the maximum number of P4 constructs that can run simultaneously on that device.
- τ_d stands for the memory space in terms of the total number of rules that can be stored in a P4 device d.
- P_d stands for the expected power consumption by P4 device d when it is active. The power consumption of the device is assumed to be constant if the device is in use, and zero if it is not in use.
- $Cost_d$ stands for the Capital expenditures (CAPEX) cost or price of P4 device d.

Table 5.2.: Description of symbols used for modeling the cloud infrastructure variables and P4 devices' characteristics.

Symbol	**Infrastructure Symbols Description**
$\mathcal{D}^P$	Set of P4 programmable devices
$\mathcal{D}^N$	Set of non programmable devices
$\mathcal{D}$	Set of all devices $\mathcal{D} = \mathcal{D}^P \cup \mathcal{D}^N$
$\mathcal{X}$	Set of all physical links in the network
$\mathcal{A}$	Set of possible P4 architectures
$\mathcal{E}$	Set of possible P4 extern functions
δ_d^{BP}	Base processing delay on a P4 device $d \in \mathcal{D}^P$
δ_d^c	Processing delay of executing P4 construct c on device $d \in \mathcal{D}^P$
δ_d^e	Processing delay of a P4 extern function e on device $d \in \mathcal{D}^P$
ω_d	Available processing resources in device $d \in \mathcal{D}^P$
τ_d	Number of rules that can be stored in P4 device $d \in \mathcal{D}^P$
P_d	Power consumption of P4 device $d \in \mathcal{D}^P$ when active
$Cost_d$	Price of P4 device $d \in \mathcal{D}^P$
$Arch_d^a$	Boolean parameter equal to 1 if device $d \in \mathcal{D}^P$ supports architecture $a \in \mathcal{A}$
Ext_d^e	Boolean parameter equal to 1 if device $d \in \mathcal{D}^P$ supports extern $e \in \mathcal{E}$
T_d	Supported throughput by P4 device $d \in \mathcal{D}^P$ on each of its interfaces
T_d^e	Supported throughput when running extern $e \in \mathcal{E}$ on P4 device $d \in \mathcal{D}^P$
$T(d_i, d_j)$	Capacity of physical link $(d_i, d_j) \in \mathcal{X}$ for $d_i, d_j \in \mathcal{D}$
k_d	Delay of non programmable devices $d \in \mathcal{D}^N$

- $Arch_d^a$ is a Boolean parameter equal to 1 if P4 device d supports architecture $a \in \mathcal{A}$, and equal to 0 otherwise.
- Ext_d^e is a Boolean parameter equal to 1 if P4 device d supports extern $e \in \mathcal{E}$, and equal to 0 otherwise.
- T_d stands for the maximum supported throughput by P4 device d on each of its interfaces.
- T_d^e denotes the maximum supported throughput when running extern e on P4 device d.

Each non-programmable switch $d \in \mathcal{D}^N$ is characterized by a constant forwarding delay denoted by k_d. When these programmable P4 devices and the non-programmable switches are connected with physical links, they create a network. Let $\mathcal{X}$ denote the set of all physical links in the network and $T(d_i, d_j)$ denote the capacity of physical link $(d_i, d_j) \in \mathcal{X}$ for $d_i, d_j \in \mathcal{D}$. Table. 5.2 summarizes the description of all the symbols used for modeling the cloud infrastructure variables and P4 devices' characteristics defined in this subsection.

5.2.2. Network Function Requirements

The processing workload that needs to be supported by the cloud infrastructure is made up of NFs. In the following, we describe the requirements associated with NFs workload.

We denote $\mathcal{F}$ to be the set that includes the NF instances workload that should be placed into the network. The set $\mathcal{Y}$ includes all the possible **types** of NFs. For example, there could be more than one instance of Firewall NF in $\mathcal{F}$, but all these NFs have the same type $y \in \mathcal{Y}$. The binary variable ψ_f^y is defined to be equal to 1 when NF $f \in \mathcal{F}$ is of type $y \in \mathcal{Y}$. Note that some functionalities, such as the L2Fwd or L3Fwd, need to run on every used packet processor to guarantee proper packet forwarding between devices. The set of required functions is denoted by $\mathcal{F}_{req}$. Set $\mathcal{F}_{tot} = \mathcal{F} \cup \mathcal{F}_{req}$ includes all the NFs defined in a scenario.

An NF written as a P4 program is made up of a group of atomic P4 operations as described in Subsection 4.2.2. We define set $\mathcal{C}$ to include all the possible P4 constructs that may be used to write a P4 program. Any NF of type $y \in \mathcal{Y}$ can be written as a combination of the P4 constructs contained in $\mathcal{C}$. The set of P4 constructs that are needed to compose a NF instance of type y is denoted by $\mathcal{C}_y$, where $\mathcal{C}_y \subset \mathcal{C}$. The following variables summarize the requirements associated with any NF f of type y:

- σ_y^c represents the number of occurrences of each construct $c \in \mathcal{C}_y$ needed to build a NF of type y.
- ω_y represents the total number of P4 constructs required to describe a NF of type y, i.e., $\omega_y = \sum_{c \in C_y} \sigma_y^c$. This variable reflects the complexity of the NF, and consequently, the expected required processing resources by this NF.
- $Arch_y^a$ is a Boolean parameter equal to 1 if the NF of type y is compatible with P4 architecture $a \in \mathcal{A}$, and equal to 0 otherwise.
- Ext_y^e is a Boolean parameter equal to 1 if the NF of type y requires extern $e \in \mathcal{E}$, and equal to 0 otherwise.
- τ_f represents the expected required number of rules that need to be stored when hosting NF instance $f \in \mathcal{F}$.
- QoS_f^T represents the expected throughput that needs to be processed by NF instance $f \in \mathcal{F}$.

Table. 5.3 summarizes the description of all the symbols used for modeling NFs workload and their associated requirements.

Table 5.3.: Description of symbols used for modeling NFs workload and their associated requirements.

Symbol	Workload Symbols Description
$\mathcal{F}$	Set of all NF instances to be placed
$\mathcal{F}_{req}$	Set of required NFs for forwarding
$\mathcal{F}_{tot}$	$\mathcal{F} \cup \mathcal{F}_{req}$
$\mathcal{Y}$	Set of all types of NFs
$\mathcal{C}$	Set of all P4 constructs/operations
$\mathcal{C}_y$	Set of P4 constructs used to build a NF of type $y \in \mathcal{Y}$
ψ_f^y	Boolean parameter equal to 1 if NF f is of type y
σ_y^c	Number of occurrence of each construct c in $\mathcal{C}_y$ needed to build an NF of type y
ω_y	Total number of P4 constructs required to describe NF of type y
QoS_f^T	Expected throughput that should be processed by NF instance $f \in \mathcal{F}$
τ_f	Number of rules that need to be stored when hosting NF instance $f \in \mathcal{F}$
$Arch_y^a$	Boolean parameter equal to 1 if NF of type y is compatible with P4 architecture $a \in \mathcal{A}$
Ext_y^e	Boolean parameter equal to 1 if NF of type y requires extern $e \in \mathcal{E}$

5.2.3. Problem Formulation

The PA-P4VNF-RA problem searches for the best set of P4 devices that can host a given workload of NFs while minimizing the forwarding latency and the capital expenditure cost. For this problem, we need the predeveloped performance models to calculate a priori the forwarding latency corresponding to different NF placement options. In the following, we first describe how the delay of different NF placement options is calculated, and then we define the objective function and constraints related to the PA-P4VNF-RA problem.

NF Delay Calculation

The formulation of this problem makes use of the studies presented in Chapters 3 and 4 related to understanding the performance of different P4 devices and their performance variation at the level of atomic P4 constructs. This enables quantifying the expected forwarding delay of all NF placement options of any P4 device.

Using the previously provided formulation of the requirements of any NF of type $y \in \mathcal{Y}$ and the capabilities of any P4 device $d \in \mathcal{D}^{\mathcal{P}}$, we can calculate the delay that will result from any NF placement option. This delay is calculated as the summation of the base processing delay of the hosting P4 device, denoted by δ_d^{BP}, and the delay related to processing the programmable logic in the device, denoted by Δ_d^y. The latter delay component is expressed as follows:

$$\Delta_d^y = \sum_{c \in C_y} \sigma_c^y \delta_d^c + \sum_{e \in E | Ext_f^e = 1} \delta_d^e, \tag{5.1}$$

where it is equal to the delay of processing the different P4 constructs that constitute the NF of type y, and the delay for executing extern functions required by this NF.

Objective Function

The objective of this problem is to minimize the overall packet forwarding delay $\mathbb{D}$ in the considered environment, as well as the overall CAPEX costs $\mathbb{C}$, each expressed as:

$$\mathbb{D} = \sum_{d \in \mathcal{D}} \sum_{f \in \mathcal{F}_{tot}} \sum_{y \in \mathcal{Y}} \alpha_d^f \times \psi_f^y \times (\delta_d^{BP} + \Delta_d^y), \tag{5.2}$$

$$\mathbb{C} = \sum_{d \in \mathcal{D}} x_d \times Cost_d, \tag{5.3}$$

where the Boolean decision variable α_d^f is equal to one if NF f is placed on P4 device d, and the dependent variable x_d indicates whether a P4 device d is being used:

$$x_d = \begin{cases} 1 & \text{if } \sum_{f \in \mathcal{F}_{tot}} \alpha_d^f \geq 1 \\ 0 & \text{otherwise.} \end{cases} \tag{5.4}$$

The final goal is a multi-objective function that targets minimizing both the forwarding delay and the cost of the system. This function is formulated as a weighted sum of the two metrics $\mathbb{D}$ and $\mathbb{C}$ as shown in Eq. (5.5):

$$Minimize\ (\mu\mathbb{D} + \varepsilon\mathbb{C}). \tag{5.5}$$

As the two considered metrics have different ranges and units (μs and dollars), they are normalized according to the maximum value recorded for each metric. This is necessary to make sure that both objectives affect the placement decision equally without one objective overshadowing the other. The weights μ and ε are used to tune the relative importance of each of the two metrics in the optimization problem.

Constraints

The placement decision should recognize a set of constraints. First, each NF f in $\mathcal{F}$ should be placed just once on any P4 device, as expressed in Eq. (5.6):

$$\sum_{d \in \mathcal{D}^{\mathcal{P}}} \alpha_d^f = 1 \ \forall f \in \mathcal{F}. \tag{5.6}$$

As mentioned earlier, some functions in $\mathcal{F}_{req}$ should be placed on every device used. For example, in this problem, it is assumed that L3Fwd functionality is needed on each operating device. This requirement can be achieved by the constraint shown in Eq. (5.7):

$$\alpha_d^f = x_d \ \ \forall d \in \mathcal{D}^{\mathcal{P}} \ \forall f \in \mathcal{F}_{req}. \tag{5.7}$$

The dependent variable π_d^y indicates whether any NF instance of type y is placed on device d:

$$\pi_d^y = \begin{cases} 1 & \text{if } \sum_{f \in \mathcal{F}_{tot}} \alpha_d^f \times \psi_f^y \geq 1 \\ 0 & \text{otherwise.} \end{cases} \tag{5.8}$$

We make sure that a NF f of type y can be hosted on a device d only if the device has a compatible architecture and supports all the extern functions required by the NF of type y. In Eq. (5.9), the decision variable π_d^y is forced to be zero in case the architecture required by the NF and that supported by the P4 device do not match. On the other hand, Eq. (5.10) makes sure that π_d^y can be equal to 1 only if all the extern functions required by NF of type y are available on device d. Note that N is a large constant bigger than the maximum number of externs required by any NF.

$$\pi_d^y \leq \sum_{a \in \mathcal{A}} Arch_y^a \times Arch_d^a \ \ \forall y \in \mathcal{Y} \ \forall d \in \mathcal{D}^{\mathcal{P}}, \tag{5.9}$$

$$\sum_{e \in \mathcal{E}} Ext_y^e \leq \sum_{e \in \mathcal{E}} Ext_y^e \times Ext_d^e + N(1 - \pi_d^y) \ \ \forall y \in \mathcal{Y} \ \forall d \in \mathcal{D}^{\mathcal{P}}. \tag{5.10}$$

Eq. (5.11) makes sure that the required processing resources of all NFs to be placed on any P4 device d never exceed the limited processing capacity of that device. The problem realizes that if two instances of NFs of the same type y are placed on the same device d, then it is enough to place this NF only once to save processing resources on the hosting device. For this reason, the constraint limits the utilization of processing resources on a P4 device based on the requirements of different **types** of NFs hosted on that device, assuming that NFs of the same type will not be placed more than once on the P4 device.

$$\sum_{y \in \mathcal{Y}} \pi_d^y \omega_y \leq \omega_d \ \ \forall d \in \mathcal{D}^{\mathcal{P}}. \tag{5.11}$$

The required number of rules by all NFs to be placed on any P4 device d should never exceed the total memory space available on that device.

$$\sum_{f \in \mathcal{F}_{tot}} \alpha_d^f \tau_f \leq \tau_d \ \ \forall d \in \mathcal{D}^{\mathcal{P}}. \tag{5.12}$$

Table 5.4.: Description of variables and objectives used to formulate the PA-P4VNF-RA problem.

Symbol	**Symbols Description for PA-P4VNF-RA problem**
	Decision Variable
α_d^f	Boolean variable equals to one if NF f is placed on P4 device $d \in \mathcal{D}^P$
	Variables
x_d	Boolean variable equals 1 if any NF instance is placed on P4 device $d \in \mathcal{D}^P$
π_d^y	Boolean variable equals to 1 if any NF instance of type y is placed on P4 device $d \in \mathcal{D}^P$
Δ_d^y	Delay needed to process programmable logic (P4 blocks and externs) when running NF of type y on P4 device d
μ	Weighting factor between 0 and 1 for delay objective
ε	Weighting factor between 0 and 1 for cost objective
	Objectives
$\mathbb{D}$	Total forwarding delay in the system
$\mathbb{C}$	Total CAPEX cost to equip the system

On the other hand, Eq. (5.13) makes sure that the cumulative throughput required by different NFs to be placed on a device never exceeds the limited throughput of that device. Similarly, Eq. (5.14) ensures that the cumulative throughput of NFs that require offloading some functionalities to an extern function on a device never exceeds the limited throughput of that extern.

$$\sum_{f \in \mathcal{F}_{tot}} QoS_f^T \times \alpha_d^f \leq T_d \ \ \forall d \in \mathcal{D}^{\mathcal{P}}, \tag{5.13}$$

$$\sum_{f \in \mathcal{F}_{tot}} \sum_{y \in \mathcal{Y}} \alpha_d^f \times \psi_f^y \times Ext_y^e \times QoS_f^T \leq T_d^e \ \ \forall e \in \mathcal{E} \ \ \forall d \in \mathcal{D}^{\mathcal{P}}. \tag{5.14}$$

A constraint to limit the total CAPEX costs is added to ensure that the total cost of used devices according to the optimal placement solution never exceeds a predefined limit denoted by $MaxCost$ as shown in Eq. (5.15):

$$\sum_{d \in \mathcal{D}^{\mathcal{P}}} x_d \times Cost_d \leq MaxCost. \tag{5.15}$$

Finally, the following two constraints are introduced to set the dependent variable x_d to 1 when at least one NF is placed on device d. Note that M is a large constant bigger than the number of NF instances in the network.

$$x_d \leq \sum_{f \in \mathcal{F}_{tot}} \alpha_d^f \ \ \forall d \in \mathcal{D}^{\mathcal{P}}, \tag{5.16}$$

$$\sum_{f \in \mathcal{F}_{tot}} \alpha_d^f \leq M x_d \ \forall d \in \mathcal{D}^{\mathcal{P}}. \tag{5.17}$$

Table 5.4 summarizes the description of all the symbols used for the formulation of the PA-P4VNF-RA problem.

5.2.4. Surveying NFs' and P4 Devices' Parameters

To evaluate the proposed model, we select a scenario with a realistic set of NFs and P4 devices whose parameters are surveyed from the literature.

Surveyed NFs

We consider seven different NFs with different complexities: *(i)* L2Fwd, *(ii)* L3Fwd, *(iii)* Firewall (FW), *(iv)* VxLAN Decapsulation (VDecap), *(v)* Load Balancer (LB), *(vi)* Tunneling, *(vii)* Network Address Translation (NAT). In PA-P4VNF-RA problem, the workload is made up of these NFs, where L3Fwd network function is selected to be in the set of functions required to be in every functioning device to ensure proper routing on different used packet processors. On the other hand, the PA-P4SFC-E problem targets embedding SFCs made up of these NFs with L2Fwd being the NF that is required on all functioning devices that need to forward packets within the network. All NFs support the three most common P4 architectures: V1Model (V1M), SimpleSumeSwitch (SUME), and PISA). Only the LB NF requires the use of an external hashing function. Using the methodology described in Subsection 4.2.2, each NF is decomposed to its atomic P4 constructs. For example, L2Fwd NF necessitates parsing and modifying a single header (Ethernet), whereas L3Fwd necessitates the same operations for both Ethernet and IPv4 headers. Note that in the delay calculation of NFs, we subtract one from the number of parsed headers and one from the number of added tables when calculating the delay related to the P4 program to account for the presence of these two constructs in the base P4 program whose delay is included in the base processing delay component for each device. Table 5.5 summarizes the various parameters and requirements corresponding to all the considered NFs in this evaluation.

Surveyed P4 Devices

For this evaluation, we select four P4 programmable packet processors that belong to different types of processing platforms: CPU, NPU, FPGA, and ASIC.

The different criteria related to the different selected types of P4 devices are summarized

Table 5.5.: Surveyed parameters of different used Network functions.

	L2Fwd	**L3Fwd**	**FW**	**VDecap**	**LB**	**Tunneling**	**NAT**
Required	PA-P4SFC -E	PA-P4VNF -RA	no	no	no	no	no
Compatible P4 Arch.	V1M, PISA, SUME	V1M, PISA, SUME	V1M, PISA, SUME	V1M, PISA, SUME	V1M, PISA, SUME	V1M, PISA, SUME	V1M, PISA, SUME
Externs	none	none	none	none	SipHash-2-4	none	none
# Const.							
Parse Hdr.	1	2	3	7	3	3	3
Modify Hdr.	1	2	0	0	0	0	2
Copy Hdr.	0	0	0	3	0	0	0
Remove Hdr.	0	0	0	4	0	0	0
Add Hdr.	0	0	0	0	0	0	0
Add Table	1	1	1	2	1	1	1

in Table 5.6. The supported P4 architecture of each device is based on its datasheet. The throughput and the per P4 construct latency values are based on the evaluation results presented in Chapter 3 and another work in literature [32]. Note that the delay values used in this paper are collected when devices are running at line rate, and accordingly it is assumed that no queueing is taking place in the system's devices. The performance related to the extern "SipHash-2-4" function is derived from [40] for packet size equal to 64 Bytes since the Load Balancer function requires calculating the hash function of some headers only. The maximum number of supported constructs is based on the comparative analysis provided in [32] and [3]. Since no work in the literature gives a quantitative evaluation regarding the maximum number of supported constructs on different devices, we assume some reasonable numbers that follow the order of devices in terms of the available processing resources. The cost or price of devices resembles the real costs of the different types of P4 devices. For CPU-based switches, we assume that the cost is approximately equal to the price of 2 cores, which is the typical minimum requirement for running P4-based software switches. Because no evaluation in the literature quantifies the variation of latency on ASIC devices as a function of P4 constructs and P4 externs, it is assumed that ASIC-based devices are powerful enough to run any P4 program with no latency variation and to execute hashing functions with higher performance than other device types. Finally, the power consumption of each device is surveyed from the literature and the datasheets of the devices. [122] states that the peak power consumption of the NetFPGA-SUME device can reach as high as 184 Watts. According to [123], the Thermal Design Power (TPD), which reflects the maximum power consumption, of 4 cores CPU-based processor can reach up to 105 Watts. So we take the peak power consumption for running CPU-based P4 packet processors to be approximately equal to half this value since 2 cores are the typical minimum requirement for running P4-based software switches. Based on measurements reported in [68], the overall power consumption of a P4 ASIC-based switch, when measured at the plug, is approximately equal to 100 Watts. Finally, we assume that the power consumption of NPU-based P4 devices equals the power

Table 5.6.: Surveyed parameters corresponding to P4 devices of different types.

Device / **Param.**	**CPU**	**NPU**	**FPGA**	**ASIC**
P4 Architecture	V1M	V1M	SUME	PSA
Throughput per link (Gbps)	9	10	10	100
Power Consumption (Watts)	50	65	184	100
Max. Constructs	1000	500	250	100
Cost ($)	160	500	5000	40000
Extern (SipHash-2-4)				
Present	yes	yes	yes	yes
Throughput (Gbps)	3.3	7.6	4.2	20
Latency (μs)	90	40	0.5	0.2
Per P4 Construct Latency (μs)				
Base Processing Delay	45.9	7.45	3.54	2
Parse Header	≈ 0	0.11	0.17	0
Modify Header	0	0.5	0	0
Copy Header	0	0.28	0	0
Remove Header	-0.29	1.43	-0.02	0
Add Header	0.4	1.59	0.23	0
Add Table	0.08	0.37	0.13	0

consumption of other NPU-based devices, whose peak power consumption is reported to be equal to 65 Watts according to its data sheet [121].

The web chart in Fig. 5.3 depicts a visual representation of the comparative advantages of the different devices in terms of different metrics [104]. The chart is based on the values presented in Table 5.6 after normalizing these values between zero and one based on the maximum value for each metric.

5.2.5. Evaluation

In this subsection, we first define the evaluation scenario selected for the PA-P4VNF-RA problem, and then we analyze the collected results.

Evaluation Scenario

The goal is to find the best placement of NFs as well as the best set of P4 devices to use to satisfy a given set of workloads as input. This workload varies as multiples of the previously defined NFs in Table 5.5, ranging from 1 to approximately 200 NFs. It is assumed that the NFs workload is distinct even though they are of the same type. This

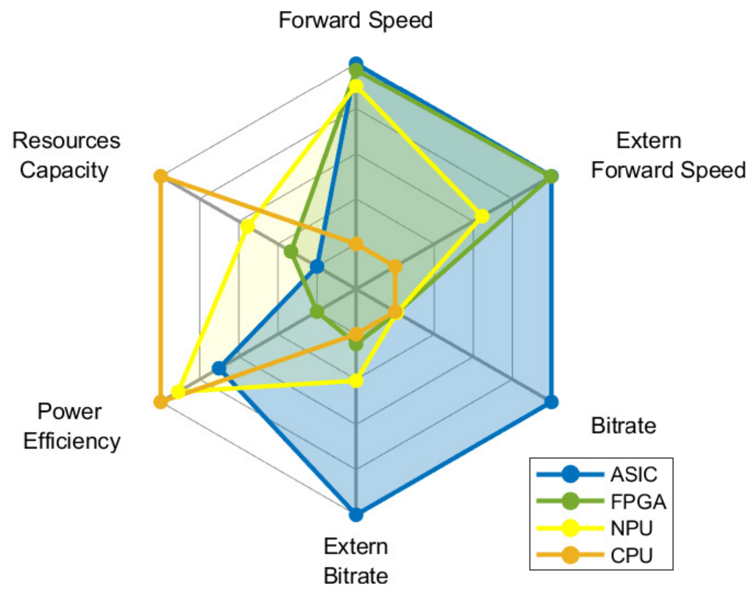


Figure 5.3.: Web chart showing the comparative advantage of different P4 device types.

assumption is taken because otherwise the processing of NFs belonging to the same type will be shared resulting in a reduced workload equal to the number of NF types. Each NF is assumed to have a 1 Gbps traffic to be handled. The widely used Gurobi solver [124] is used to solve the optimization problem.

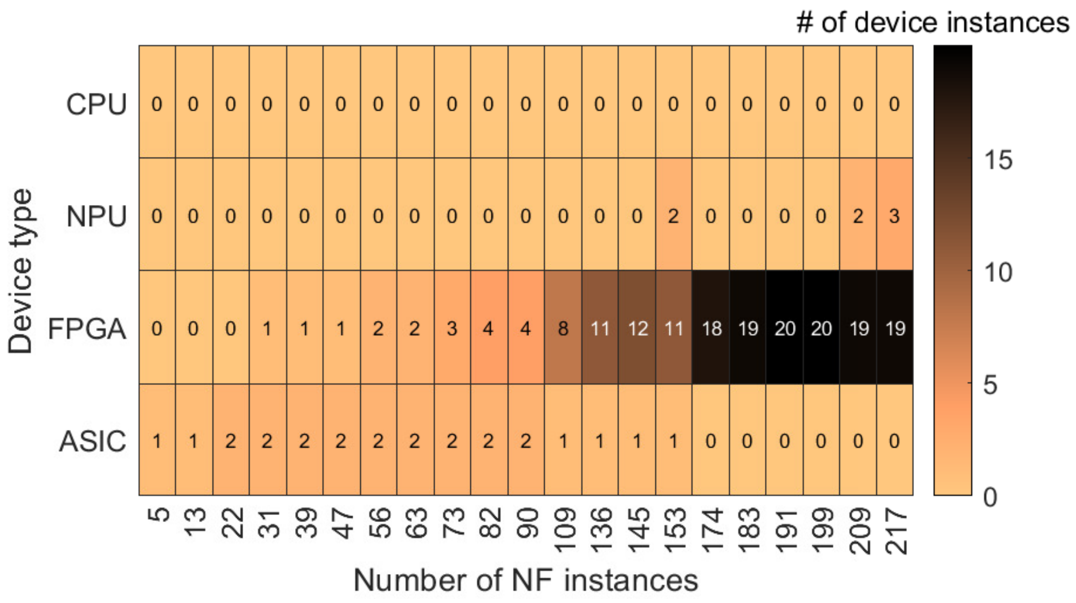
This evaluation considers four scenarios in which μ and ε in Eq. (5.5) are varied to change the relative importance of the performance and cost objectives:

1. Scenario 1: with $\mu = 1$ and $\varepsilon = 0$ so that the goal is to achieve the best performance regardless of cost.

2. Scenario 2: with $\mu = 0$ and $\varepsilon = 1$ so that the goal is to provide the cheapest solution where best-effort performance is tolerable.

3. Scenario 3: with $\mu = \varepsilon = 0.5$ so that the goal is to find a balanced solution where both performance and costs are equally weighted.

4. Scenario 4: with $\mu = 1$ and $\varepsilon = 0$ but with a predefined cost constraint of \$100k.



Device type	5	13	22	31	39	47	56	63	73	82	90	109	136	145	153	174	183	191	199	209	217
CPU	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
NPU	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	2	3
FPGA	0	0	0	1	1	1	2	2	3	4	4	8	11	12	11	18	19	20	20	19	19
ASIC	1	1	2	2	2	2	2	2	2	2	2	1	1	1	1	0	0	0	0	0	0

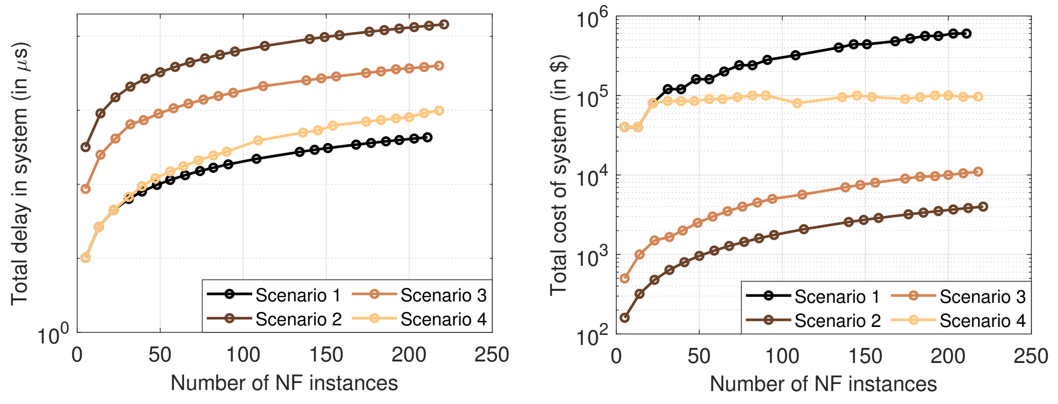
Figure 5.4.: Utilized P4 devices in Scenario 4.

Evaluation Results

The optimal solution for Scenario 1 is trivial, as all NFs are placed on ASIC-based devices, which are the most performant. The evaluation revealed that 15 ASIC devices are required in this case to place all NFs. In Scenario 2, on the other hand, the cheapest CPU-based devices are chosen to place all NFs when the goal is only to minimize costs. In this case, 25 CPU-based devices are required to place all NFs. NPU-based devices achieve the best trade-off between performance and cost in Scenario 3, where 22 NPU devices are required to place all NFs in this scenario. More interestingly are the results corresponding to Scenario 4, wherein a cost limit of $100k is specified.

The optimal placement solution for Scenario 4 is shown in Fig. 5.4, along with the number of instances of each device type required for hosting an increasing number of NFs. It can be seen that for low workload, one ASIC device is chosen because it provides the best performance while remaining affordable given the cost limit. Then, another ASIC device is used when up to 22 NFs must be placed, where the second device is required because the first device's processing resources constraint is reached.

After this point, no more ASIC devices could be used because the remaining $20k budget only allows for the second-best performing device, which is an FPGA. In this case, up to four FPGA devices are required to process the additional NFs until the total number of NFs reaches 90. Following this point, one ASIC device is sacrificed to afford more FPGA devices capable of handling the increased workload. At 174 NFs, the second ASIC device is also replaced with more FPGAs until reaching up to 20 FPGAs when the workload increases up to 199 NFs. Following this, the optimal solution sacrifices performance even further by replacing FPGA devices with the next



(a) Total forwarding delay for all scenarios. (b) Total CAPEX cost for all scenarios.

Figure 5.5.: Objective functions results for the different scenarios evaluated in the PA-P4VNF-RA problem.

best performant device, i.e., NPUs, to support processing all NFs within the available budget. Note that the limiting factor for using more FPGA and NPU devices when the workload increases is the throughput constraint.

The delay and cost functions of the optimal solution for different scenarios as a function of NFs to be placed are shown in Figs. 5.5a and 5.5b, respectively. As expected, the overall delay in Scenario 1 is the shortest, while the overall cost in Scenario 1 is the smallest. The results corresponding to Scenario 3 show the trade-off between the two objectives, where the overall delay and cost are both minimized. The results of Scenario 4 show that the system's delay is as low as that in Scenario 1 (when only the delay is optimized) until the budget constraint is reached after the point when 22 NFs need to be placed. After this point, the system's delay begins to increase in comparison to Scenario 1, while the cost stays always less than the preset budget of $100k.

5.3. PA-P4SFC-E Problem

In this section, we describe the PA-P4SFC-E problem formulation and its evaluation. This problem uses the same modeling of the P4-enhanced NFV environment infrastructure and the capabilities of the P4 packet processors as described in Subsection 5.2.1. The description of the workload in terms of SFCs requirements is illustrated in Subsection 5.3.1. The formulation of the optimization problem under study, i.e., PA-P4SFC-E, is provided in Subsection 5.3.2, where the decision variables, objective functions, and constraints are described. A performance-agnostic version of the problem is designed in Subsection 5.3.3 to serve as a baseline for assessing the importance of the performance awareness feature in the PA-P4SFC-E problem formulation. Then, a greedy algorithm for the problem is designed in Subsection 5.3.4 for solving the problem faster

Table 5.7.: Description of symbols used for modeling SFCs workload and their associated requirements.

Symbol	Workload Symbols Description
$\mathcal{S}$	Set of all the Service Function Chains (SFCs)
$\mathcal{F}^s$	Set of all the NFs constituting SFC $s \in \mathcal{S}$
$\mathcal{L}^s$	Set of all the logical links between NFs of $s \in \mathcal{S}$
f_k^s, f_{k+1}^s	Two consecutive NFs in SFC $s \in \mathcal{S}$
QoS_s^T	expected throughput that should be processed by SFC s
QoS_s^D	QoS requirement for delay limit

at runtime. Finally, a detailed evaluation of the PA-P4SFC-E problem is conducted in Subsection 5.3.5, wherein we define different experiments to evaluate: (i.) the impact of different PA-P4SFC-E problem parameters on the embedding solution; (ii.) the benefits of the performance-awareness feature adopted in the PA-P4SFC-E problem formulation when compared to the performance-agnostic baseline scenario; and (iii.) the proposed greedy solution in a scaled-up scenario.

5.3.1. Service Function Chains Requirements

A Service Function Chain (SFC) describes a processing path made up of more than one NF connected in a particular order. For example, it could be Firewall functionality followed by a Load Balancer, followed by L3Fwd. We define the set $\mathcal{S}$ to contain all the SFCs that should be supported by the network. The set $\mathcal{F}^s$ is defined to include all the NFs that build an SFC $s \in \mathcal{S}$, while the set $\mathcal{L}^s$ is defined to contain all the logical links between the NFs that constitute the SFC $s \in \mathcal{S}$. The pair f_k^s, f_{k+1}^s stands for any two consecutive NFs in SFC $s \in \mathcal{S}$.

A request to embed an SFC may be associated with certain QoS levels in terms of desired throughput or forwarding delay. The expected throughput associated with an SFC s is denoted by QoS_s^T. The expected throughput to be processed by all NFs $f \in \mathcal{F}^s$ that constitute an SFC s is equal to the required throughput to be processed by SFC S, i.e., $QoS_f^T = QoS_s^T$ for every $f \in \mathcal{F}^s$. If an SFC does not have a QoS requirement in terms of throughput, a "best-effort" QoS level is assumed with throughput equal to a predefined small value. The QoS requirement for an SFC S in terms of the upper limit on the forwarding delay is denoted by QoS_s^D. If an SFC does not have a maximum delay requirement, it is given a "best-effort" service with QoS_s^D set equal to a very large number. Table. 5.7 summarizes the description of all the symbols used for modeling SFCs workload and their associated requirements.

5.3.2. Problem Formulation

The PA-P4SFC-E problem searches for the optimal embedding of a given workload of SFCs into a given P4 infrastructure while satisfying all the functional and QoS constraints associated with each SFC. For this problem, we need the predeveloped performance models to calculate a priori the forwarding latency of different SFC embedding options. In the following, we first describe how the delay of different SFC embedding options is calculated and then we define the objective function and constraints relevant to this problem.

SFC Delay Calculation

The decision variables in this problem specify where the constituting NFs and the logical links of an SFC are embedded. The **first Boolean decision** variable in this problem, $\alpha_d^{f_k^s}$, is equal to one if the k^{th} NF of SFC s is placed on P4 device d. The **second Boolean decision variable** $\gamma_{f_k^s, f_{k+1}^s}^{d_i, d_j}$ is for mapping the logical links of an SFC to the physical links of the network. It is equal to one if logical link $(f_k^s, f_{k+1}^s) \in \mathcal{L}^s$ utilizes the physical link $(d_i, d_j) \in \mathcal{X}$. Note that, unlike the physical links, the logical links of an SFC have direction. This is why the order of parameters in the γ subscripts and superscripts matters.

The following considerations are observed when calculating the SFC delay:

1. It is possible that the NFs of one SFC get placed on the same device or on different devices across the network. Therefore, the delay calculation of an SFC should include the base delay, δ_d^{BP}, of all devices used by this SFC. This base delay accounts for the delay to access a P4 device and process the non-P4 programmable blocks inside the device. In case consecutive NFs of the same SFC are placed on the same device, then the base delay is added only once for accessing the device.

2. The average processing delay for running an arbitrary NF instance f of type y on device $d \in \mathcal{D}^P$ is denoted by Δ_d^y, and can be calculated according to Eq. (5.1).

3. Some functionalities must run on each operating device in the network. For example, it is assumed for this problem that each operating P4 device should run L2Fwd functionality, if the device hosts a part of an SFC that split over multiple devices that need to forward packets within the network, to ensure proper forwarding between connected devices in the network. So the delay to execute this required L2Fwd functionality should be added to the total delay of an SFC if it traverses from one to another device. For the sake of generality, the set $\mathcal{F}_{req}$ is already defined to include all possible required functionalities. The delay to process the required NFs in the set $\mathcal{F}_{req}$ is calculated according to the following equation:

$$\Delta_d^{req} = \sum_{f \in \mathcal{F}_{req}} \sum_{y \in \mathcal{Y}} \psi_f^y \times \Big(\sum_{c \in C_y} \sigma_y^c \delta_d^c + \sum_{e \in E | Ext_y^e = 1} \delta_d^e \Big) = \sum_{f \in \mathcal{F}_{req}} \sum_{y \in \mathcal{Y}} \psi_f^y \times \Delta_d^y. \tag{5.18}$$

4. The propagation delay over the links is neglected because we are dealing with a scenario where the network is in a cloud data center with relatively short link delays.

5. NFs constituting an SFC can be placed on separate devices. As a result, it is possible that packets need to be forwarded using devices in the network other than the hosting ones to complete their NF processing chain. While the forwarding delay of programmable devices $\mathcal{D}^P$ can be calculated as shown in Eq. 5.18, the forwarding delay over the non-programmable switches $\mathcal{D}^N$ is set to be a constant. All these forwarding delays are also added to the overall SFC delay calculation.

Taking these aspects into consideration, the total forwarding delay of a packet traversing an SFC s is calculated as shown in the following equation:

$$\Delta^s = \sum_{d \in \mathcal{D}^P} \left(\delta_d^{BP} \alpha_d^{f_0^s} + \theta_{s,d}^{in} \delta_d^{BP} + \theta_{s,d}^{out} \Delta_d^{req} + \sum_{f_k^s \in \mathcal{F}^s} \sum_{y \in \mathcal{Y}} \alpha_d^{f_k^s} \psi_{f_k^s}^y \Delta_d^y \right) + \sum_{d \in \mathcal{D}^N} \theta_{s,d}^{out} \kappa_d, \tag{5.19}$$

where θ_{s,d_i}^{in} and θ_{s,d_i}^{out} are dependent variables that count the number of logical links corresponding to SFC s that enter and leave a P4 device $d \in \mathcal{D}$, respectively. These two dependent variables are calculated as follows:

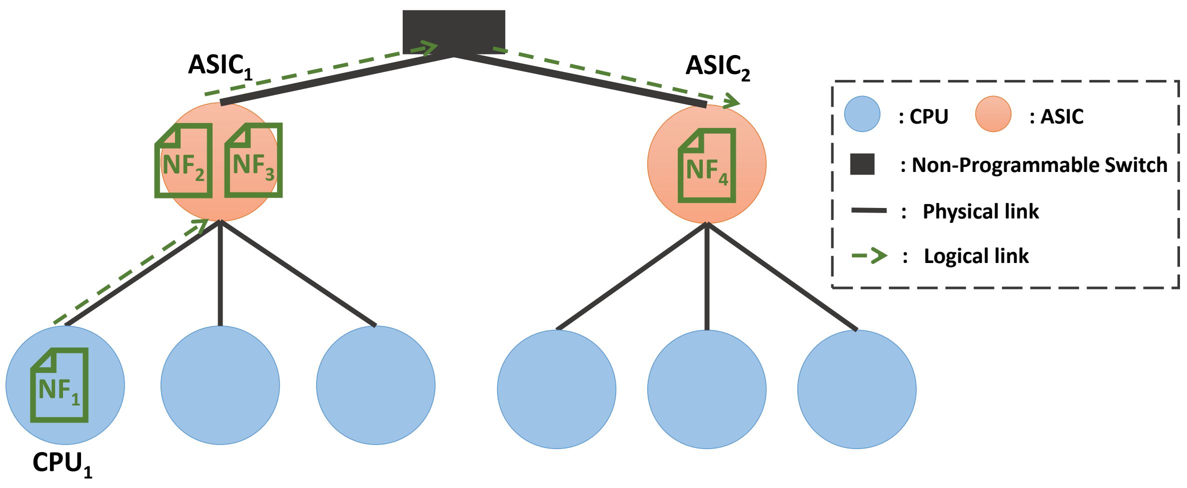
$$\theta_{s,d_i}^{in} = \sum_{(f_k^s, f_{k+1}^s) \in \mathcal{L}^s} \sum_{d_j \in \mathcal{D} | (d_i, d_j) \in \mathcal{X}} \gamma_{f_k^s, f_{k+1}^s}^{d_j, d_i}, \tag{5.20}$$

$$\theta_{s,d_i}^{out} = \sum_{(f_k^s, f_{k+1}^s) \in \mathcal{L}^s} \sum_{d_j \in \mathcal{D} | (d_i, d_j) \in \mathcal{X}} \gamma_{f_k^s, f_{k+1}^s}^{d_i, d_j}. \tag{5.21}$$

The delay components that contribute to the overall forwarding delay when traversing an SFC s shown in Eq. (5.19) are explained in the following:

- The term $\sum_{d \in \mathcal{D}^P} \delta_d^{BP} \alpha_d^{f_0^s}$ counts for the base processing delay while accessing the first hosting P4 device, while the term $\sum_{d \in \mathcal{D}^P} \theta_{s,d}^{in} \delta_d^{BP}$ counts for this base processing delay at all following P4 devices used for hosting SFC s.

- The term $\sum_{d \in \mathcal{D}^P} \theta_{s,d}^{out} \Delta_d^{req}$ is equal to the sum of the processing delays of all required NFs when leaving a P4 device. In this scenario, the required NF is selected to be L2Fwd.



$$\Delta_{\mathbf{SFC}} = \boldsymbol{\delta}^{\mathbf{BP}}_{\mathbf{CPU_1}} + \Delta^{\mathbf{NF_1}}_{\mathbf{CPU_1}} + \Delta^{\mathbf{L2Fwd}}_{\mathbf{CPU_1}} + \boldsymbol{\delta}^{\mathbf{BP}}_{\mathbf{ASIC_1}} + \Delta^{\mathbf{NF_2}}_{\mathbf{ASIC_1}} + \Delta^{\mathbf{NF_3}}_{\mathbf{ASIC_1}} + \Delta^{\mathbf{L2Fwd}}_{\mathbf{ASIC_1}} + \mathbf{k_d} + \boldsymbol{\delta}^{\mathbf{BP}}_{\mathbf{ASIC_2}} + \Delta^{\mathbf{NF_4}}_{\mathbf{ASIC_2}}$$

Figure 5.6.: Example showing the delay calculation for an SFC distributed across different P4 devices.

- The term $\sum_{d\in\mathcal{D}^P}\sum_{f_k^s\in\mathcal{F}^s}\sum_{y\in\mathcal{Y}}\alpha_d^{f_k^s}\psi_{f_k^s}^{y}\Delta_d^y$ is equal to the sum of the processing delays of the P4 programmable blocks of all the NFs in SFC s over all the used P4 devices.

- The term $\sum_{d\in\mathcal{D}^N}\theta_{s,d}^{out}k_d$ counts for the constant forwarding delay of all used non-programmable switches for forwarding packets between NFs of SFC s.

As previously stated, SFCs can share NFs. If two SFCs have the same type of NF placed on one P4 device, then the placement of this function is done only once and the two SFCs can share the usage of this NF. This is necessary to save processing resources on the hosting P4 devices by avoiding duplicate placement of NFs of the same type. Also, note that L2Fwd is assumed as a required function on devices that host a part of an SFC that split over multiple devices, to ensure proper forwarding between connected devices in the network.

For illustration purposes, we elaborate on how the delay is calculated for an SFC distributed across three different P4 devices, as shown in Fig. 5.6. The SFC is made up of 4 NFs, the network is made up of CPU and ASIC-based P4 devices, and the racks are connected with a non-programmable switch. The placement of the NFs and logical links of this SFC is spread over three different P4 devices. The delay calculation for this SFC is equal to the summation of the following components: **(i)** the base processing delay for accessing CPU_1, the delay for processing NF_1 on CPU_1, and the delay to process L2Fwd function on CPU_1 to forward packets to the second hosting device; **(ii)** the base processing delay to access $ASIC_1$, the delay for processing NF_2 and NF_3 on $ASIC_1$, and the delay to process L2Fwd function on $ASIC_1$ to forward packets to the following hosting device; **(iii)** the constant forwarding delay k_d on the non-programmable switch; **(iv)** finally, the base processing delay to access $ASIC_2$ and the delay for processing NF_4 on $ASIC_2$. There is no need to add the delay of L2Fwd at the end of the SFC because we assume that the last NF of any SFC should have L3Fwd functionality to guarantee proper routing of packets when leaving the network.

Objective Function

The objective of this problem is to minimize the operational costs when running the system. We select the total power consumption of active devices in the network, denoted by $\mathbb{P}$, as a representative of this cost, which is expressed as follows:

$$\mathbb{P} = \sum_{d \in \mathcal{D}^P} P_d \times x_d, \tag{5.22}$$

where the dependent variable x_d, defined in Eq. (5.4), indicates whether the device $d \in \mathcal{D}^P$ is used in the embedding solution.

The objective is to minimize the power consumption of active devices in the system. Accordingly, the model will favor using a combination of devices that consume less energy to reduce the total power consumption in the network.

$$Minimize\ (\mathbb{P}). \tag{5.23}$$

Eq. (5.23) depicts the objective function corresponding to the PA-P4SFC-E problem.

Constraints

The same constraints previously defined in the PA-P4VNF-RA problem for ensuring that all the NFs $\in \mathcal{F}$ that constitute the workload of SFCs are placed, architecture compatibility, availability of required externs, limited availability of processing resources and memory capacity in Eqs. (5.6), (5.9), (5.10), (5.11), (5.12), respectively still hold in the PA-P4SFC-E problem.

The following two constraints are added to ensure proper forwarding between connected devices in the network by placing the L2Fwd NF as a required function on devices that host a part of an SFC that split over multiple devices. Note that M is an arbitrarily big number that should be greater than the maximum number of logical links leaving any P4 device.

$$\alpha_d^f \leq \sum_{s \in \mathcal{S}} \theta_{s,d}^{out} \ \ \forall d \in \mathcal{D}^P \ \ \forall f \in \mathcal{F}_{req}, \tag{5.24}$$

$$\sum_{s \in \mathcal{S}} \theta_{s,d}^{out} \leq M \alpha_d^f \ \ \forall d \in \mathcal{D}^P \ \ \forall f \in \mathcal{F}_{req}. \tag{5.25}$$

The constraint presented in Eq. (5.14) for ensuring that the cumulative throughput of NFs that require offloading some functionalities to an extern function on a device never exceeds the limited throughput of that extern on the device still holds. In this problem, we assume that each programmable device in the network has one ingress and one egress port to serve as an ingress or egress node for the traffic corresponding to any SFC. Hence, the cumulative traffic rate of all SFCs entering and leaving each programmable device on the ingress and egress ports, respectively should never exceed the capacity of these ports. This is achieved by the following two constraints:

$$\sum_{s \in \mathcal{S}} QoS_s^T \times \alpha_d^{f_0^s} \leq T_d \, \forall d \in \mathcal{D}^P, \tag{5.26}$$

$$\sum_{s \in \mathcal{S}} QoS_s^T \times \alpha_d^{f_{end}^s} \leq T_d \, \forall d \in \mathcal{D}^P, \tag{5.27}$$

where $\alpha_d^{f_0^s}$ and $\alpha_d^{f_{end}^s}$, respectively indicate whether the first (ingress) NF of SFC s and the last (egress) NF of s were placed on device d.

To make sure that all the logical links of all SFCs are mapped to physical links while respecting flow conservation, the following constraint is needed:

$$\sum_{d_j \in \mathcal{D} | (d_i, d_j) \in \mathcal{X}} (\gamma_{f_k^s, f_{k+1}^s}^{d_i, d_j} - \gamma_{f_k^s, f_{k+1}^s}^{d_j, d_i}) = (\alpha_{d_i}^{f_k^s} - \alpha_{d_i}^{f_{k+1}^s}) \ \forall s \in \mathcal{S} \ \ \forall (f_k^s, f_{k+1}^s) \in \mathcal{L}^s \ \forall d_i \in \mathcal{D}. \tag{5.28}$$

This constraint ensures that for any logical link between 2 consecutive NFs (f_k^s, f_{k+1}^s) of an SFC s, and for each device d_i, if the NF f_{k+1}^s of SFC s is not placed on d_i as its predecessor f_k^s, then there should be a neighboring device d_j such that the logical link between f_k^s and f_{k+1}^s uses its physical link with d_i. This equation also ensures that the sum of all incoming flows and outgoing flows of logical links for each device adds up to 0, i.e., the net flow is equal to 0 on all devices.

The following constraint is needed to ensure that a logical link can not be mapped to the same physical link in both directions.

$$\gamma_{f_k^s, f_{k+1}^s}^{d_i, d_j} + \gamma_{f_k^s, f_{k+1}^s}^{d_j, d_i} \leq 1 \, \forall s \in \mathcal{S} \ \ \forall (f_k^s, f_{k+1}^s) \in \mathcal{L}^s \ \forall d_i, d_j \in \mathcal{D} \mid (d_i, d_j) \in \mathcal{X}. \tag{5.29}$$

The capacity of physical links between devices should never be exceeded. This is ensured with the following constraint:

Table 5.8.: Description of variables and objectives used to formulate the PA-P4SFC-E problem.

Symbol	**Symbols Description for PA-P4SFC-E problem**
Decision Variables	
$\alpha_d^{f_k^s}$	Boolean variable equals 1 if the k^{th} NF f of SFC s is placed on P4 device $d \in \mathcal{D}^P$
$\gamma_{f_k^s, f_{k+1}^s}^{d_i, d_j}$	Boolean variable equals 1 if logical link (f_k^s, f_{k+1}^s) uses the physical link (d_i, d_j)
Variables	
x_d	Boolean variable equals 1 if any NF instance is placed on P4 device $d \in \mathcal{D}^P$
π_d^y	Boolean variable equals 1 if any NF instance of type y is placed on P4 device $d \in \mathcal{D}^P$
$\theta_{s,d}^{in}$	Integer variable equals the number of logical links in SFC s that enter a device $d \in \mathcal{D}$
$\theta_{s,d}^{out}$	Integer variable equals the number of logical links in SFC s that leave a device $d \in \mathcal{D}$
Δ_d^y	Delay needed to process programmable logic (P4 blocks and externs) when running NF of type y on P4 device d
Δ^s	Average forwarding delay when traversing SFC s
Objective	
$\mathbb{P}$	Total power consumed by active devices in the network

$$\sum_{s \in \mathcal{S}} \sum_{(f_k^s, f_{k+1}^s) \in \mathcal{L}^s} QoS_s^T \times \gamma_{f_k^s, f_{k+1}^s}^{d_i, d_j} \leq T(d_i, d_j) \; \forall (d_i, d_j) \in \mathcal{X}. \tag{5.30}$$

Finally, to support the QoS requirements in terms of forwarding delay associated with SFC embedding requests, the following constraint should be satisfied for all SFCs:

$$\Delta^s \leq QoS_s^D \quad \forall s \in \mathcal{S}. \tag{5.31}$$

If an SFC does not have a delay QoS requirement (i.e., best-effort service), then QoS_s^D is set to a very large number.

Table 5.8 summarizes the description of all the symbols used for the formulation of the PA-P4SFC-E problem.

5.3.3. Performance-Agnostic Baseline Scenario

To evaluate the effectiveness of integrating performance awareness into the proposed PA-P4SFC-E problem, we implement a performance-agnostic baseline scenario of the problem. The only difference between the PA-P4SFC-E problem and the baseline scenario is that we remove the predeveloped performance models used for the a priori

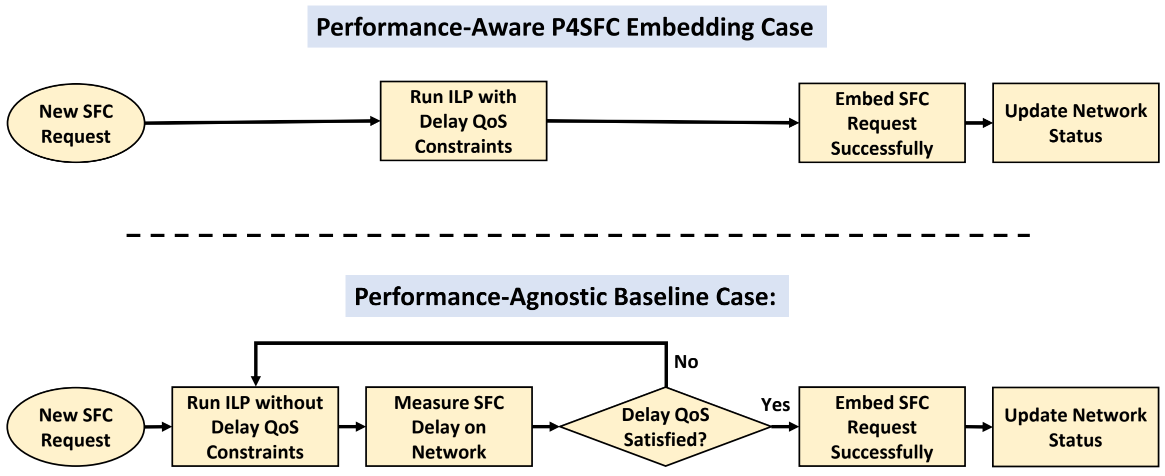

Performance-Aware P4SFC Embedding Case
New SFC Request
Run ILP with Delay QoS Constraints
Embed SFC Request Successfully
Update Network Status
Performance-Agnostic Baseline Case:
New SFC Request
Run ILP without Delay QoS Constraints
Measure SFC Delay on Network
Delay QoS Satisfied?
No
Yes
Embed SFC Request Successfully
Update Network Status

Figure 5.7.: Workflow in PA-P4SFC-E approach versus the performance-agnostic baseline approach.

SFC delay calculation as shown in Fig. 5.7. In this performance-agnostic baseline version, the solution will be found using the same Integer Linear Programming (ILP) formulation presented for the performance-aware scenario, except that the constraints related to the delay QoS are removed because this QoS information is missing in this scenario. Instead, the delay resulting from the embedding solution will be derived from measuring it on the real system. In case the required QoS level is not met, a new solution should be found after excluding the previously examined placement solution (combination of P4 device types) until satisfying the required QoS level. After embedding the SFC successfully, the network status is updated by subtracting the consumed processing resources on the hosting devices and the utilized rates from link capacities.

5.3.4. Greedy Algorithm for PA-P4SFC-E Problem

The SFC embedding problem is known to be a NP-hard problem. Therefore, it is important to consider a greedy approach that helps in solving the problem in a short time when the network is large even if this came at the cost of reduced optimality. In this direction, we propose and implement a greedy algorithm as illustrated in Fig. 5.8.

The algorithm restricts the placement of an SFC to take place only on a single device, i.e., no SFC splitting on more than one device is allowed. This assumption is taken based on preliminary evaluation, wherein we found that the optimal solution rarely results in splitting the placement of SFCs into more than one device because of the performance penalty (additional base processing delay) incurred when accessing new devices. On the other hand, this assumption has a big impact on simplifying the solution because it saves searching for the optimal logical link placement. In this

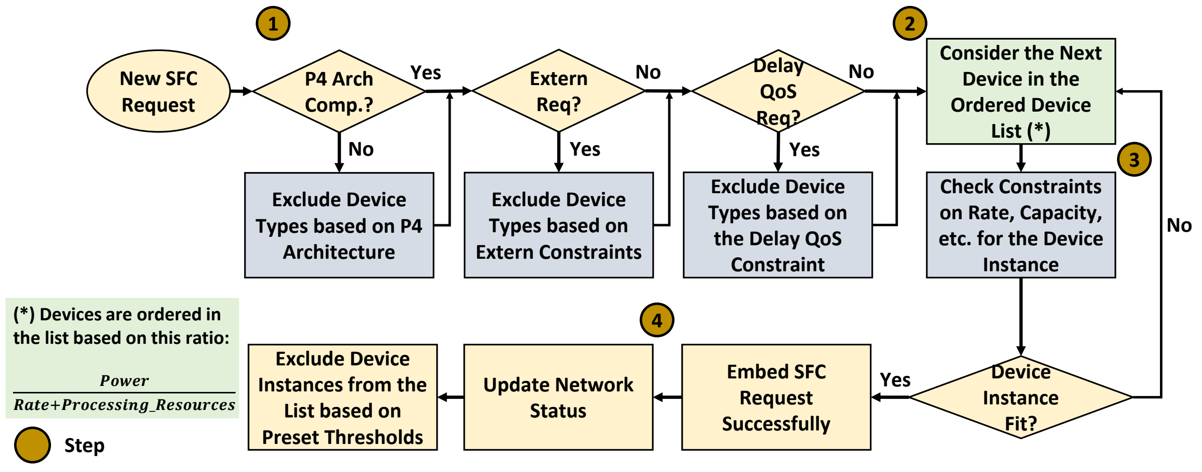

1
New SFC Request
P4 Arch Comp.?
Yes
No
Exclude Device Types based on P4 Architecture
Extern Req?
No
Yes
Exclude Device Types based on Extern Constraints
Delay QoS Req?
No
Yes
Exclude Device Types based on the Delay QoS Constraint
2
Consider the Next Device in the Ordered Device List (*)
3
Check Constraints on Rate, Capacity, etc. for the Device Instance
No
Device Instance Fit?
Yes
Embed SFC Request Successfully
4
Update Network Status
Exclude Device Instances from the List based on Preset Thresholds
(*) Devices are ordered in the list based on this ratio:
Power / Rate+Processing_Resources
Step

Figure 5.8.: Proposed greedy algorithm for the PA-P4SFC-E problem.

case, the link capacity throughput constraint is discarded and only the throughput constraints on the externs, ingress port, and egress port are applicable.

When a new SFC request arrives, the algorithm searches for the best placement solution that reduces power consumption in the system while satisfying all the applied constraints. In the ILP problem, this solution is found through the Branch and Bound Algorithm. In our proposed algorithm, the solution is found by applying the following steps as shown in Fig. 5.8:

1. Excluding Device Types: The algorithm checks first the constraints to exclude incompatible device types, such as CPU, NPU, FPGA, and ASIC, for hosting the given SFC. Then, it excludes all device instances belonging to a certain type in case this type does not satisfy all the constraints. These device type-related constraints include the P4 architecture compatibility, Extern requirements, and delay QoS requirement. Note that the last constraint depends on SFC delay calculation, which is the same for a given device type.

2. Searching Reduced Space: After shortlisting the candidate hosting P4 devices, the algorithm searches the reduced solution space by visiting an ordered list of devices which is created to guide the search, instead of a random search, to reduce the execution time of the algorithm. The devices in the list are ordered in an ascending order based on a newly defined ratio equal to:

$$\frac{norm(Power)}{norm(Rate) + norm(Processing\_Resources)} \tag{5.32}$$

This metric represents the power efficiency of a device per available resources. Thus, if the power consumption is smaller, or the device supports higher throughput capacity and has more processing resources, this ratio will be smaller and thus the device is more favored. This way, the SFC embedding solution is guided to minimize the power consumption in the system while recognizing the resources available on the devices and their throughput capacities. Note that this ratio is calculated for each device type after normalizing the constituting metrics (power, rate, and processing resources) between 0 and 1. This normalization is done by subtracting the minimum value recorded for that metric across different device types and dividing by the range of the values (i.e. maximum value - minimum value of the normalized metric among the device types). In the considered evaluation scenario, the device types are ordered according to the proposed ratio as follows: CPU, NPU, ASIC, FPGA, meaning that the placement algorithm will first favor checking the CPU devices, which have the lowest power consumption, as long as the other constraints, including delay QoS constraint, are satisfied. On the contrary, FPGA devices, with the highest power consumption and relatively fewer processing resources, will be checked as a last resort.

3. Checking Constraints: The next step is to check if the selected P4 device instance satisfies the remaining constraints which are applied for each device instance. These constraints include the throughput capacity of the devices as well as the processing/compute resources capacity. If all constraints are satisfied, then the embedding is successfully applied, otherwise, the next device in the ordered list from the previous step is examined.

4. Update Network Status & Excluding Device Instances: In this last step, the network status is updated by subtracting the resources consumed by the current SFC request from the hosting device. The ordered list is updated by excluding the device instances whose remaining processing resources or throughput capacity is smaller than preset thresholds, without changing the order of the devices. These preset thresholds are input parameters to the algorithm, whose values depend on the granularity of the SFC's requests in terms of processing resources and throughput. In other words, these thresholds are set equal to the expected smallest throughput required in the system and the expected smallest SFC size (in terms of the total number of constituting P4 constructs) to be embedded in the system.

5.3.5. Evaluation

In this subsection, we define different experiments for evaluating the PA-P4SFC-E problem. The problem is formulated and implemented as an ILP problem. The commonly used Gurobi solver [124] is used to solve the ILP problem. Note that the selection of the system's parameters related to NFs and P4 devices is the same as that described in Subsection 5.2.4.

The model's complexity grows in direct proportion to the size of the network and the number of SFCs to be placed. The first three experiments were designed to understand the impact of different factors related to the SFC workload and network topology on the performance of the system. The evaluation of these three experiments is conducted on small networks as the purpose is only to reveal the impact of these factors. In these experiments, the entire workload of SFCs must be placed at the same time to achieve the optimal placement based on our model. In other words, it is assumed that no functions are running on the network before the placement optimization. If any subset of the SFCs to be placed on the network cannot be placed, the entire placement of the SFC workload is considered infeasible. The SFCs used in these experiments are chains made up of the NFs defined in Table 5.5. The workload to be placed is made up of a repeated set of SFCs, where the set of SFCs corresponding to different experiments is selected differently as will be elaborated in the experiments' description. The NFs constituting the SFCs are assumed to be distinct even though they belong to the same type because otherwise, NF sharing will take place reducing the overall workload to be placed into the network. The purpose of each of the first three experiments is defined in the following:

- **Experiment 1**: In this experiment, the characteristics of the SFC workload are analyzed. For this purpose, we vary both the length of the SFCs as well as the delay QoS required by these SFCs.
- **Experiment 2**: In this experiment, the effect of replacing traditional ToR switches with programmable ASIC switches is investigated.
- **Experiment 3**: In this experiment, we target understanding the impact of equipping the servers in the network with Infrastructure Processing Unit (IPU)s (i.e., NPU and FPGA-based SmartNICs) on the performance of the network in terms of the embedding capacity as well as the power consumption savings.

In **Experiments 4** and **5**, the model is required to embed SFCs into a network already populated with other running SFCs. In other words, the problem takes into consideration the network state when it handles SFC embedding requests at runtime. The purpose of the 4^{th} experiment is to highlight the gains achieved by adopting performance awareness in the formulation of the problem compared to a performance-agnostic case that serves as a baseline scenario. In the 5^{th} experiment, the proposed greedy solution is evaluated to evaluate its execution time and the optimality gap compared to the ILP-based solution, when dealing with a scaled-up scenario where bigger networks are considered.

Experiment 1- Examining Impact of SFC Characteristics

The goal of this experiment is to examine the impact of SFC characteristics on the embedding solution. For this purpose, we vary the length of the SFCs to be made up of 2, 4, and 6 connected NFs. Moreover, we consider 3 delay QoS levels for any SFC: 10 μs, 100 μs, and Best-Effort (no delay QoS requirement). To ensure a fair comparison when

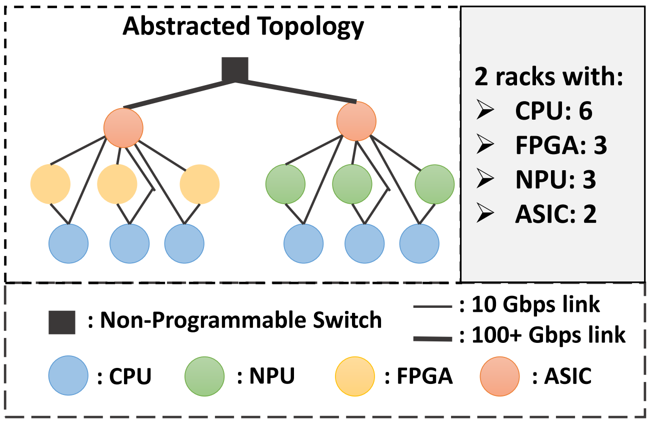
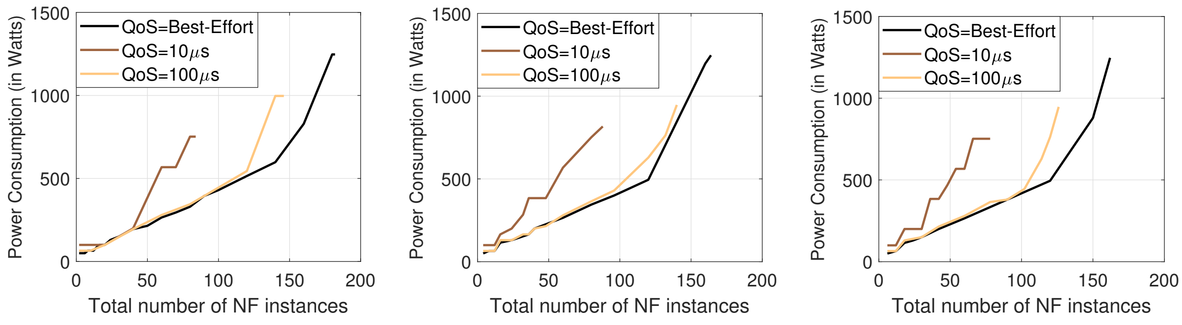


Figure 5.9.: Network setup used in the first experiment where SFC characteristics are varied.

examining the impact of SFC length on the performance of the system, we compare the three cases with different SFC lengths based on the total number of NFs to be embedded. Moreover, the throughput QoS for each SFC is selected in a way to ensure that the overall traffic coming to the system is identical for the different SFC length cases. So the throughput QoS for SFCs of length 6 is x1.5 that of SFCs of length 4, and x3 that of SFCs of length 2. In our case, we select the QoS throughput for SFCs of lengths 2, 4, and 6 to be equal to 1, 2, and 3 Gbps, respectively. This way, a request to embed a workload of one SFC of length 6 with 6 NFs to be processed and a total throughput of 3 Gbps is comparable to a request to embed a workload of x1.5 SFCs of length 4 each requiring 2 Gbps, or a workload of x3 SFCs of length 2 each requiring 1 Gbps. In these three cases, the workload in terms of the total number of NFs to be processed and the total incoming traffic rate is the same, and the only difference is in the chaining complexity between these NFs based on the length of the SFCs. The SFCs to be embedded are all made up of the NFs described in Table 5.5, and they all include LB NF that require SipHash-2-4 extern functionality. The evaluation is conducted assuming an infrastructure made up of different types of P4 devices as shown in Fig. 5.9.

The results corresponding to this experiment are shown in Fig. 5.10. The plots in this figure show the value of the objective function in terms of the total power consumption in the system in Watts (W) as a function of the total number of NF instances embedded into the system for different delay QoS cases. The different subplots 5.10a, 5.10b, 5.10c correspond to the cases where the length of the SFCs is set equal to 2, 4, and 6, respectively. Note that we plot on the x-axis the total number of NF instances constituting these SFCs to have a normalized way of presenting the processing workload of the different cases. In this evaluation, we gradually increase the number of SFCs to be embedded up to a point where no feasible solution could be found, i.e., the processing capacity of the network is reached.



(a) SFC of length 2. (b) SFC of length 4. (c) SFC of length 6.

Figure 5.10.: Results corresponding to the optimal placement solution in experiment 1 where delay QoS and SFC length are varied.

For all SFC length cases, we can observe that when the delay QoS is more stringent, the total number of NFs (or SFCs) that are successfully embedded decreases. For example, looking at the case when the SFC length is equal to 6 in Fig. 5.10c, we can observe that the total number of NFs successfully embedded is equal to 78 NFs (= 13 SFCs) when delay QoS is equal to 10 μs, which is less than the case when delay QoS is set to 100 μs where 126 NFs (= 21 SFCs) could be embedded, which is less than the Best-Effort delay QoS case where 162 NFs (=27 SFCs) could be embedded. The reason for this is that when the required delay QoS is more stringent, the P4 devices with low performance can not fulfill these requirements and thereby they get shortlisted. For example, we can see from Table 5.6 that the base processing delay to access the CPU-based devices is approximately equal to 46 μs, accordingly, this device is automatically excluded from being an option to host SFCs with 10 μs delay QoS requirement.

We can also observe from these figures that for a given NF workload, the total power consumption in the system increases when the delay QoS requirement is more stringent. For example, looking at Fig. 5.10c, when 60 NFs (=10 SFCs) are to be embedded, the power consumption yielding from the optimal solution is equal to 280 W when delay QoS is equal to 100 μs, which is less than 568 W power consumption needed for hosting this workload when the delay QoS is equal to 10 μs. The reason for this is that the more performant P4 devices needed to suffice the embedding of SFCs with more stringent delay QoS requirements consume more power as can be inspected from Table 5.6.

When comparing the three subplots in Fig. 5.10, we can observe that the impact of the length of SFCs on the optimal placement solution is minimal. Recalling that the incoming traffic rate per NF workload for all SFC lengths are normalized, we can conclude that the complexity of the SFCs in terms of the number of NFs chained does not affect the optimal management and performance of the system. Note that when the NF workload is high, we can observe a minor difference in the total number of NFs that can be placed into the network between the different SFC length cases. This is due to the difference in the granularity of the SFCs to be placed in terms of the number

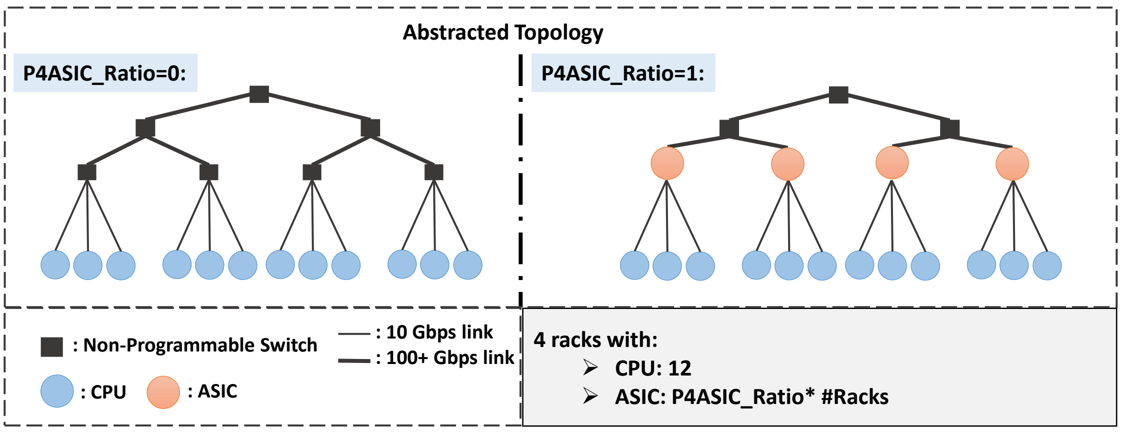


Figure 5.11.: Network setup used in the second experiment where the ratio of programmable switches adoption is varied.

of constituting NFs and the throughput requirements. In other words, just before the saturation of the processing capacity of the P4 devices, it is possible to fit a few short SFCs with low throughput requirements into the remaining processing capacity of the system before saturation. This is not possible for longer SFCs with higher throughput requirements. Note that in most of the cases in this experiment, the capacity of the devices is reached because of the extern throughput constraint, i.e, the throughput capacity for running external functions.

Experiment 2- Examining Impact of Programmable ASICs Adoption

The purpose of the second experiment is to study the impact of replacing traditional ToR switches with P4 programmable ASICs. These P4 ASIC devices can execute NFs beside the typical forwarding. The experimental scenario is shown in Fig. 5.11. The network is made up of CPU-based processors. The ToR switches can be replaced with P4 programmable ASIC switches according to the ratio "P4ASIC_Ratio", denoted by r, where a value of zero means that no P4 ASIC switches are used, while a value of one means that all ToR switches are replaced with P4 ASIC-based devices. Five networks are emulated where r is varied taking the following values: 0, 0.25, 0.5, 0.75, and 1. Recall from Table 5.6 and Fig. 5.3 that the P4 ASIC devices possess higher performance in terms of throughput and forwarding speed but lower power efficiency and processing resources compared to CPU-based devices. SFCs of length 4 are selected as the workload to be embedded in this experiment, where the constituting NFs are based on those defined in Table 5.5 such that all the chains include LB NF that require extern function. The delay QoS for the SFC workload is set to Best-Effort to make sure that CPU devices can suffice the delay requirements and thus can be used as potential hosts for the SFCs. The throughput QoS for each SFC is set to 1 Gbps.

Fig. 5.12 shows the results corresponding to the optimal solution in experiment 2 in

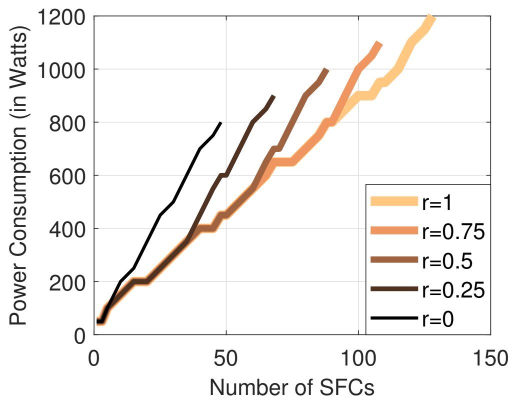


Figure 5.12.: Results corresponding to the optimal placement solution in experiment 2 where the degree of adoption of programmable ASIC switches is varied.

terms of the system's power consumption as a function of an increasing SFC workload. Different plots correspond to different cases where the degree of adoption of P4 ASICs is varied between zero and one. The results corresponding to the different cases when P4 ASICs are used (i.e., r>0) resemble a similar pattern where the curves overlap up to a certain workload before they increase sharply. In the overlapping stage, the model makes use of the available P4 ASIC-based devices until they are fully occupied before falling back to the CPU-based devices. The reason for this is that the ASIC-based devices can handle higher throughput (x11 normal traffic and x6 traffic requiring extern function) compared to CPU-based devices as can be inspected from Table 5.6. This makes the power efficiency of these ASIC devices higher compared to CPU-based servers since one ASIC device, even though consumes more power, can handle a larger workload that may require multiple CPU-based devices to suffice. After the overlapping stage, the slopes of the curves corresponding to the cases when the ratio of P4 ASIC adoption is equal to 0.25, 0.5, 0.75, and 1 suddenly change after handling around 30, 60, 90, and 120 SFCs, respectively. At these points, the P4 ASIC devices are fully occupied, and only the other available type of devices, i.e., CPU-based, is used to handle the remaining SFC workload. It is observed that the slope of these curves after the sudden change is equal to that in the case when r=0 where only CPU-based devices are available to host the SFC workload. When the throughput constraint on the CPU-based devices is reached, no more SFC embedding requests can be supported. Note that it is expected that the same patterns will get repeated when evaluating bigger networks with more racks and devices per rack.

In general, we can observe that using more programmable ASICs as ToR switches increases the embedding capacity of the system. Moreover, for the same SFC workload, using more programmable ASIC devices reduces the operational cost of the system presented as the power consumption. The selection of ASIC devices takes place, even though the power consumption of a single programmable ASIC device is higher than the power consumption of alternative CPU-based devices because a single ASIC device has a higher throughput capacity for accommodating a bigger number of SFCs,

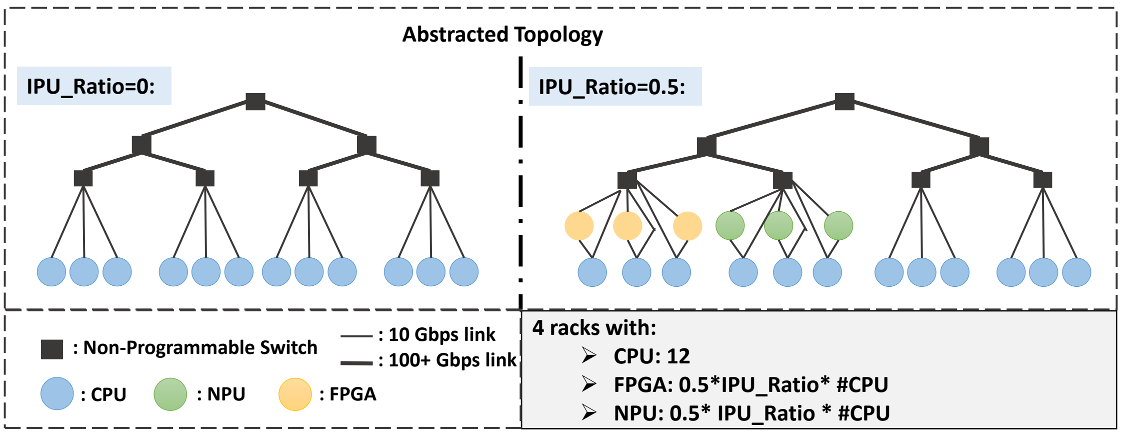


Figure 5.13.: Network setup used in the third experiment where IPUs adoption ratio is varied.

especially when extern function for LB NF is needed. However, it should be recalled that the performance gains when adopting ASIC devices come with a higher CAPEX cost as the programmable ASIC devices may be more expensive than traditional ToR switches.

Experiment 3- Examining Impact of IPUs Adoption

The purpose of this experiment is to study the impact of adopting more hardware accelerators in the cloud infrastructure on the performance of the system. These accelerators or IPUs can be NPU or FPGA-based SmartNICs attached to the servers in the network. The experimental scenario is shown in Fig. 5.13. The network is made up of CPU-based servers. These servers can be equipped with programmable IPUs according to the parameter "IPU_Ratio", denoted by r. Three networks are emulated where r is varied taking the following values: 0, 0.5, and 1. A value of zero means that no IPUs are used, while a value of one means that all servers are upgraded with IPUs. A ratio of 0.5 means that half of the racks are equipped with IPUs (a quarter with FPGAs and the other quarter with NPUs). Recall from Table 5.6 and Fig. 5.3 that the NPU and FPGA-based IPUs possess higher performance in terms of throughput and forwarding speed but lower power efficiency and processing resources compared to CPU-based devices. SFCs of length 4 are selected as the workload to be embedded in this experiment, where the constituting NFs are based on those defined in Table 5.5 such that all the chains include LB NF that require extern function. The delay QoS for the SFC workload is set to Best-Effort to make sure that CPU-based devices can suffice the delay requirements and thus can be used as potential hosts for the SFCs. The throughput QoS is set to 2 Gbps for all the SFC requests.

The results corresponding to this experiment are shown in Fig. 5.14. The power consumption in the system is plotted as a function of an increasing SFC workload to be

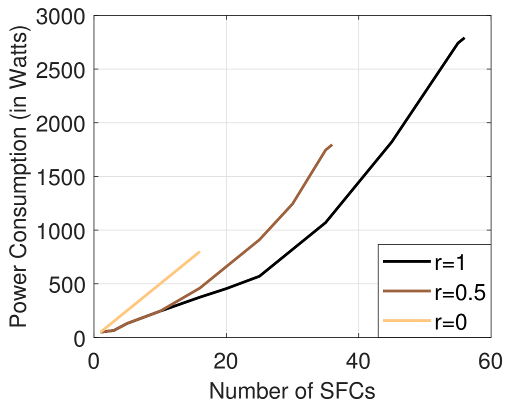


Figure 5.14.: Results corresponding to the optimal placement solution in experiment 3 where the degree of adoption of IPU devices is varied.

embedded for different IPU adoption degrees. We can clearly observe that upgrading the network with IPUs increases the capacity for handling more SFC requests. This increase reaches 125% when half of the racks are equipped with IPUs and 250% when all the racks are upgraded with IPUs. This is mainly because these IPUs can handle additional traffic rates especially when inline processing takes place there. Moreover, we can observe that for the same SFC workload such as at 20 SFCs, the power consumption in the system decreases when using more IPUs even though the per IPU power consumption is higher compared to CPU-based servers. The reason for this is that the model efficiently utilizes the higher throughput capacity of IPUs in handling extern functions when hosting the LB NF. This results in using fewer devices and thus higher power efficiency in the system. Similar to the previous experiment, the performance gains in the system when using IPUs as a function of adoption ratio r follows an identifiable pattern, which permits extrapolating the impact of these IPUs on more scaled-up scenarios with more racks and devices per rack. Again, this comes at a higher CAPEX cost for purchasing these IPUs.

Experiment 4- Examining the Impact of Performance Awareness

The goal of this experiment is to evaluate the effectiveness of integrating performance awareness in the proposed PA-P4SFC-E problem. For this purpose, we evaluate the performance-agnostic baseline scenario explained in Subsection 5.3.3, wherein the same PA-P4SFC-E ILP problem is formulated except that we remove the predeveloped performance models used for the a priori SFC delay calculation. In this evaluation, the proposed PA-P4SFC-E model and the baseline model will host newly coming SFC requests one at a time without altering the previously embedded SFCs in the network.

The evaluation is conducted on a network made up of 4 racks each with 40 CPUs and 40 IPUs. The CPUs and IPUs per rack are connected with programmable ASIC ToR

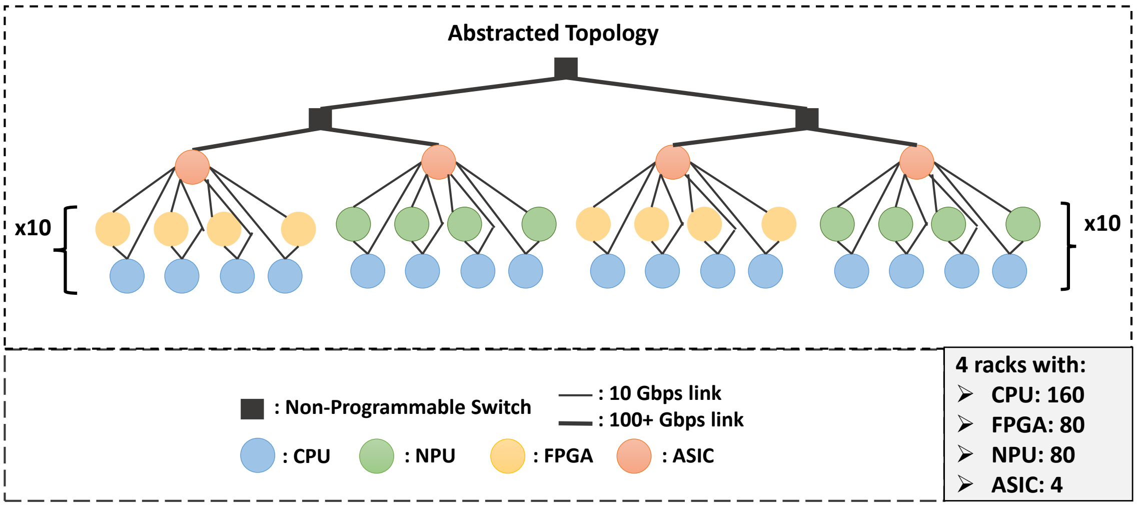


Figure 5.15.: Network setup used in the fourth experiment for evaluating the impact of performance awareness.

switches as shown in Fig. 5.15. In total, the programmable processing substrate is made up of 160 CPUs, 80 NPUs, 80 FPGAs, and 4 programmable ASICs.

Three realistic SFCs taken from the literature are used as the workload in this experiment. we always assume that L3Fwd NF is available at the end of the SFCs to assure proper packet routing at the end of the processing chain. The adopted SFCs along with their delay and throughput QoS requirements are listed in the following:

- Firewall → NAT → L3Fwd [70] with 10 μs delay QoS and 2 Gbps throughput QoS.
- Load Balancer → L3Fwd [71] with 100 μs delay QoS and 1 Gbps throughput QoS.
- Firewall → Load Balancer → L3Fwd [72] with Best-Effort delay QoS and 3 Gbps throughput QoS.

The power consumption in the system, in Watts, for both PA-P4SFC-E and baseline approaches as a function of an increasing total number of NFs to be placed is shown in Fig. 5.16. Note that we show on the x-axis the total number of NFs instead of the total number of SFCs because the considered SFCs to be embedded in this experiment have different lengths. The embedding in this experiment is done at runtime, where the previous placements and network status are preserved when a new SFC embedding request is served. We can observe that the power consumption in the system is the same in both the PA-P4SFC-E and baseline approaches, meaning that they are both yielding almost the same optimal solution. This is expected because the performance-agnostic baseline solution will keep searching until it finds a solution that satisfies all the QoS requirements. The cost of this search is reflected in the execution time.

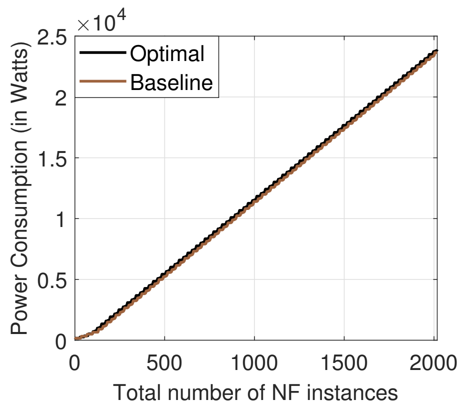


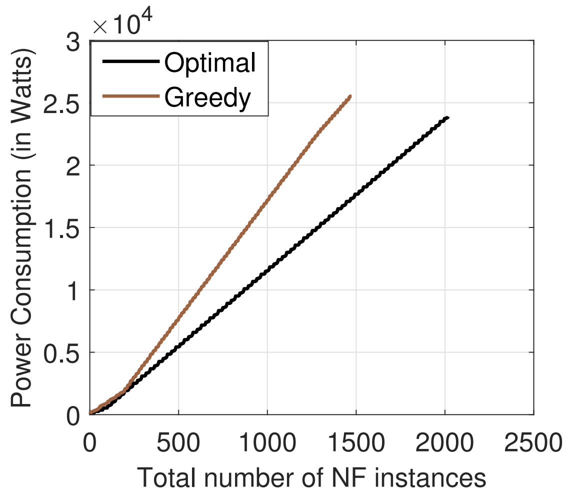
Figure 5.16.: Results corresponding to the optimal and baseline solutions in experiment 4 where the impact of performance-awareness is studied.

We found that the execution time to find a solution in the baseline scenario is around 2.6 times that needed to find a solution using our proposed PA-P4SFC-E problem. This extra time is due to the extra trials needed by the performance-agnostic baseline approach to finding a solution that satisfies all QoS requirements. We found that on average, the baseline solution needs additional 1.68 extra trials until it embeds an SFC request successfully compared to the solution found by the performance-aware problem formulation. These extra trials are due to the failure of the baseline solution in satisfying the QoS requirements because it is performance-agnostic. Moreover, it should be noted that for each trial in the baseline approach, the time and cost for evaluating the delay of the tried/unsuccessful solution should be added, which makes this approach almost non-practical for real-life deployments that require fast solutions.

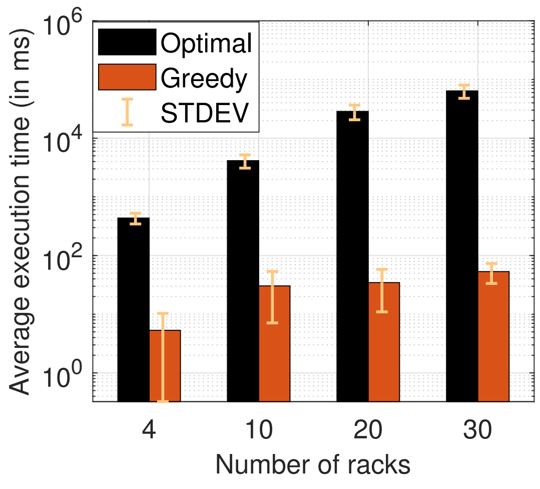
Experiment 5- Evaluating the Greedy Solution

The purpose of this experiment is to evaluate the effectiveness of the proposed greedy solution in Subsection 5.3.4 compared to the original ILP problem's solution. The trade-off between the execution time and the optimality gap is analyzed. For this evaluation, we use a realistic SFC workload similar to that used in the previous (4^{th}) experiment. We evaluate 4 different topologies of increasing size to investigate the impact of scalability on the execution time of the algorithm. The first used topology is similar to the network used in the 4^{th} experiment shown in Fig. 5.15, where it is made up of 4 racks, each with 40 CPUs, 40 IPUs (40 FPGAs or 40 NPUs), and connected with ASIC devices. The other evaluated networks are the same but with an increased number of racks made up of 10, 20, and 30 racks, respectively.

The results of this experiment are shown in Fig. 5.17. The power consumption in the system, when made up of 4 racks, in Watts, for both optimal and greedy solutions as a function of an increasing total number of NFs to be placed, is shown in Fig. 5.17a. Note that we show on the x-axis the total number of NFs instead of the total number of SFCs



(a) Power consumption in the 4 rack network.



(b) Execution time to find a solution.

Figure 5.17.: Results corresponding to the optimal and greedy solutions in experiment 5 where scaled-up scenarios are tested.

because the considered SFCs to be embedded in this experiment have different lengths. The embedding in this experiment is done at runtime, where the previous placements and network status are preserved when a new SFC embedding request is served. For this reason, we can see that the cumulative difference in the total power consumption in the system, compared to the optimal case, increases when the number of SFCs to be embedded increases, as the per SFC embedding optimality gap accumulates. The total number of NFs that could be embedded into the system is equal to around 1480 when the greedy solution is applied, approximately 26.6 % less than that when the ILP solution is applied, where the total number of NFs that could be embedded is equal to 2018. At the maximum number of NFs that could be embedded using the greedy solution, the optimality gap in terms of power consumption reaches around 8000 Watts.

Fig. 5.17b shows, in a logarithmic scale, the average execution time (in ms) needed for embedding an SFC request when the greedy and optimal solutions are applied as the number of racks in the network substrate increases. The standard deviation of the different recorded execution times is recorded and plotted on top of the averages. While the execution time to solve the ILP problem increases largely when the network size increases, our proposed greedy solves scaled-up scenarios faster. The optimal solution requires 435, 4137, 28500, and 64000 ms to be reached when the number of racks is equal to 4, 10, 20, and 30 racks, respectively, while the greedy solution only requires 5, 30, 35, and 53 ms to solve the problem in these cases. When looking at execution time results, it is clear that such a greedy solution is important, especially when considering large networks where tens of seconds are needed to find an embedding solution at runtime using the ILP formulation.

5.4. Summary

In this chapter, we utilize the performance models derived in Chapter 4 to formulate and evaluate two optimization problems related to managing P4-enhanced NFV environments.

The first problem, named PA-P4VNF-RA, targets finding the optimal planning of the infrastructure substrate of P4-enhanced NFV environments. The optimization problem looks for the optimal set of P4 packet processors that can handle a given processing workload and the placement solution of this workload into the selected hosting devices. The multi-objective function targets maximizing the performance in the system while minimizing the capital expenditure costs when selecting the optimal set of P4 packet processors that can handle a given processing workload. The problem constraints the solution space to satisfy the requirements of the NF workload, while recognizing the distinct capabilities of the different candidates hosting P4 devices. The predeveloped performance models and evaluation in Chapters 3 and 4 enable the a priori calculation of the forwarding delay resulting from different possible placement options, to guide the search towards the most performance-efficient solution.

The second optimization problem is named PA-P4SFC-E, and targets finding the optimal embedding of SFCs into P4-enhanced NFV environments at runtime. The optimization problem searches for the optimal placement and routing of SFCs into P4 programmable substrate. The functional and QoS requirements of the different SFC embedding requests are fulfilled by finding the best placement solutions, using the acquired knowledge from Chapters 3 and 4 related to the performance and capabilities of the different available P4 packet processors. Furthermore, a performance-agnostic scenario is designed to serve as a baseline scenario to assess the effectiveness of the proposed performance-aware solution. Finally, a greedy solution is designed and implemented to solve the PA-P4SFC-E problem faster at runtime.

A detailed evaluation of the two problems is conducted, where the model's parameters are populated based on surveyed literature works. The trade-off between the cost and the performance objective functions is highlighted when evaluating the PA-P4VNF-RA problem. On the other hand, when evaluating the PA-P4SFC-E problem, the impact of the different system parameters such as the length of SFCs, and the degree of adoption of IPUs and programmable ASICs is evaluated. Furthermore, the baseline scenario is evaluated, where we found that it requires on average extra 1.68 trials to find the optimal placement if performance awareness does not exist, where each trial includes conducting measurements of the system to check for the satisfaction of QoS requirements. Finally, the evaluation of the greedy solution revealed the effectiveness of the solution for handling scaled-up scenarios in terms of the execution time, on the cost of reduced optimality.

6. P4 Applications: Use Case Studies

After evaluating the performance of P4 programmable devices and optimizing their integration into cloud environments, we provide in this chapter some use cases and applications to demonstrate the advantages of adopting programmable data planes for future networks.

The first application is related to 5G cellular networks. More specifically, we propose a microservice-based design for the 5G User Plane Function (UPF) and provide a proof-of-concept implementation of this design using P4. Then, we discuss how the microservice-based UPF can be deployed into the **P4-enhanced NFV environment** proposed in Chapter 5 to achieve optimized performance levels.

The second application introduces a programmable traffic management solution, where virtual queue-based Active Queue Management (AQM) is used to enforce throughput and delay limits on a per-flow or slice basis. While the programmable traffic management application holds as a stand-alone solution that can be used in many scenarios, it also complements the UPF implementation by enforcing the required QoS levels. The contributions related to implementing and evaluating the programmable traffic management solution are based on our published work [8].

The remainder of this chapter is organized as follows. Background information on 5G Core UPF and programmable traffic management using P4 is provided in Section 6.1 along with the relevant SOTA works. The description of the design, implementation, and integration of the proposed microservice-based UPF into the previously proposed **P4-enhanced NFV environment** is provided in Section 6.2. In Section 6.3, we elaborate on the design and implementation details related to the second proposed application, i.e, the P4-based programmable traffic management solution. Section 6.4 includes a detailed evaluation of the P4-based programmable traffic management solution. Finally, a summary for this chapter is provided in Section 6.5.

6.1. Background and Related Work

In this section, we provide background information and discuss some SOTA works related to 5G Core networks and programmable traffic management using P4 in Subsections 6.1.1 and 6.1.2, respectively.

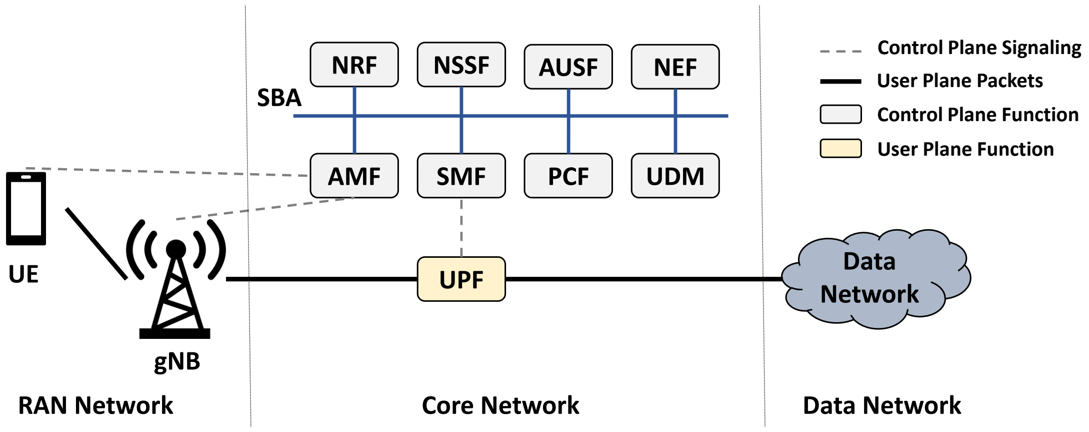


Figure 6.1.: 5G system architecture.

6.1.1. 5G Core Networks and P4

The 5G cellular network is the latest generation of mobile networks. Its major role, similar to its predecessor generations, is to ensure connectivity for mobile users. On top of basic connectivity, 5G increased the network's capacity and enhanced the performance in terms of data rate and latency.

The 5G system is made up of two main components as shown in Fig. 6.1. These components are:

- Radio Access Network (RAN): This component includes User Equipment (UE)s, which represents any end-user equipment. The UE can be mobile and it connects to the Next Generation NodeB (gNB)s, which are part of the RAN. The UE and gNB implement the radio access technology to communicate over the air interface.

- 5G Core: The 5G Core supports the traffic routing between the UE and the Data Network. It also handles security, authentication, and session management-related tasks. It adopts a Service-Based Architecture (SBA) similar to cloud environments, where all the constituting NFs are interconnected.

According to Release 16 5G specifications from 3rd Generation Partnership Project (3GPP) [127], the 5G Core network splits between the control plane and user/data plane as shown in Fig. 6.1. While most of the functions in the 5G Core handle control plane functionalities, the UPF is the only function that handles the user data traffic. In the following, we list and briefly describe some of the NFs constituting the 5G Core control plane:

- Access and Mobility Management Function (AMF): This function interacts with the UE and the gNB to take care of basic access and mobility management tasks.

- Session Management Function (SMF): This function manages the sessions established in the network between UEs and Data Network. It mainly interacts with the UPF to communicate relevant session management information.
- Policy Control function (PCF): This function provides policy rules to other control plane functions to be enforced.
- Unified Data Management (UDM): This function generates authentication credentials.
- NF Repository Function (NRF): This function handles service discovery tasks.
- Network Slice Selection Function (NSSF): This function selects a set of network slice instances to serve a specific UE.
- Authentication Server Function (AUSF): This function takes care of authentication services.
- Network Exposure Function (NEF): This function takes care of exposing capabilities and events to other NFs.

On the user plane side, the UPF handles all the packet processing of user data in the 5G Core network. In the following, we list some of the main functions designated to the UPF:

- Packet forwarding and routing.
- Allocation of IP address and prefix to UE upon SMF request.
- External Protocol Data Unit (PDU) session point for interconnecting with the Data Network.
- Traffic usage reporting for billing purposes.
- QoS handling for user plane traffic by applying, for example, rate-limiting.
- Packet duplication in the downlink direction and elimination in the uplink direction. This may be useful when dealing with handover scenarios while high reliability is required.
- Lawful intercept when needed.
- Downlink data buffering when UE is in idle mode.

There are few prior works in the literature that investigate the usage of P4 language for implementing 5G Core UPF. Shah et al. proposed in [88] offloading a part of the user state in the mobile packet core network to programmable switches to perform signaling

Table 6.1.: Summary of the related works and the completeness of their UPF implementations compared to the proposed implementation described in this chapter.

UPF Tasks	*vEPG* [89]	*PoC UPF* [129]	*P4-based UPF* [91]	*Work in this Chapter*
Packet Forwarding & Routing	✓	✓	✓	✓
GTP-U En/Decapsulation	✓	✓	✓	✓
Packet Detection Rule Inspection	✓	✓	✓	✓
Forward Action Rule Inspection	X	✓	✓	✓
Traffic Usage Reporting	X	X	✓	✓
QoS Handling	X	X	X	✓
Packet Duplication & Elimination	X	X	X	✓
Lawful Intercept	X	X	X	✓

at the data plane. Authors of [90] demonstrate the usage of transport network slicing to accommodate the various network service requirements of different applications. In this direction, they integrate a P4-based implementation of the UPF into their 5G testbed.

Singh et al. [89] implemented a basic set of user plane functions of the virtual Evolved Packet Gateway (vEPG) using P4. They showed that offloading the user plane function to hardware programmable switches can achieve high-performance gains compared to software-based execution. Authors of [129] provide a PoC implementation of 5G UPF using P4 focusing on the interaction of this P4-based UPF with the control plane. Finally, an extended version of the UPF was implemented in [91] using P4, where more UPF-related tasks are realized.

Compared to these prior works, our P4 implementation of UPF is more extensive, covering more functionalities designated to the UPF. Moreover, the proposal in this chapter takes one step further towards designing a microservice-based UPF, wherein a sample separation of the functions assigned to the UPF is proposed. Table 6.1 summarizes some of the UPF tasks implemented in the most relevant SOTA papers compared to the tasks implemented in this chapter.

6.1.2. Programmable Traffic Management using P4

While the P4 language is comprehensive in expressing packet processing logic, its support for programming traffic managers is still limited. As highlighted in Fig. 2.1, while all the processing stages in the V1model P4 architecture are programmable, the traffic manager stage, where packets are queued before leaving to egress stage, is still non-programmable. In the following, we elaborate on prior works that deal with customizing traffic management at the data plane.

Authors of [76] emphasize the importance of tailoring data plane algorithms, such as scheduling and queuing management strategies, to application requirements. Sharma et al. [77] proposed Approximate Fair Queuing, which prioritizes packets to achieve shorter flow completion times and is designed to run on programmable switches; namely, a hardware prototype based on a Cavium network processor, and a programmable switch implementation using P4. They proposed a programmable calendar queue using either data-plane primitives or control plane commands to dynamically modify the schedule status of queues in a follow-up work [78]. Cascone et al. [79] introduce Fair Dynamic Priority Assignment, a design for a packet forwarding pipeline that uses primitives common in data plane abstractions such as P4 and OpenFlow to enforce approximate fair bandwidth sharing. Sivaraman et al. propose a programmable scheduler in [80] that can implement variants of priority scheduling and ideal fair queuing at line rate using a Push-In-First-Out (PIFO) priority queue. PIFO allows packets to be enqueued at any point in the queue (enabling programmable packet scheduling), but it permits dequeueing only from the head. Strict-Priority-Push-In-First-Out (SP-PIFO), an approximation of PIFO queues by First-In-First-Out (FIFO) queues, is introduced by the authors in [81]. Shrivastav proposes a Push-In-Extract-Out (PIEO) data structure to express buffer management policies in [82]. PIEO, like PIFO, keeps an ordered list of elements, but it allows dequeueing from any position in the list by supporting programmable predicate-based filtering at the dequeueing stage. Another work by Mittal et al. [83] demonstrates that the classical Least Slack Time First (LSTF) algorithm approaches being a universal scheduling function. Shrivastav claims in [82] that LSTF has the same limitations as PIFO because it is based on a priority queue abstraction.

Focusing on queue management schemes, and after analyzing different AQM approaches, Sivaraman et al. [76] argue that there is no "one-size-fits-all" algorithm. They enable data plane programmability by attaching an FPGA to the fast path of a hardware switch and implementing Controlled Delay (CoDel) and Random Early Drop (RED) as proof of concept. Kundel et al. [84], [128] demonstrated that such algorithms can be implemented for P4 programmable data planes while elaborating on the P4 capabilities and constraints. [85] demonstrates ways how to overcome P4 limitations when developing one AQM algorithm. The authors of [86] present an implementation of activity-based congestion management using P4, including new target-specific externs for floating-point operations to support rate measurement, activity computation, activity averaging, and drop threshold computation. Finally, authors of [87], uses the P4 context to implement Proportional Integral controller Enhanced (PIE) and RED AQM schemes.

All of these approaches improve queue utilization within shared network infrastructures (links), but they do not provide guarantees on bandwidth and delay on a per slice basis. The authors of [75] use built-in P4 meters and priority scheduling to manage bandwidth per slice without addressing delay requirements. In our proposed virtual queue-based traffic management mechanism, we can support customizable congestion control capabilities for elastic traffic, ensuring both rate and delay limits per slice.

6.2. Microservice-based UPF for B5G Networks

In this section, we elaborate on the first developed application, i.e, microservice-based UPF for B5G networks. In Subsection 6.2.1, we illustrate and motivate the design choices taken for the development of the microservice-based UPF. A proof of concept implementation for this design using P4 is described in Subsection 6.2.2. Finally, the optimal deployment and orchestration of the proposed microservice-based design in cloud environments using the framework proposed in Chapter 5 is discussed in Subsection 6.2.3.

6.2.1. Design Proposal

Compared to monolithic designs, a microservice-based implementation divides the application's logic into smaller and well-encapsulated services that are loosely coupled and can be distributed across multiple computing devices. Each service has its IP address on the network and exposes a language-independent public interface. A Representational State Transfer (REST) API is the most commonly used type of language-agnostic interface, but other communication models exist. When they go live, microservices are typically deployed as containers.

The microservice design of the UPF follows the general design principles for building microservice-based applications. These principles include satisfying the following criteria [130]:

- A microservice should have a single concern, meaning that it should be responsible for a single task.
- A microservice should be discrete, having clear boundaries separating it from other microservices.
- A microservice should be transportable so that it can be moved between runtime environments easily.
- A microservice should carry its own data, where data sharing can take place only via clearly defined interfaces.
- A microservice is ephemeral so that it can be created, destroyed, and restored easily.

In this direction, we propose separating the UPF into the microservices shown in Fig. 6.2. These microservices are:

1. Ingress Steering Function (ISF): This microservice handles the incoming packets

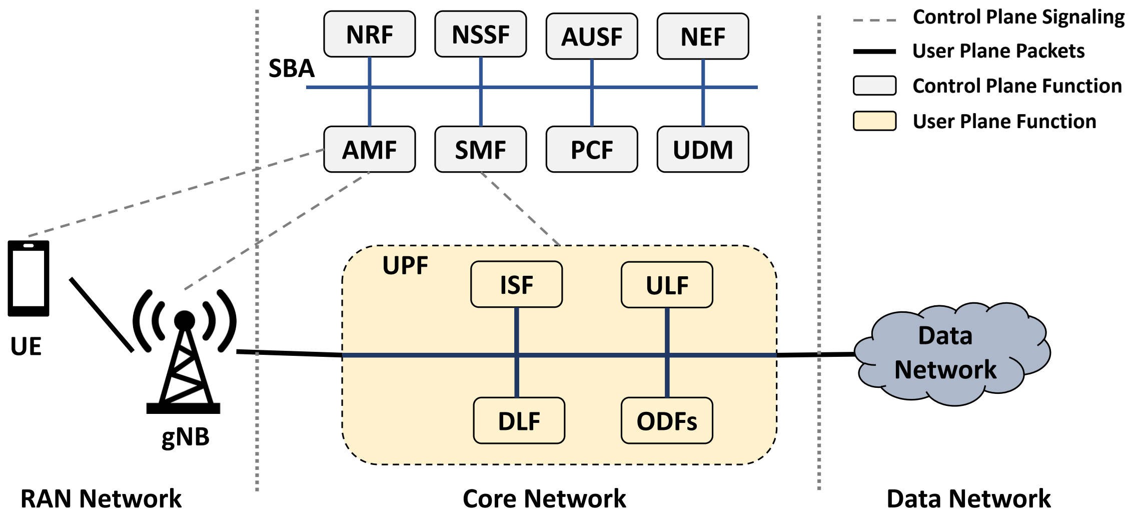


Figure 6.2.: 5G architecture with the microservice-based UPF integrated.

when they arrive at the UPF. First, it checks the validity of the received packets based on defined admission control rules to drop the invalid packets. Then, it steers the packets to uplink or downlink microservices. This steering is done based on checking whether the packet has a GTP-U header (uplink packet) or not (downlink packet). It is also possible to steer the packets based on control plane rules that associate the port on which the packet is received to the uplink or downlink processing paths. Moreover, this microservice can implement load balancer mechanisms to balance the processing workload when multiple replicas of the following microservices (i.e., uplink or downlink) exist.

2. Downlink Function (DLF): This microservice handles the packets coming from the Data Network (DN) to the gNB and later on the UE. It implements all the packet processing required to fulfill the basic tasks assigned to the UPF when handling downlink data traffic.

3. Uplink Function (ULF): This microservice handles the packets coming from the UE through the gNB to the DN. It implements all the packet processing required to fulfill the basic tasks assigned to the UPF when handling uplink data traffic.

4. On-Demand Function (ODF)s: This list of microservices includes all the optional tasks assigned to the UPF, which can be activated on demand. For example, this list may include a lawful intercept microservice, where data is intercepted for some devices and over a certain period of time on a per-demand basis.

The reason for splitting the uplink and downlink user plane processing is to efficiently handle unsymmetric traffic incoming to the core. This way, the resources assigned to processing downlink packet streams can be scaled up/down independently based on the volume of downstream traffic. The same holds for upstream processing resources. Moreover, we decide to make the optional functions of UPF as ODFs to flexibly activate/deactivate them on demand and also to scale them up/down over time as needed.

Note that this proposal should not be the only way to design microservice-based UPFs. Nevertheless, we believe that this design option keeps the dependency between the functions at a low level, where boundaries between functions can be clearly drawn as illustrated in the implementation proposal in Subsection 6.2.2. Note that in this architecture, the interaction of the UPF microservices with the control plane will still be via the SMF, where an intermediate layer or another microservice can take care of that.

6.2.2. Implementation using P4

In this subsection, we elaborate on the implementation of the UPF using P4. The tasks implemented are the ones listed in Table 6.1. Fig. 6.3 shows a flow diagram depicting the UPF processing logic implemented using P4. The distinction between the SOTA open-sourced implementation and our extended implementation is highlighted in the diagram by using different color codes. Moreover, the splitting of the implementation logic into the different microservices described in Subsection 6.2.1 is also highlighted.

Ingress Steering Function (ISF)

When a packet arrives at the UPF, its validity and other admission control rules are checked to keep or discard the packet accordingly. If the packet is valid, it will go through the uplink or downlink processing path based on the availability of GTP-U headers in the packet or based on a control plane rule that specifies the next destination. At this stage, the ISF processing is complete. Note that it is also possible to extend the functionality of ISF by incorporating load balancing functionality to steer the traffic into different ULF and DLF replicas.

DownLink Function (DLF)

In the DLF, the packet goes through the following processing. First, P4 tables are used for packet classification to map packets to their corresponding UE and traffic classes. For example, the UPF can match the packet based on the 5 tuples (i.e, IP destination and source addresses, source and destination ports, and the transport protocol in use) to retrieve the Packet Detection Rule (PDR). The PDR contains information that instructs the UPF on how to handle/process the received packet. Note that a UE may have more than one PDR for its packets corresponding to the different directions of the traffic (uplink or downlink), and the different traffic classes (QoS levels, etc.). The

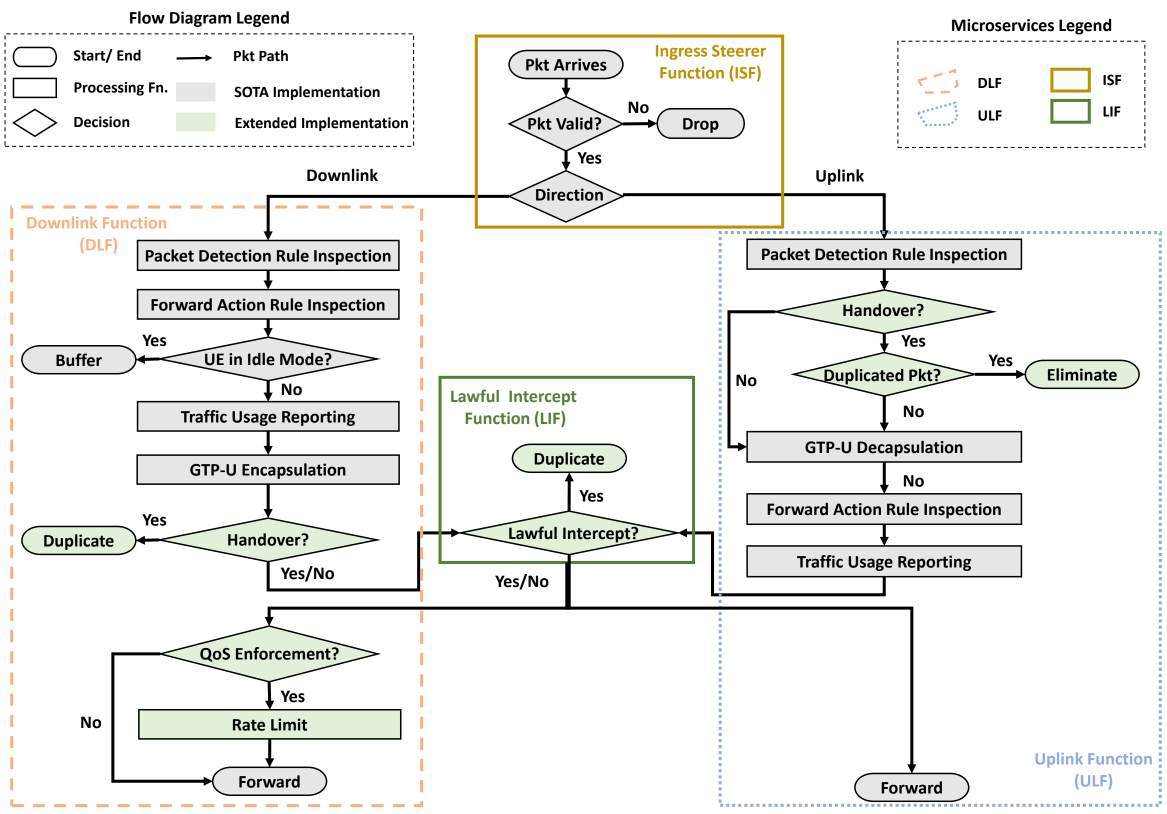


Figure 6.3.: Flow diagram showing the processing logic in the UPF and a sample proposed microservice-based design for it.

control plane is responsible for adding, updating, and removing PDRs into/from the UPF when the UE attaches, moves to another gNB, and detaches from the network, respectively.

According to the PDR, the Forward Action Rule (FAR) corresponding to the packet under process is decided. The FAR specifies the action that will be applied to the packet. These actions could be forwarding a packet in normal cases, buffering packets when a device is in idle mode, or notifying the control plane to wake an idle device. In the FAR inspection step, a P4 table is used again to retrieve the relevant information for applying the action specified by the FAR. For example, to forward traffic in the downlink direction, the Tunnel Endpoint Identifier header field and the IP address of the base station are needed. Then, the packet is buffered if the FAR dictates that. This will be the case when the device is in idle mode. Note that as P4 devices do not support buffering packets, the packets are forwarded to another service that is assumed to be capable of buffering packets such as a database.

Next, P4 counters are used to collect information about the volume of traffic used in the active session. This information is necessary for billing and accounting purposes.

After that, the UPF adds GTP-U, UDP, and IP headers into the packet's header stack between the Layer 2 and Layer 3 headers of the processed packet. The fields in the added headers are populated to ensure the correct forwarding of the packet. Moreover, the QoS Flow Identifier (QFI) field in the GTP-U header is populated to specify certain required QoS levels. Also, the GTP-U header fields related to enabling the reflective QoS are populated. The reflective QoS is used to instruct the UE to use certain QoS levels when sending packets to the DN in the uplink direction.

If the UE is in a handover procedure, the UPF may be required to forward the packet to both the source and destination gNBs of the handover procedure to ensure that the packet is delivered to the UE reliably. If this is the case, the clone-egress-to-ingress P4 action is used to duplicate the processed packet. The forwarding information in the duplicated packet is updated before sending the packet to the other gNB.

After checking on-demand optional functions such as the Lawful Intercept Function (LIF) as shown in Fig. 6.3, the QoS requirements of the active session are checked to enforce the required QoS level using rate limiting. The description of the implementation of this functionality is illustrated in detail in Section 6.3. Finally, the packet is forwarded to its destination.

Uplink Function (ULF)

If the packet is in the uplink direction, it will visit again the PDR inspection step, where P4 tables are used to match the 5 tuples and retrieve the relevant information telling how to process this particular packet.

In the uplink direction, it is possible to receive duplicate packets from the UE through the gNBs if the UE is in a handover procedure. Again, this feature is supported in 5G for resiliency purposes. For such a scenario, the UPF checks such situations and eliminates duplicated packets at an early processing stage. Each active PDU session is assigned a stateful P4 register. The register saves the GTP-U sequence number header field of the last received packet in the active session. If the packet under process has a sequence number less than or equal to the stored number in the P4 register, then the packet is dropped, otherwise, the register will store the sequence number of the packet currently under process. Note that if the order of sequence numbers inside the GTP-U header of incoming packets can be altered due to any enabled reordering mechanisms, then the presented implementation of duplicated packet elimination should be revisited.

Afterward, the packet will go through the decapsulation step, wherein the GTP-U, UDP, and IP headers relevant for packet handling in the cellular network are removed before the packet leaves to the DN.

Similar to the DLF, the FAR is inspected to retrieve relevant forwarding information from the control plane. Note that in the uplink direction, buffering packets is not an option, so packets can only be forwarded.

Then, traffic usage reporting is performed using P4 counters for accounting and billing purposes. Finally, the packet is forwarded to its destination in case no other on-demand functions are active. Note that the QoS enforcement step is only applied in the downlink direction.

On-Demand Function (ODF)

The optional on-demand functions are processed only when these features are activated. For example, in Fig. 6.3, we show the implementation of LIF, where packets are duplicated using P4 clone-ingress-to-egress action, and then sent to another destination in case the lawful intercept feature is active.

Discussion on Limitations

It is worth raising some of the limitations in P4 language and P4 targets, which limit the full implementation of UPF using P4. For example, as P4 language syntax and P4 targets handle packets one at a time and do not support storing full packets inside the targets, it is not possible to fully implement buffering mechanism needed in the downlink processing path.

Moreover, to identify duplicate packets to be eliminated in the uplink processing, P4 registers are needed to keep track of the packets' sequence numbers in the active PDU session with the ongoing handover procedure. Accordingly, the maximum number of available P4 registers in a P4 target limits the number of active PDU sessions with ongoing handover procedures that can be supported by the P4 target.

Finally, the limited number of rules that can be supported by a P4 target also limits the number of active PDU sessions that can be supported by the running UPF, which needs rules from the control plane to get instructed on how to process the packets corresponding to the different active sessions.

6.2.3. Deployment into P4-enhanced Cloud Environments

The proposed microservice-based UPF can be orchestrated using the framework described in Subsection 5.1.1 as depicted in Fig. 6.4. This integration serves as a PoC demonstration for the usage of the framework presented in Chapter 5 to manage a specific workload scenario related to 5G Core user plane processing.

In this case, the requests to embed UPF instances, presented as SFCs, will be the

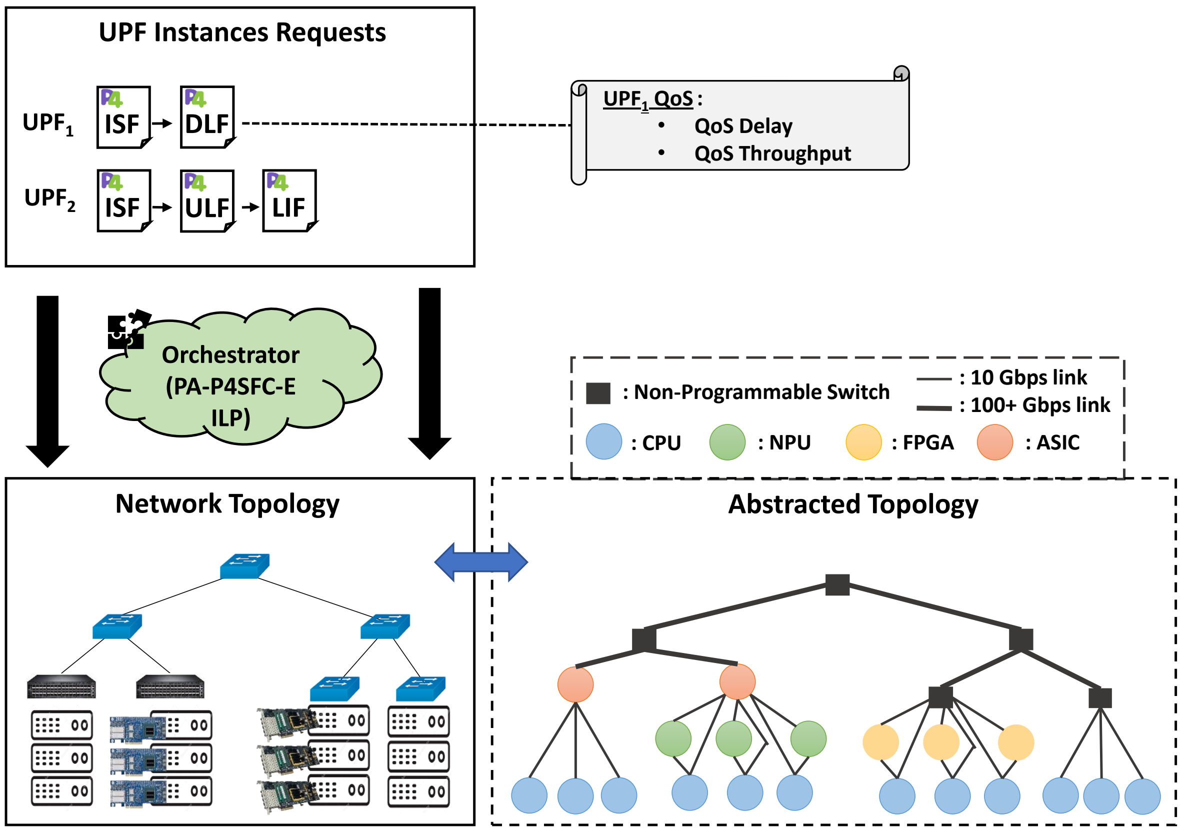


Figure 6.4.: PoC demonstration on the integration of the microservice-based 5G Core user plane processing into the framework presented in Chapter 5.

workload to be handled by the cloud infrastructure. Each UPF instance, or SFC, is made up of a series of microservice functions that together constitute a complete UPF. For example, if a UPF instance is designated to do only basic downlink processing, then it will be made up of ISF followed by DLF. Alternatively, the UPF instance can be made up of ISF, followed by ULF, followed by LIF to perform uplink processing followed by the optional (on-demand) lawful intercept task. Each UPF instance embedding request can be associated with QoS requirements in terms of throughput and delay. Different QoS requirements can define different UPF instances that belong to different network slices.

The PA-P4SFC-E problem defined in Section 5.3 can be used to optimize the placement of the SFC requests into the heterogeneous cloud environment with a programmable substrate. The optimization solver can reside in the orchestrator entity of the cloud environment.

6.3. P4-based Programmable Traffic Manager

In this section, we describe in detail the implementation of the programmable traffic management solution. This function can be used for the QoS enforcement step in the UPF processing as discussed in Subsection 6.2.2, where the QFI GTP-U header field is populated to specify certain required QoS levels in terms of throughput and delay.

The description of this implementation is done in a separate section because it holds as a stand-alone implementation that can serve different applications such as guaranteeing some performance levels for different network slices, besides serving the QoS requirements in the 5G Core UPF. Note that slices and flows are used interchangeably in the following as we assume that each slice is made up of a single flow. The contributions presented in this section are based on our publication [8].

In the following, we describe the design and then the implementation of the P4 programmable traffic management solution in Subsections 6.3.1 and 6.3.2, respectively. Then, in Subsection 6.3.3, we comment on the advantages of our proposed solution compared to the standard P4 meters, which serve as a baseline solution. Finally, in Subsection 6.3.4, we discuss some identified portability issues when applying the solutions to different P4 devices and the ways applied to mitigate these issues.

6.3.1. Design

The proposed P4 programmable traffic management design allows for the customization of traffic characteristics associated with different flows. In this context, we ensure that the proposed design also meets the following criteria:

1. **Traffic Customization**: The design should enable managing traffic characteristics on a per slice basis, where rate limits and maximum tolerable latency can be set.

2. **State Isolation**: Slices should be managed and controlled independently while customizing their traffic characteristics. As a result, state information per slice should be kept in the data plane.

3. **Performance Isolation**: Service-level key performance indicators should be met per slice, regardless of congestion and/or performance levels of other slices sharing the same infrastructure.

The proposed solution incorporates a *Traffic Classifier* and a *Virtual Queue based Traffic Manager*. The design of these components is described in the following.

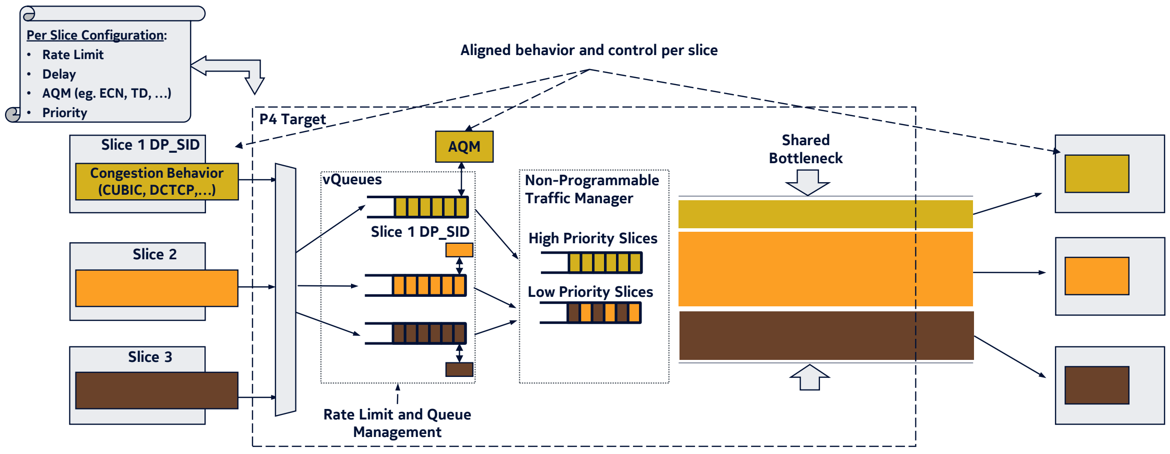


Figure 6.5.: Virtual queues and data place slicing.

Traffic Classifier

We perform per-flow traffic classification using a match-action table, assuming that the controller assigns a unique local id per slice called *Data Plane Slice ID* (*DP_SID*). The VxLAN tag of incoming traffic, for example, can be used as *DP_SID*. This *DP_SID* is read from the table and stored in the packet's user-defined metadata to classify the packets based on their originating flows. This classification enables the data plane to make local policy decisions (i.e., QoS enforcement) on a per-packet basis. Note that this *DP_SID* can be mapped to the QFI if this proposed traffic management solution is used in the context of QoS enforcement for UPF as discussed in Subesection 6.2.2.

Virtual Queue based Traffic Management

A Virtual Queue (vQueue) is a mechanism used to model the length of the queue, or in our case the sojourn latency, as if the packets arriving at the real queue were served by a link with a capacity less than the actual capacity of the link. It contains no packet data. It is a number that is incremented as packets arrive and decremented based on the model. The virtual queue's latency, for example, can be used to drive an AQM scheme, replacing the same metric from the real queue. A detailed description of the structure and applications of vQueues in an exemplary AQM can be found in [73].

The vQueue is implemented using commonly supported P4 constructs (i.e., registers) as a design option, to allow easy portability of the implementation across different P4 targets. Furthermore, by associating a network slice with a vQueue implemented with registers, we can keep network slice state information. The control plane can also access these registers during runtime, allowing for per-slice monitoring and management. Because we cannot use custom (e.g., multi-dimensional) data structures to index such registers, we use the predefined local identifier *DP_SID* to allocate memory for a specific slice and its corresponding port in a network element.

Each vQueue in the example used in this section is associated with a network slice, as color-coded in Fig. 6.5. As a result, by dropping or marking excess traffic, virtual queues and thus network slices can be individually rate limited, while each vQueue can be assigned a unique AQM realization based on the requirements of each network service. An exemplary use of different queue management schemes and transport-layer congestion control schemes to satisfy the requirements of the different network slices and their running applications is illustrated in Fig. 6.5, where local queueing latency limits are enforced. Furthermore, when available, we use the scheduling capabilities of the non-programmable Traffic Manager to ensure performance isolation, where each slice can be assigned a priority based on its application requirements. We assume that the priorities of different network slices are calculated and pushed top-down to the data plane by the control plane. As depicted in Fig.6.5, the delay-sensitive traffic from *Slice 1* can be prioritized over traffic from *Slice 2* and *Slice 3* to be forwarded with minimal delay. The need for different priorities or the opportunity to share priorities will be determined by how slice rate limits relate to the worst-case real bandwidth limit remaining and latency caused by other higher and equal priority slices. It is beyond the scope of this work to discuss how the control plane will assign these priorities, but the equations coming in Subsection 6.4.2 could be used.

6.3.2. Implementation using P4

For each slice, we apply traffic management rules that match the corresponding *DP_-SID* carried by every packet in the user metadata. We can apply rate limiting and queue management per slice using vQueue in the *queue_manage* action as can be inspected from Listing 6.1), depending on the type and version of the transport protocol used and application requirements. We demonstrate our approach by implementing two basic schemes: Tail Drop (TD) and Explicit Congestion Notification (ECN) step AQM to support both classic (e.g., TCP Cubic) and scalable (e.g., Data Center TCP (DCTCP) [74]) congestion controls.

For the sake of efficiency, we use a single table to perform both per-flow traffic classification and the standard IPv4 forwarding functionality L3Fwd. The L3Fwd match-action table is part of the P4 pipeline's ingress processing, and it forwards the packet to the appropriate egress port while also injecting the *DP_SID* into the packet's user-defined metadata. In our example, a slice is a collection of flows with the same destination IP address.

Listing 6.1 describes the per slice vQueue management action. For the sake of brevity, we assume equal-sized packets and define the rate using a packet transmission time parameter. The algorithm is easily adaptable to use byte transmission time rather than packet transmission time to account for variable packet sizes. The algorithm is implemented by using (i) the *slice_ts* register, which stores the global timestamp of the previous packet for the slice at the control block (lines 7-13), and (ii) the *delay* register, which holds the virtual queue size for the slice (line 19). We determine how long the current packet has to wait in the virtual queue based on the time that has passed since the previous packet of the vQueue (slice) was processed (lines 20-24). We define *C_DELAY* parameter for each virtual queue to be the maximum tolerated delay in milliseconds whose value is prescribed by the Service Level Agreement (SLA) for the slice. The burst limit is calculated using this delay, the average packet size, and the rate limit (maximum throughput) specified in the SLA. If the size of the vQueue, after adding the current packet's transmission delay (*T_DELAY*), exceeds the burst size *C_DELAY*, the packet is dropped; otherwise, the vQueue size is incremented by the packet's transmission delay (lines 25-29). In any case, the delay register is accordingly updated (line 30).

If the used TCP version at the flow level supports ECN functionality, it is possible to ECN mark the packets when the virtual queue delay exceeds the marking burst limit denoted by *M_DELAY*, indicating congestion (lines 32-34). The latter can be used as a proactive measure to control the traffic before the drop burst limit *C_DELAY* is reached. The algorithm can be extended to support more complex schemes, wherein distinct AQM mechanisms can be implemented in separate actions.

The P4 code is implemented for two P4 targets: software BMv2 switch and hardware Agilio CX SmartNIC. Several target-specific modifications were required to enable running the solution on each of the two targets. The queue admission control should be performed at the ingress stage before enqueuing packets. However, due to some SmartNIC target limitations, meters do not work in the ingress. Nevertheless, since queues are located after the egress pipeline in the case of the SmartNIC, metering functionality can be safely executed in the egress pipeline, but this violates the P4 program's generality. A P4 program in the BMv2 case can access information about the real queue in the egress pipeline as part of the packet metadata. As a result, we can also check the true queuing delay and decide whether to drop or mark a packet. By including the relevant parameters received at ingress in packet metadata fields, the additional match at egress can be avoided. Finally, for multi-threaded targets, concurrent execution by threads that manipulate the same memory locations can cause inconsistency issues. The @atomic feature of the P4-16 language can be used to instruct the compiler to execute a code block atomically.

6.3.3. Advantages Compared to P4 Meters

For bandwidth management, standard P4 meters can be used [75]. In this work, we consider the metering mechanism as the baseline scenario compared to our proposed traffic management mechanism. In the following, we argue about the advantages of the proposed vQueue-based management solution compared to the meter-based baseline

```
action queue_manage(bit<64> T_DELAY, bit<64> C_DELAY, bit<64> M_DELAY){
  bit<64> delay=0;                                                //Delay reported
  meta.ts=hdr.intrinsic_metadata.current_global_timestamp;
  bit<64> c_ts = meta.ts;
  bit<64> p_ts;
  bit<64> delta = 0;
  @atomic {                                                       //Update timestamps
    slice_ts.READ_REG(p_ts,meta.slice_id);
    slice_ts.WRITE_REG(meta.slice_id, c_ts);
  }
  if ((p_ts==0) || (p_ts>c_ts)) {                                 //For reordered packets
      p_ts = c_ts;
  }
  delta = c_ts-p_ts;
  if (delta >= 3294967296) {                                      //Wrap up timestamps
      delta=delta-3294967296;
  }
  @atomic {                                                       //Update delay
      slice_delay.READ_REG(delay,meta.slice_id);
    if (delta > delay) {
        delay = 0;
    } else {
      delay = delay - delta;
    }
    if (delay + T_DELAY > C_DELAY) {
      meta.DropFlag = 1;                                          //Drop Packet
    } else {
      delay = delay + T_DELAY;
    }
    slice_delay.WRITE_REG(meta.slice_id,delay);
  }
  if ((meta.flag == 0) && (hdr.ipv4.ecn != 0) && (delay > M_DELAY)) {
      hdr.ipv4.ecn = 3;                                           //Mark Packet
  }
}
```

Listing 6.1: Queue Management P4 Action (excerpt).

solution.

Meters are used to record statistics and the state of a flow to maintain the state. This data can be used to drop or mark packets based on burstiness and bit rate criteria. For packet classification, P4-16 supports the two rate Three Color Marker (trTCM) [125]. The trTCM meters a packet flow and colors its packets green, yellow, or red based on their Peak Information Rate (PIR) and Committed Information Rate (CIR), as well as their associated burst sizes. The P4 program can make use of this metering information to implement custom actions such as dropping or marking packets based on their meter-related colors.

In terms of bandwidth management, the use of P4 meters and, optionally, priority schedulers should suffice. However, to support the delay requirements for elastic congestion-controlled traffic, we must use additional traffic management schemes (e.g., AQM, etc.) to deal with congestion. Yet, the traffic management logic in forwarding devices is still not programmable. Virtual queues are used not only to enable programmable active (virtual) queue management and to regulate the load on the actual queue(s) but also as a slicing abstraction that allows for the implementation of stateful data plane algorithms on a per slice basis. This also enables the integration of new traffic management approaches easily.

Additionally, some P4 targets may not support metering functionality in their architecture as in the case of NetFPGA-SUME [15]. Moreover, even if meter functionality is available on a P4 target, it may have proprietary behavior or limited functionality. With meter-based rate limiting, the state variables (current queue/burst size), which could be useful in the implementation of AQM mechanisms, may be hidden.

6.3.4. On the Portability of P4 Implementations

The proposed vQueue-based approach and the baseline indirect P4 meters solutions are implemented and evaluated on the BMv2 software switch and Agilio CX SmartNIC. In the following, we summarize the lessons learned from this exercise focusing on the identified portability issues.

- The P4 registers used in the vQueue implementation are not synchronized with the SmartNIC's in-hardware flow cache. This results in a flaw in the design logic unless we disable the *cache-flow* option on the SmartNIC, even though this may affect the card's performance.
- The SmartNIC's multi-thread processing causes a lack of synchronization between register read/write operations. This problem was resolved by requiring read and write register operations to be "atomic operations" (lines 7, 18).
- The time-stamping mechanisms of the two targets are not the same. Because the SmartNIC's 64-bit time is represented by two 32-bit fields (seconds and nanoseconds), we must subtract $2^{32} - 10^9$ whenever one second is exceeded (lines 15-16).
- Although the SmartNIC is compatible with the v1model architecture [118], still, its queues are located after the egress pipeline, which makes it impossible to read standard metadata fields that report queue occupancy in the egress stage. As a result, we can only observe the status of the virtual queue and not the real queues when evaluating the SmartNIC in Subsection 6.4.1.
- The assignment of different queue priorities on SmartNIC is currently not well supported [126].
- When meters are executed in the ingress pipeline, the SmartNIC behavior is undefined. As a result, we had to use traffic policing at the egress pipeline (via the queue_manage action or meters execution).
- The SmartNIC meters have only one threshold. Therefore, two meters have been used in tandem to implement the trTCM.

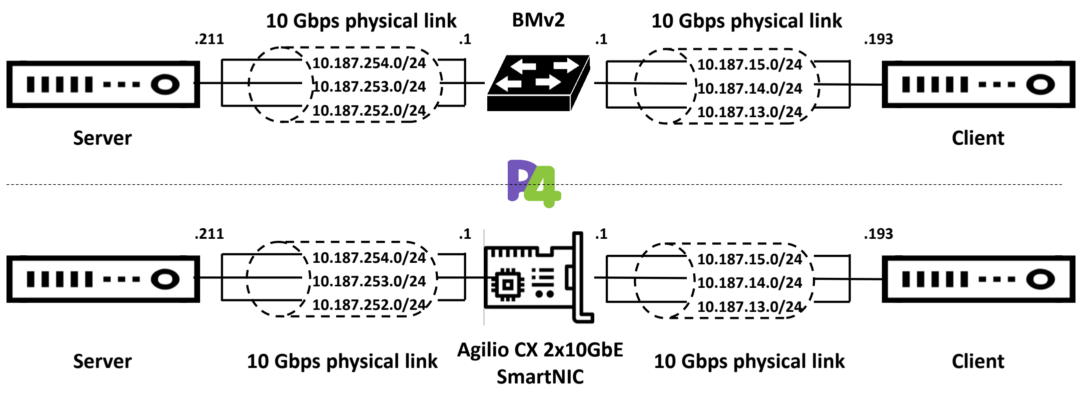


Figure 6.6.: Setup used for conducting the rate and delay management experiments.

6.4. Evaluation of the P4-based Programmable Traffic Manager

This section focuses on a set of representative experiments that demonstrate the level of flexibility and portability of our proposed programmable traffic manager solution by utilizing two different state-of-the-art P4 targets: the BMv2 software switch and the Agilio CX SmartNIC. We validate and compare the performance of our approach, denoted as *vQueue*, to the baseline approach described in Subsection 6.3.3 using P4 *Meters* [75].

Subsection 6.4.1, in particular, validates the effectiveness of the proposed approaches in controlling throughput and delay per slice, ensuring performance isolation. Subsection 6.4.2 investigates the operational limits and trade-offs of the proposed vQueue approach. Finally, in Subsection 6.4.3, we compare the processing efficiency of the vQueue implementation in the SmartNIC setup to that of the Meters.

6.4.1. Rate and Delay Management

Evaluation Environment and Reporting

As shown in Fig. 6.6, the testbed used in this set of experiments consists of three Linux machines that serve as traffic client, server, and host for the P4 target. The client and the server are configured with Macvlan [] to assure traffic isolation at the hosts. We build two experimentation setups: one for the BMv2 switch and the other for Agilio CX SmartNIC. In the first case, we use three machines each with a different Intel CPU (i7-4770 @ 4x3.40GHz, Pentium D @ 2.8GHz, and i5-4590 @ 4x3.30GHz) running Ubuntu 16.04 with Linux Kernel version 4.4.0. Each machine has at least two 1GbE NICs with MTUs set to 1500 bytes. In the Agilio CX SmartNIC setup, we use machines running

Table 6.2.: Per slice traffic characteristics.

Slice / Parameter	DC		Enterprise		Peering	
	BMv2	SmartNIC	BMv2	SmartNIC	BMv2	SmartNIC
RTT (msec)	5		10		30	
TCP Flavor	DCTCP		Cubic		Cubic	
TCP flows (#)	10	100	1	10	10	100

18.04 Ubuntu based on Linux Kernel version 4.19.0, each with 16 cores (dual-socket Intel Xeon CPU E5-2630 v3 @ 2.40GHz), 64GB of 2133 MHz Double Data Rate Fourth Generation (DDR4) memory, and an 82599ES 10GbE NIC.

In terms of reporting, the IPv4 header's identification field has been repurposed to report measurement results. This is done solely for the purpose of validating the Proof of Concept (by generating the corresponding graphs) and is not required for the real-world deployment of the proposed approach. When a BMv2 switch is used as the target, the six least significant bits store the packet's virtual queuing delay in milliseconds (ms), the next six bits report the packet's real queuing delay in milliseconds (up to 63ms), and the four most significant bits report the number of dropped packets in the first next non-dropped packet of the same slice. Because of size constraints, up to 15 drops can be reported. Any additional dropped packets will be carried over to the next packet, bringing the maximum reportable drop rate to 15/16 = 94%. When virtual queues are not used, the packet's real queuing delay in milliseconds is stored in all 12 least significant bits (up to 4s). In the case when the SmartNIC is used as the target, the real queuing delay cannot be reported, as discussed in Subsection 6.3.4. Therefore, in this case, we use the 16-bit identification field to report the virtual queuing delay (in milliseconds) using the least 9 significant bits and the number of dropped packets using the upper 7 bits.

Network Slice description, traffic & configuration

We assume three network slices, each with its own set of requirements based on its intended use: (i) data center interconnection (*Data Center (DC) slice*), (ii) enterprise Wide Area Network (WAN) connectivity (*Enterprise slice*), and (iii) peering between two virtual network operators (*Peering slice*).

Table 6.2 contains the traffic characteristics per slice (number of TCP flows and RTT values) used in the experiments. The greedy TCP traffic is only limited by the congestion control scheme chosen and its interaction with drops and marks by the slice's configured AQMs. Before running these applications, the congestion control is set to Cubic or DCTCP on the client(s) and server(s).

Table 6.3 lists the configuration parameters required for the three slices when running

Table 6.3.: Per slice configurations

Slice / Parameter	DC		Enterprise		Peering	
	BMv2	SmartNIC	BMv2	SmartNIC	BMv2	SmartNIC
Rate limit (Mbps)	48	480	12	120	240	2400
Burst limit (msec)	20		10		30	
(pkts)	80	800	10	100	600	6000
Queue Management Scheme	ECN_Step		Tail Drop		Tail Drop	
Target delay (msec)	5		-		-	

on the BMv2 switch and the SmartNIC. To match the forwarding devices' throughput capabilities, which are relatively limited, particularly in the case of the BMv2 reference software switch [1], we have reduced the aggregated slices' rate limit (approximately threefold decrease for the BMv2 switch and the SmartNIC compared to their respective 1Gbps and 10Gbps line rate values).

The rate and target delay (burst limit) per slice are initially set arbitrarily. The burst limit is set as the delay caused by a burst of packets to a real queue served by the specified rate limit. We assume that the most stringent delay and loss requirements apply to *DC slice*. As a result, the DCTCP active queue management algorithm with an immediate ECN step is used [74]. We use ECN packet marking to control the virtual queue's delay to a 5ms target while avoiding loss by allowing a larger (exceptional) burst limit before packets are dropped. The other two slices only use their drop-based burst limit, resulting in a delay-based TD (virtual) queue.

Experiments, Measurements & Evaluation Metrics

The **first** experiment (*Experiment 1- BMv2 software switch*) assesses the efficacy of the proposed approach using *vQueue* for policing and AQM, as well as the system's interaction with TCP flows on the **BMv2 switch**. The results are compared to the baseline *Meter*-based implementation. We run the three slices concurrently over the shared non-congested link. All traffic is routed to the same physical queue at the switch's egress port, with the size set high enough (e.g., 200k packets) to ensure that congestion control is performed only at the virtual queues by the queue management algorithm.

In this experiment, we measure the virtual queue and the physical queue's throughput and delay over time. Measurements are taken 50 seconds after the experiment begins to capture only the steady state and span a time interval of 250 seconds. The packets are captured at the outgoing switch-to-client interface. The throughput plots are based on averages taken over one-second intervals. To see the TCP variations, we measure the queuing delay per packet and zoom in on the plot (over a 10 s period). We also plot

[1]https://github.com/p4lang/behavioral-model/blob/main/docs/performance.md

the queuing delay's CDF. The queuing delay CDFs are based on the per-packet queue delays counted in bin-sizes of 1024mus, ranging from 0 ms to 30 ms.

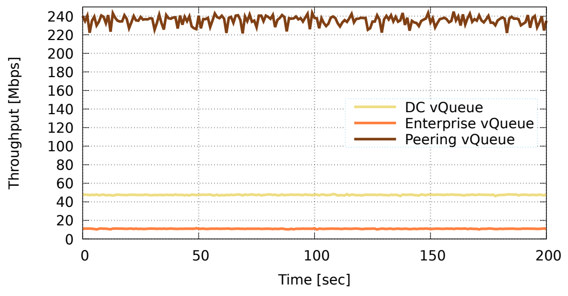
The **second** experiment (*Experiment 2- Agilio CX SmartNIC*) is performed to assess the performance of the *vQueue* management mechanism when running on the P4 programmable **Agilio CX SmartNIC**. In this experiment, the same measurement procedure described for the first experiment is used. However, because the physical queue follows the ingress/egress pipeline in the SmartNIC architecture/implementation, we do not report the packet delay in the physical queue. The purpose of the experiment is to validate the mechanism's portability by running it on a hardware target.

The **third** experiment (*Experiment 3- BMv2 performance isolation with over-utilized link*) is conducted to investigate the potential benefits of the proposed *vQueue* implementation for slice performance isolation (i.e., meeting throughput and latency bounds per slice). For this purpose, we repeat the previously described experimental procedure, but this time with the three slices operating concurrently over a shared bottleneck. To this end, we rate limit the link on which packets leave the switch to 285Mbps, which is just under the combined capacity of the three slices of 300Mbps. Furthermore, in addition to vQueue-based traffic management, we make use of the target's non-programmable traffic manager. In particular, we use traffic prioritization with two priority queues, assigning traffic from the DC and Enterprise slices to the highest priority queue. This experiment is only carried out with the BMv2 target because real queuing delay cannot be measured on the SmartNIC and configuring queue priorities with P4 is not well supported on this target.

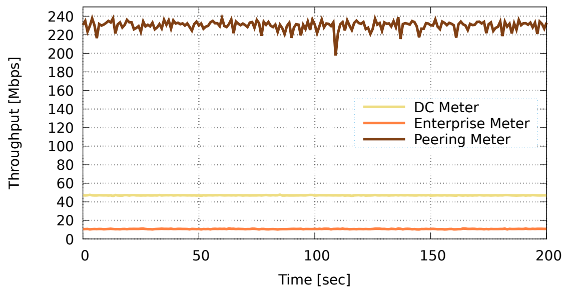
Experiment 1- BMv2 software switch

The results of the BMv2 switch experiment are shown in Fig. 6.7. The plots in Figs. 6.7d, 6.7e, and 6.7f display the per packet and average delay in the virtual and real (switch egress port) queues, as well as packet drops (zoomed in for the first 10 seconds of the experiment) for the DC, enterprise, and peering slices, respectively. The delay CDFs over the duration of the experiment is depicted in Fig. 6.7c. The measured average throughput for all slices when using vQueue implementation is shown in Fig. 6.7a, while Fig. 6.7b depicts the average throughput in the baseline case when using the standard P4 packet classifier/meters.

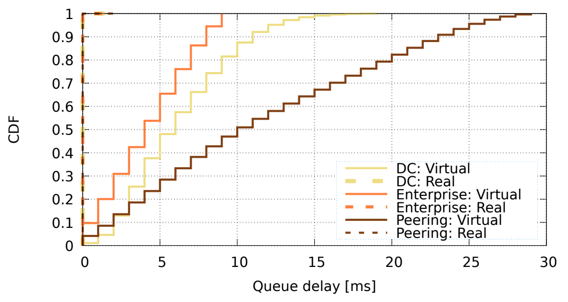
The gauged TCP throughputs for the three slices in Fig. 6.7a confirm the effectiveness of the policing approach using vQueues; the results are similar to the baseline in Fig. 6.7b, where we limit the per slice rate using P4 meters. Unlike the off-the-shelf P4 meters-based implementation, the vQueue implementation can also control the behavior and limits of each slice in terms of drop/target delay (when applicable) besides throughput. This is checked by inspecting the delay CDF plots when using vQueue implementation depicted in Fig. 6.7c. The results demonstrate that for a non-congested physical link, the real queuing delay for all slices sharing that link is in the order of microseconds



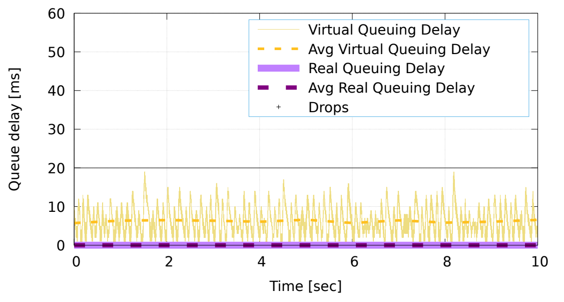
(a) vQueue throughput.



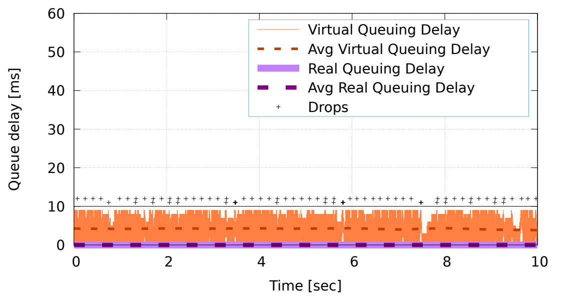
(b) Throughput meters.



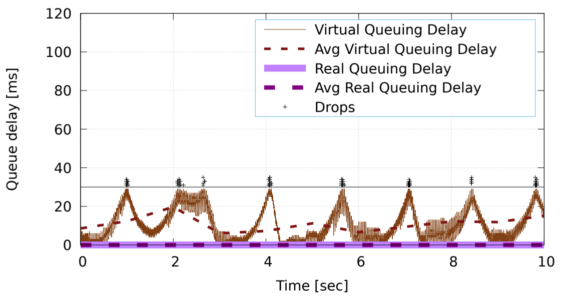
(c) CDF.



(d) DC slice.



(e) Enterprise slice.



(f) Peering slice.

Figure 6.7.: Results of BMv2 software switch in the first experiment.

(dashed plots at very low values), while the virtual queue delay is kept below the corresponding burst limit and the hard rate limits are never violated.

Looking at the vQueue results corresponding to DC slice in Fig. 6.7d, we can observe that the DCTCP flows (slice DC) are controlled at the target rate and around the 5ms marking threshold (in line with the slice configurations set in Table 6.3). Because packets are marked rather than discarded, the average virtual queue delay for DCTCP flows can exceed the 5ms marking threshold. The additional 15ms virtual queue size is required to reduce packet drops, which still occur on occasion when this threshold is exceeded. The higher delay variation in the virtual queue is due to the virtual queue's fast integration function and the TCP senders' delayed mark response.

The Enterprise slice results plotted in Fig. 6.7e reveal that due to the non-shaped (non-ACK-paced) traffic, the full congestion window of the single flow is transmitted repetitively at line rate, resulting in a very bursty virtual queuing delay. Because the virtual transmission time of a single packet is 1 ms on a virtual rate limit of 12 Mbps, only bursts of approximately 10 packets per RTT (10 ms) are permitted. This reduces the actual rate to just above 10 Mbps.

We can also see the Cubic TCP behavior (in Reno mode) for the flows of the Peering slice in Fig.6.7f. The throughput variations are due to the abrupt 30% Cubic backoff on loss and slow increase. A high level of jitter is observed due to line-rate bursts at congestion window size, but it is not limited by the virtual queue's burst limit in this case. We see a high level of synchronization even with 10 flows. This results in frequent episodes of minor under-utilization, as seen in the throughput and CDF plots. It is worth noting that the real queuing delay of the three slices is always in the microsecond range, and thus close to 0 ms.

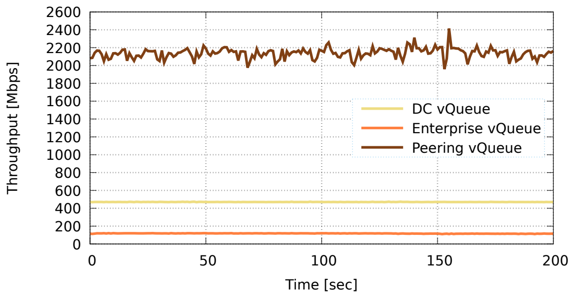
Finally, the results demonstrate the efficacy of the proposed approach in supporting alternative traffic management schemes per slice, tailored to the TCP flavor used and application requirements. Slice (state and performance) isolation is obviously possible when sharing a real queue and capacity, provided that both are sufficiently large to accommodate the slices.

Experiment 2- Netronome SmartNIC

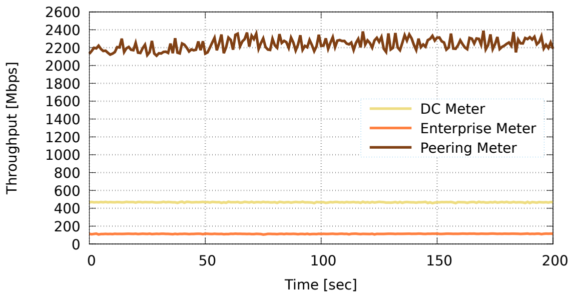
The results of the SmartNIC experiment, like the BMv2 experiment, are shown in Fig 6.8. In general, we see a similar trend to the BMv2 results but on a larger scale (x10 flows). The throughput results for the three slices in Fig. 6.8a are similar to the baseline implementation results depicted in Fig. 6.8b. Although we can only report the virtual queue delay in the SmartNIC setup for reasons discussed earlier, the virtual queue delay for all three slices is kept below the corresponding burst limit, and the hard rate limits are followed.

More specifically, the DCTCP flows (slice DC) in Fig.6.8d fluctuate around the 5ms marking threshold, similar to BMv2. However, because congestion control (via marking) takes a few RTTs to respond and reduce the rate to the target value, DCTCP traffic fills the virtual queue in this case. The synchronized TCP oscillation, manifested as delay variation and the typical on/off marking pattern, is smoothed out after about two seconds as on/off marking is broken due to micro marking bursts spread more evenly over time.

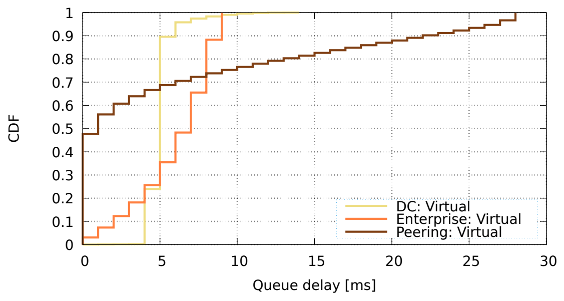
Looking at the delay plot of the Enterprise slice in Fig. 6.8e, we observe a tenfold increase in the number of dropped packets, which is attributed to the tenfold increase in the number of flows. We also see better utilization of the slice capacity, indicating that no TCP flow synchronization is taking place.



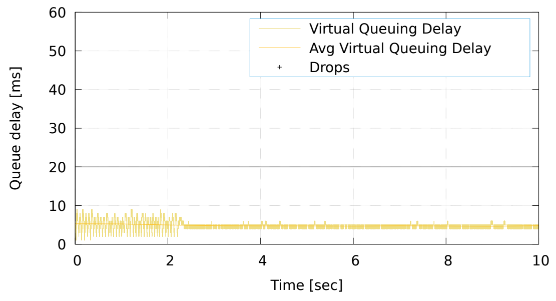
(a) vQueue throughput.



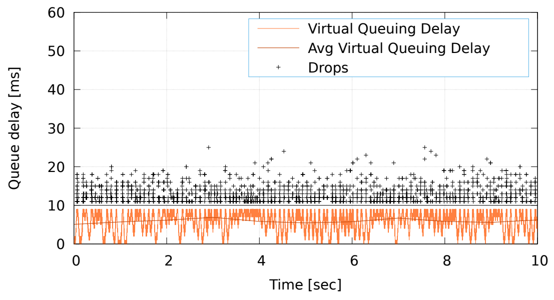
(b) Throughput meters.



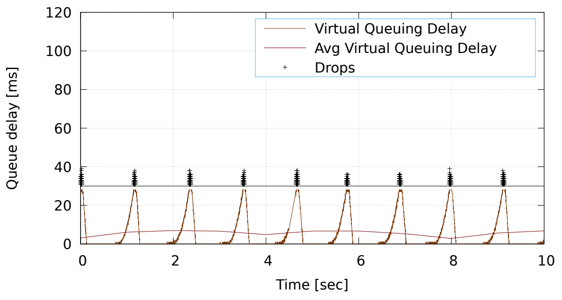
(c) CDF.



(d) DC slice.



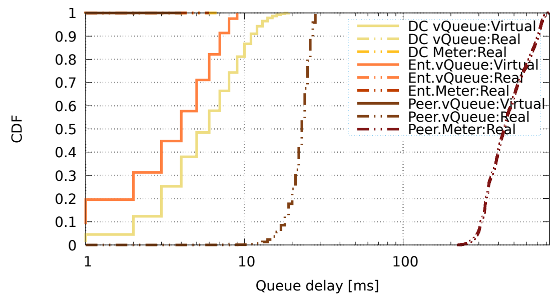
(e) Enterprise slice.



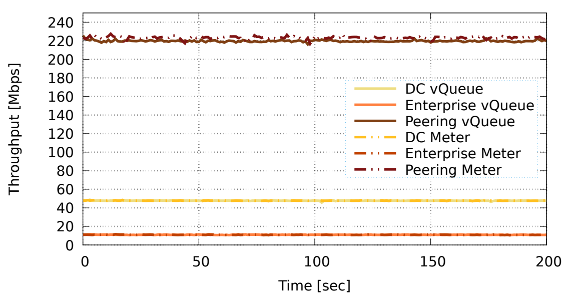
(f) Peering slice.

Figure 6.8.: Results of SmartNIC software switch in the first experiment.

For the Peering slice in Fig. 6.8f where the number of flows is scaled up (x10), the effects of global synchronization noted in Fig. 6.7f for the Cubic TCP flows are amplified. This results in increased under-utilization of the slice capacity, as shown by the CDF and throughput plots. To break synchronization issues, one could use our vQueue implementation and a random dropping scheme, such as the RED-like scheme, enforcing a linear increase in the drop probability starting at a minimum delay threshold and ending at a maximum threshold where 100% of the packets are dropped.



(a) CDF.



(b) Throughput.

Figure 6.9.: BMv2 - performance isolation with over-utilized link results.

Experiment 3- BMv2 performance isolation with over-utilized link

In the context of network slicing, performance isolation means that service-specific performance requirements are always met on each network slice instance, regardless of the congestion and workloads of other slice instances running on the shared infrastructure. The corresponding delay and throughput, using priority queues for both the virtual queue and P4 meters, are depicted in Fig. 6.9a and Fig. 6.9b. The DC and Enterprise slices traffic is prioritized over the Peering slice traffic, where they share the switch's high-priority physical queue. The three slices operate concurrently, sharing the limited link capacity of a single egress port (285 Mbps). Because the real rate limit is less than the sum of the virtual rate limiters, the real queue delays must also be managed. To that end, the virtual queue policies we implemented use the maximum of packets' virtual and real queue delays for drop and mark decisions.

In the vQueue case, we see that the QoS requirements of the two high-priority slices are met, while the low-priority one (Peering Slice) sees a decrease in throughput (Fig. 6.9b) and an increase in the packets' real queue delay, while this delay remains within the pre-defined bounds (30 ms) (Fig. 6.9a). While in the cases of DC and Enterprise slices, there is no real queuing delay because their traffic is prioritized, the Peering slice is controlled by the real queuing delay (Peer.vQueue:Real). The vQueue is no longer operational in the Peering slice case because it cannot reach the virtual rate limit and it reaches the burst limit of 30 ms as long as the higher priority slices use their maximum throughput levels.

In the standard P4 meters case, the throughput results are similar to vQueue results as shown in Fig. 6.9b, with the same noticeable decrease in throughput for the Peering slice. Looking at the delay results in Fig. 6.9a, also there is no real queuing delay for the DC and Enterprise slices, as expected because their traffic is prioritized. However, because it is not controlled as in the vQueues case, the increase in packets' real queuing delay for the low-priority traffic (Peering slice) is significant (up to 800ms).

From these results, we can realize that using the P4-standard meters does not sufficiently isolate the performance of the low-priority slice when the delay is considered the key performance indicator for performance isolation. Meters only support under-provisioned rate limits when latency guarantees are required, unless additional real queue latency configuration and checks are implemented, similar to our vQueue implementation. To avoid the use of additional real queue latency controls, we must manage the capacity margin between the (configured) sum of the slices' (vQueues) throughput and the physical link capacity.

6.4.2. Guaranteed Performance Targets Using vQueues

As mentioned in the previous experiment (*Experiment 3- BMv2 performance isolation with over-utilized link*), it is necessary to investigate the capacity margin required between the (configured) sum of the vQueues and the actual link capacity in order to ensure that all slices meet their performance target. To that end, we assume the behavior of a Markovian non-preemptive priority queuing system and estimate the physical link capacity required to support the delay and throughput targets set per slice under various slice priorities.

Priority scheduling effectively blocks the queues of lower-priority slices during the busy period of higher-priority slices. We can calculate the maximum real queuing delay (worst case scenario) for each slice based on the maximum busy period for each slice $s \in S$ with priority $i \in I$, where $|I|$ is the highest priority. The maximum busy period t_i^s for each slice s with priority i can be estimated using its burst limit b_i^s (in bytes) and the bottleneck's residual capacity r_i for priority i as follows:

$$t_i^s = \frac{b_i^s}{r_i}. \tag{6.1}$$

The residual capacity available to each slice s with priority i and rate limit L_i^s over the bottleneck with nominal capacity C is derived as follows:

$$r_i = C - \sum_{\forall j \in S} \sum_{k=i+1}^{|I|} L_k^j. \tag{6.2}$$

According to Eqs. (6.1) and (6.2), the worst-case queuing delay d_i for each slice s with priority i is equal to the aggregate busy period of all higher or equivalent priority slices, including slice s, as follows:

$$d_i = \sum_{\forall j \in S} \sum_{k=i}^{|I|} t_k^j. \tag{6.3}$$

Table 6.4.: Per slice priority.

Case \ Slice	DC	Enterprise	Peering
A	7	7	7
B	7	6	6
C	7	7	6
D	7	6	5

The extra worst-case queuing delay $\Delta d_i = d_i - d_{i+1}$ for all slices with priority i is determined by (i) the residual capacity of the link for that priority r_i and (ii) the sum of all burst sizes (derived from the delay burst limits and their respective rate limits) of all slices with that priority, as follows:

$$\Delta d_i = d_i - d_{i+1} = \frac{\sum_{\forall j \in S} b_i^j}{r_i}. \tag{6.4}$$

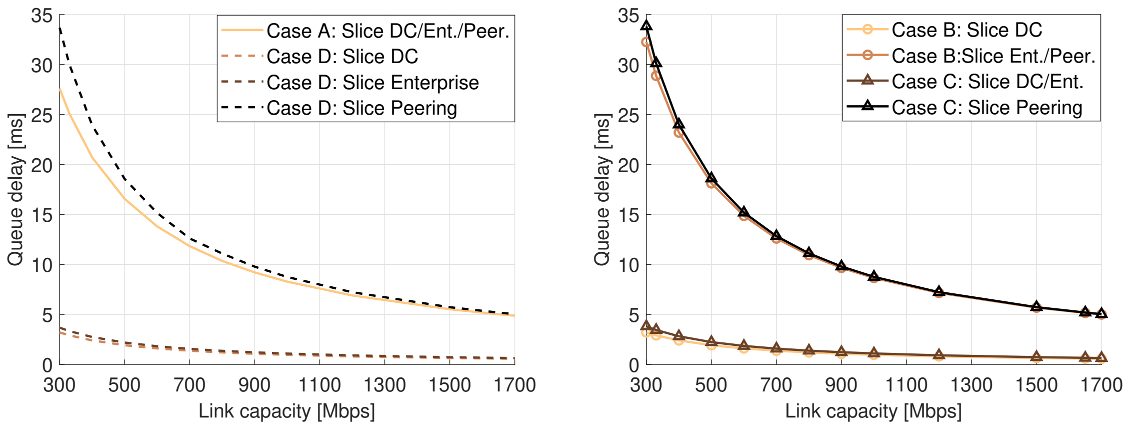
We can derive from this reworked equation the bound on the maximum extra latency Δd_i for slices of priority i given a residual capacity r_i. Alternatively, we can derive the bound on the minimum residual capacity r_i required for the given maximum extra latency Δd_i that is allowed for the slices of priority i.

When the real queuing latency cannot be controlled, these bounds restrict the number of slices that can be supported for a given link capacity. When the real queues, on the other hand, can be controlled, these bounds provide the worst-case rate that a slice with a given priority can experience.

We plot in Fig. 6.10 the worst-case queuing delay of the three slices used in the experiments above (i.e., slice DC, Enterprise, and Peering with target rates of 48 Mbps, 12 Mbps, and 240 Mbps, respectively) for various rates (bottleneck links), with priorities as shown in Table 6.4. The figure has been divided into two subplots to improve readability because cases B, C, and D partially overlap. Case A is similar to the initial slice setup used in the previous first two experiments for BMv2 and SmartNIC evaluations in *Experiment 1* and *Experiment 2*, while Case C is similar to the slice setup used in the third experiment, *Experiment 3*, for the over-utilized link evaluation. Case D yields results similar to Case C, where DC and Enterprise traffic are both prioritized over Peering traffic. Case B is defined to identify the corresponding limits When traffic from slice DC is prioritized over the other two slices.

The graph corresponding to *Case A: Slice DC/Ent./Peer*, in Fig. 6.10a shows the worst-case queueing delay for each slice when no traffic prioritization is applied. We find that a capacity of 1650 Mbps is required to ensure a (maximum) 5 ms delay for DC Slice.

In case B, the queuing delays for DC, Enterprise, and Peering slices are denoted as



(a) Cases A and D. (b) Cases B and C.

Figure 6.10.: Worst-case delay as a function of link capacity.

Case B:Slice DC and *Case B:Slice Ent./Peer*. From Fig. 6.10b, we observe that in this case, where DC traffic is prioritized over traffic from the other two slices, the Slice Enterprise's 10 ms delay limit is only met when the bottleneck rate exceeds 900 Mbps.

In case C, where traffic from DC and Enterprise slices is prioritized over Peering traffic, the worst-case queuing delays are denoted as *Case C:Slice DC/Ent.* and *Case C:Slice Peering*. Similarly, graphs *Case D: Slice DC*, *Case D: Slice Enterprise*, and *Case D: Slice Peering* show the worst-case queueing delay for each slice when different priorities are applied between the slices. The results of cases C and D are depicted in Figs. 6.10b and 6.10a, respectively. These results verify that both priority policies allow us to support the corresponding maximum burst limits for all slices, even in the worst-case scenario, as long as the bottleneck is at least 330 Mbps. This is in line with the evaluation results of *Experiment 3* analyzed before.

In general, we can observe that by using the corresponding maximum burst limits (worst-case queuing delay) of the slices, strict slice priority allows us to determine the necessary capacity margin required for the vQueues to operate efficiently under different conditions. For example, 330 Mbps link capacity is needed in cases C and D for the configured per slice QoS and rate targets. To meet their Key Performance Indicator (KPI)s, an intelligent control plane could assign slices to different priorities based on their requirements and physical link capacity.

6.4.3. Packet Processing Latency

We evaluate the efficiency of our proposed vQueue-based approach in the SmartNIC setup in terms of the pipeline's processing delay and memory consumption. We again use P4 meters as a baseline approach for the comparison. To determine the additional

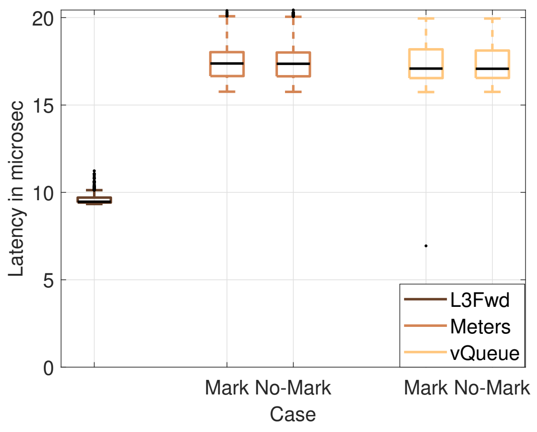


Figure 6.11.: Packet processing latency results.

processing latency required by meters and vQueue implementations, we measure and report the latency of a program that only performs L3Fwd functionality.

We use the same experimentation setup depicted in Fig. 3.2 and described in Subsection 3.2.1 earlier. In this setup, MoonGen [31] is used to benchmark the performance of Agilio CX SmartNIC. MoonGen is configured to send 1500 Byte UDP packets at 2.4 Gbps to the SmartNIC after loading the three P4 implementations under test (L3Fwd, vQueue, and Meters). The packets are processed in the card before being returned to MoonGen, where packet latency is reported.

Furthermore, we investigate the effect of different actions taken by the examined P4 implementations on packet processing latency. This is done by configuring each traffic management implementation to limit the rate at 2.3 Gbps to be smaller than the 2.4 Gbps generated traffic rate (i.e., apply Mark Action), and also at 5 Gbps to be greater than the incoming traffic rate (i.e., Apply No-Mark Action). We ensured that no packets were dropped by the traffic management mechanisms, as this would cause the packet latency measurements to be disrupted. This was accomplished by arbitrarily increasing the dropping burst size and minimizing the difference between the limited rate in the Mark-Action case, i.e., 2.3 Gbps, and the sent rate, i.e 2.4,Gbps, to ensure that the drop burst threshold is never reached and no packet is dropped.

The box plots of measured latency, in μs, for the various P4 implementations with different configurations are shown in Fig. 6.11. On top of the L3Fwd pipeline, the traffic management mechanisms in both vQueue and meter implementations contributed an additional 8 μs. Furthermore, our proposed vQueue implementation has a slight advantage over the meter implementation in terms of packet processing load, with the median of measured latency in the vQueue case being 0.3 μs less than that measured in the meter case. Furthermore, we can see that the packet processing latency of the two implementations is constant regardless of whether the packets are marked or not.

Memory usage in both implementations is almost the same, except for the use of Cluster Local Scratch (CLS) memory, which is responsible for storing frequently used data, whereas the vQueue implementation requires an additional 2% of the available memory compared to the meter-based implementation.

Note that an alternative vQueue implementation using one register and one atomic section did not improve the performance because accessing a register was faster than the additional processing required. Depending on the target and register constraints, it may be possible to further optimize the algorithm to achieve the optimal balance between resource usage and processing latency, but at the expense of the generalization of the implementation.

6.5. Summary

In this chapter, we implement and analyze two applications to demonstrate the benefits of adopting programmable data planes for future networks.

In the first application, we redesign the UPF of the 5G Core networks to follow the microservice-based architecture. This design enables more efficient deployments of the UPF into cloud environments to better leverage the benefits of this computing paradigm. A PoC implementation of this design using P4 is described, and the limitations in the P4 language and targets identified when conducting this exercise are discussed. Finally, the integration of this application into the P4-enhanced cloud environments is illustrated, where the optimal management scheme proposed in Chapter 5 can be used to optimize the orchestration of the microservice-based UPF.

The second contributed application is a programmable traffic management solution that enforces different configured QoS requirements in terms of rate and delay into different network slices. While this solution holds as a stand-alone solution for different use cases, it complements the UPF implementation, wherein QoS enforcement is among its designated tasks. The solution uses and implements vQueues for controlling not only the rate of different slices, a task achievable using the standard P4 meters, but also the delay. The programmable traffic management solution is implemented on software and SmartNIC-based P4 targets, where portability issues are discussed and mitigated. We validate the proposed approach's performance using elastic congestion-controlled traffic and investigate the relationship between the shared link and the required network slices' capacity for the proposed approach to operate efficiently using vQueues. In comparison to standard P4 meters, we demonstrate that our design has comparable rate-limiting performance, full access to state information, and a slightly lower processing delay.

7. Conclusion and Outlook

Emerging applications such as IoTs and CPSs have stringent connectivity requirements imposed on the underlying networks in terms of forwarding performance. On the other hand, the required connectivity services and functions by these applications are changing rapidly which demands the adoption of programmability in these networks to define their behavior flexibly. Different solutions such as SDN, NFV, and more recently P4 programmability have been proposed to fulfill these requirements. P4 programmability is regarded as a promising technology because it extends the flexibility provided by the SDN paradigm by enabling programmability at the data plane of packet processors, and it improves the performance of the NFV paradigm by enabling programmability on hardware accelerators.

To address the connectivity needs of emerging applications, the thesis proposes the integration of P4 programmable packet processors into the NFV cloud environment to enhance its processing performance without sacrificing the flexibility attained via programmability. However, this integration also raises a new plane of problems and challenges in terms of the design and management of such environments.

Given that the P4 language is target independent, this means that it can be used to program different types of packet processors such as CPUs, NPUs, FPGAs, and ASICs. Accordingly, the management plane should decide where to place the NF workload. To take the optimal placement decision, the management plane needs to know the limitations, capabilities, and performance of the different device types. However, this information is not fully explored in the literature. Accordingly, a comprehensive evaluation of the different P4 device types is needed to understand the performance of these devices.

Furthermore, keeping in mind that P4 devices enable data plane programmability, the processing latency, and thus the forwarding latency, on these devices may vary based on the complexity of the loaded packet processing pipeline. Hence, proper modeling of the performance of these devices is necessary to anticipate the performance of these devices when different influential factors are varied.

When the performance models of the different P4 devices are available, the optimal integration and deployment of P4 devices into NFV cloud environments become possible. Both the offline optimal planning for selecting and building the network's substrate and the optimal runtime management of an already built-up network are important management problems that need to be formulated and studied.

In the following, Section 7.1 summarizes the key contributions and outcomes of this thesis, and Section 7.2 reports on some interesting and challenging future research directions related to the deployment of programmable packet processors into cloud environments.

7.1. Summary

The thesis addresses the issues and challenges that arise when integrating and deploying P4 programmable packet processors into NFV cloud environments. The four major contributions in the thesis are presented in Chapters 3, 4, 5, and 6. Each contribution paves the way for the follow-up contribution until eventually the objective of the thesis is reached. The performance evaluation and measurements on P4 devices conducted in Chapter 3 provide the necessary realistic input regarding the forwarding latency on P4 data and control planes, which is needed for parametrizing the performance models presented in Chapter 4. The developed performance models in Chapter 4 enable the performance-aware optimal management of P4-enhanced cloud environments presented in Chapter 5. Finally, the applications presented in Chapter 6 demonstrate the usability of the proposed P4-enhanced NFV cloud environment, which can be optimally managed using the framework presented in Chapter 5. In the following, we summarize the individual major contributions of the thesis presented in Chapters 3-6.

Benchmarking the Performance of P4 Programmable Packet Processors. To understand the capabilities and limitations of different P4 device types, it is important to conduct a comprehensive evaluation of these devices. The first major contribution presents a performance study of the different components that build a P4-based system. Different important data plane performance metrics such as the average baseline forwarding latency, the processing latency of atomic P4 operations, the variation in forwarding latency due to scaling up the number of incoming distinct flows, and the rule insertion time were evaluated. Moreover, an evaluation of the performance of P4RT-based controller is conducted using a novel tool developed for this purpose. The measurements revealed the capabilities and limitations of different investigated P4 devices towards advancing the common knowledge on the performance of this class of devices.

Modeling the Performance of P4 Programmable Packet Processors. Different performance models for P4-based systems were proposed using the measurements collected in the previous contribution. These models realize the various factors that can affect the system performance and include them as parameters that can be tuned based on the tested scenario. The modeling exercise is divided into two parts. In the first stage, we concentrate on modeling the challenging part, which is the relationship between the data plane's forwarding latency and the complexity of the loaded P4 program. In this direction, we use the previous contribution's measurements to develop a generic method for predicting packet forwarding latency when running arbitrary P4 programs on different P4 packet processors. The proposed method is validated by running three realistic network functions on three P4 targets, with the recorded estimation accuracy always exceeding 95%.

The second stage builds on the first stage's model to propose two queueing theory-based models that abstract the performance of complete P4-based systems, including the control plane, as feedback-oriented queues with all the factors that influence the system's performance included. To keep the model simple, the first model assumes exponentially distributed service times, whereas the second model relaxes this assumption and uses generic service processes whose distribution is based on measurements to provide more realistic performance predictions. Finally, the two models are evaluated and validated through simulations that vary a wide range of parameters and analyze their impact on the packet's sojourn time in the system. When any of the influential parameters are varied, the results of the two models are very similar, with the exception that the second (refined) model has lower sojourn time values than the model with exponentially-distributed service times. The accuracy of the two models is found to be very high when compared to simulation results. This evaluation resulted in the development of constraints for dimensioning the permissible input traffic that the system can handle without packet drops.

Performance-Aware Management of P4-based Cloud Environments. The previous contribution's performance models are used to formulate and evaluate two optimization problems related to managing P4-enhanced NFV environments. The first problem called PA-P4VNF-RA seeks to optimize the infrastructure substrate of P4-enhanced NFV environments. The optimization problem seeks the optimal set of P4 packet processors capable of handling a given processing workload, as well as the optimal placement solution for this workload into the chosen hosting devices. When selecting the optimal set of P4 packet processors to handle a given processing workload, the multi-objective function aims to maximize system performance while minimizing capital expenditure costs. The problem constrains the solution space to satisfy the NF workload requirements while recognizing the distinct capabilities of the various candidates hosting P4 devices. The second optimization problem is called PA-P4SFC-E, and it aims to find the best way to embed SFCs into P4-enhanced NFV environments at runtime. The optimization problem seeks the best placement and routing of SFCs on a P4 programmable substrate. Furthermore, a greedy solution is designed and implemented to solve the PA-P4SFC-E problem faster at runtime. The performance models developed in the previous contribution allow for the calculation of the forwarding delay resulting from various possible placement options, guiding the search in the two problems towards the most performance-efficient solution that meet functional and QoS requirements.

A detailed evaluation of the two problems is carried out after populating the model's parameters using results from previous contributions and surveyed literature works. When analyzing the PA-P4VNF-RA problem, the trade-off between cost and performance objective functions is highlighted. When evaluating the PA-P4SFC-E problem, the impact of various system parameters such as SFC length and the degree of adoption of IPUs and programmable ASICs is considered. Finally, the greedy solution's evaluation revealed its effectiveness in handling scaled-up scenarios in terms of execution time, at the expense of reduced optimality.

P4 Applications: Use Case Studies. Two applications are implemented and analyzed to show the advantages of using programmable data planes in cloud environments for future networks. The UPF of the 5G Core networks is redesigned in the first application to follow the microservice-based architecture. This design allows for more efficient deployments of the UPF into cloud environments, allowing for greater utilization of the benefits of this computing paradigm. A PoC implementation of this design in P4 is described, as are the limitations in the P4 language and targets identified during this exercise. Finally, we show how to integrate this application into P4-enhanced cloud environments, where the optimal management scheme proposed in the previous contribution can be used to optimize the orchestration of the microservice-based UPF.

The second contributed application is a programmable traffic management solution that enforces different configured QoS requirements in terms of rate and delay across different network slices. While this solution can be used as a stand-alone solution for various use cases, it also complements the UPF implementation by enabling QoS enforcement on a per-flow basis. The solution employs and implements vQueues to control not only the rate of different slices, which can be accomplished with standard P4 meters, but also the delay. The programmable traffic management solution is deployed on software and SmartNIC-based P4 targets, and portability issues are discussed and addressed. We validate the proposed approach's performance using elastic congestion-controlled traffic and investigate the relationship between the shared link and network slices' required capacity for the proposed approach to operate efficiently using vQueues. We show that our design has comparable rate-limiting performance, full access to state information, and a slightly lower processing delay than standard P4 meters.

7.2. Future Work

We believe that the following research topics are interesting for future work.

Evaluating the power consumption of programmable packet processors. As the interest in building sustainable networks is increasing, evaluating the energy consumption of different programmable packet processors becomes more important. In this thesis, we benchmarked the forwarding latency of different P4 programmable packet processors and then we proposed a method for estimating this forwarding latency as a function of the complexity of the loaded data plane pipeline. A similar approach can be followed to derive models that can relate the power consumption of P4 devices to the required processing workload. Furthermore, the optimization problems proposed in Chapter 5 can be extended by incorporating the developed power models to enable management with awareness related to performance and power consumption.

Worst-case delay modeling for programmable packet processors. While the performance models derived in this thesis focus on the average delay as the metric of interest, some user applications such as those that include control loops for safety-critical applications require guarantees related to the worst-case delay in the network. Deriving such models is possible by using modeling techniques such as network calculus instead of the ones used in this thesis which is based on stochastic queueing theory.

The coexistence of different programmable data plane technologies in cloud environments. While this thesis considers P4 programmable packet processors and the deployment of this class of devices into cloud environments, it is worth investigating other technologies that also enable programmability at the data plane such as eBPF [105], PoF [17], etc. More interestingly is studying the interoperability issues that could arise when these different technologies coexist in the cloud environment. In this case, the development of a generic abstraction layer for accessing these different acceleration techniques is of paramount importance.

Deploying microservice-based RAN into P4-enhanced cloud environment. As cloud-native solutions are becoming more prominent in cellular networks, the study that focuses on the cloudification of the RAN becomes more interesting. Finding the best design that separates the different functions of the RAN, while keeping the dependency between these functions minimal is a challenging task. While this thesis considered the cloudification of the UPF and discussed the deployment of the new design into P4-enhanced cloud environments, applying a similar approach to the RAN is even more important, especially since the processing of RAN functions heavily depends on the usage of hardware accelerators.

List of Figures

List of Tables

A. Abbreviations

NIC	Network Interface Card
eBPF	extended Berkeley Packet Filter
PoF	Protocol-oblivious Forwarding
P4	Programming Protocol-Independent Packet Processors
L3Fwd	Layer 3 Forwarding
L2Fwd	Layer 2 Forwarding
FPGA	Field-Programmable Gate Array
ASIC	Application-Specific Integrated Circuit
SDN	Software-defined networking
NFV	Network Functions Virtualization
LPM	Longest Prefix Match
MAU	Match-Action Unit
API	Application Programming Interface
P4RT	P4Runtime
gRPC	Google Remote Procedure Call
P4C	P4 Compiler
JSON	JavaScript Object Notation
BMv2	Behavioral Model
OvS	Open vSwitch
NPU	Network Processor Unit
RTL	Register-Transfer Level

LUTs LookUp Tables

TCAM Ternary Content Addressable Memory

SRAM Static Random Access Memory

RAM Random Access Memory

PISA Protocol-Independent Switch Architecture

DPDK Data Plane Development Kit

CPU Central Processing Unit

NUMA Non-Uniform Memory Access

OF OpenFlow

IT Information Technology

NF Network Function

NFV Network Function Virtualization

OFCProbe OpenFlow Controller Probe

P4RCProbe P4Runtime Controller Probe

MTU Maximum Transmission Unit

PTP Precision Time Protocol

FSM Finite State Machine

ONOS Open Network Operating System

PCIe Peripheral Component Interconnect Express

RTT Round-trip Time

TCP Transmission Control Protocol

SYN Synchronize

RPC Remote Procedure Call

B5G Beyond 5G

AQM Active Queue Management

VNF Virtualized Network Function

UPF User Plane Function

SFC Service Function Chain

QoS Quality of Service

COTS Commercial off-the-shelf

IT Information Technology

IoT Internet of Things

NP Nondeterministic Polynomial Time

CFG Control Flow Graph

IR Intermediate Representation

MAC Media Access Control

TTL Time to Live

VxLAN Virtual Extensible LAN

VNI VxLAN Network Identifier

PASTA Poisson Arrivals See Time Averages

OFLOPS OpenFlow Operations Per Second

PA-P4VNF-RA Performance-Aware P4 Virtual Network Functions Resource Allocation

PA-P4SFC-E Performance-Aware P4 Service Function Chain Embedding

ToR Top of Rack

CAPEX Capital expenditures

VNF-OP VNF Orchestration Problem

VNF-AAPC Accelerator-aware VNF Placement and Chaining

FW Firewall

NAT Network Address Translation

VDecap VxLAN Decapsulation

LB Load Balancer

V1M V1Model

SUME SimpleSumeSwitch

ILP Integer Linear Programming

IPU Infrastructure Processing Unit

AIC Akaike Information Criterion

RMSE Root Mean Square Error

CDF Cumulative Density Function

NP Nondeterministic Polynomial Time

vQueue Virtual Queue

AQM Active Queue Management

ECN Explicit Congestion Notification

SLA Service Level Agreement

trTCM two rate Three Color Marker

CIR Committed Information Rate

PIR Peak Information Rate

DDR4 Double Data Rate Fourth Generation

WAN Wide Area Network

TD Tail Drop

DCTCP Data Center TCP

DC Data Center

KPI Key Performance Indicator

CLS Cluster Local Scratch

UE User Equipment

RAN Radio Access Network

UPF User Plane Function

DN Data Network

AMF Access and Mobility Management Function

AUSF Authentication Server Function

SMF Session Management Function

NSSF Network Slice Selection Function

NRF NF Repository Function

PCF Policy Control function

UDM Unified Data Management

gNB Next Generation NodeB

SBA Service-Based Architecture

NEF Network Exposure Function

PDU Protocol Data Unit

3GPP 3rd Generation Partnership Project

PIFO Push-In-First-Out

FIFO First-In-First-Out

SP-PIFO Strict-Priority-Push-In-First-Out

PIEO Push-In-Extract-Out

LSTF Least Slack Time First

CoDel Controlled Delay

RED Random Early Drop

PIE Proportional Integral controller Enhanced

vEPG virtual Evolved Packet Gateway

PoC Proof of Concept

ISF	Ingress Steering Function
ULF	Uplink Function
DLF	Downlink Function
ODF	On-Demand Function
REST	Representational State Transfer
LIF	Lawful Intercept Function
SOTA	state-of-the-art
PDR	Packet Detection Rule
FAR	Forward Action Rule
QFI	QoS Flow Identifier
TPD	Thermal Design Power
CPS	Cyber-Physical Systems

Bibliography

Publications by the author

[1] H. Harkous, M. Jarschel, M. He, R. Pries, and W. Kellerer, "P8: P4 with predictable packet processing performance," *IEEE Transactions on Network and Service Management*, vol. 18, no. 3, pp. 2846–2859, 2021. 7, 11, 23, 27, 66, 68, 72, 78, 178

[2] H. Harkous, M. Jarschel, M. He, R. Priest, and W. Kellerer, "Towards understanding the performance of P4 programmable hardware," in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, 2019, pp. 1–6. 7, 11, 23, 27, 66, 178

[3] H. Harkous, M. He, M. Jarschel, R. Pries, E. Mansour, and W. Kellerer, "Performance study of P4 programmable devices: Flow scalability and rule update responsiveness," in *2021 IFIP Networking Conference (IFIP Networking)*, 2021, pp. 1–6. 7, 24, 42, 47, 112

[4] M. Helm, H. Stubbe, D. Scholz, B. Jaeger, S. Gallenmüller, N. Deric, E. Goshi, H. Harkous, Z. Zhou, W. Kellerer, and G. Carle, "Application of network calculus models on programmable device behavior," in *2021 33rd International Teletraffic Congress (ITC 33)*, Avignon, France, 2021. 7, 71

[5] H. Harkous, N. Kröger, M. Jarschel, R. Pries, and W. Keller, "Modeling and performance analysis of p4 programmable devices," in *2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2021, pp. 67–73. 7, 68

[6] H. Harkous, K. Sherkawi, M. Jarschel, R. Pries, M. He, and W. Kellerer, "P4rcprobe for evaluating the performance of p4runtime-based controllers," in *2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2021, pp. 74–80. 7, 11, 24, 53

[7] H. Harkous, B. A. Hosn, M. He, M. Jarschel, R. Pries, and W. Kellerer, "Towards performance-aware management of p4-based cloud environments," in *2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2021, pp. 87–90. 7, 11, 97, 103

[8] H. Harkous, C. Papagianni, K. De Schepper, M. Jarschel, M. Dimolianis, and

R. Pries, "Virtual queues for p4: A poor man's programmable traffic manager," *IEEE Transactions on Network and Service Management*, vol. 18, no. 3, pp. 2860–2872, 2021. 7, 11, 12, 138, 150

[9] D. Scholz, H. Harkous, S. Gallenmüller, H. Stubbe, M. Helm, B. Jaeger, N. Deric, E. Goshi, Z. Zhou, W. Kellerer, and G. Carle, "A framework for reproducible data plane performance modeling," in *Proceedings of the Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 59–65. [Online]. Available: https://doi.org/10.1145/3493425.3502756 7

[10] N. Kröger, H. Harkous, F. Mehmeti, and W. Kellerer, "Looking beyond the first moment: Analysis of packet-related distributions in p4 systems with controller feedback," in *International Teletraffic Conference (ITC) 34*, Shenzhen, China, 2022, p. 9. 7

[11] N. Kröger, F. Mehmeti, H. Harkous, and W. Kellerer, "Performance analysis of general p4 forwarding devices with controller feedback," in *Proceedings of the 25th International ACM Conference on Modeling Analysis and Simulation of Wireless and Mobile Systems*, ser. MSWiM '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 55–64. [Online]. Available: https://doi.org/10.1145/3551659.3559045 7

General publications

[12] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford, "Pisces: A programmable, protocol-independent software switch," ser. SIGCOMM '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 525–538. 19

[13] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon, "P4fpga: A rapid prototyping framework for p4," in *Proceedings of the Symposium on SDN Research (SOSR)*. ACM, 2017, pp. 122–135. 24

[14] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman, "The p4 $\rightarrow$ netfpga workflow for line-rate packet processing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 1–9. 12, 20

[15] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "Netfpga sume: Toward 100 gbps as research commodity," *IEEE micro*, vol. 34, no. 5, pp. 32–41, 2014. 20, 155

[16] P. Vörös, D. Horpácsi, R. Kitlei, D. Leskó, M. Tejfel, and S. Laki, "T4p4s: A

target-independent compiler for protocol-independent packet processors," in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 2018, pp. 1–8. 20

[17] H. Song, "Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 127–132. [Online]. Available: https://doi.org/10.1145/2491185.2491190 11, 173

[18] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014. 2, 11, 12

[19] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015. 13

[20] M. Jarschel, C. Metter, T. Zinner, S. Gebert, and Phuoc Tran-Gia, "OFCProbe: A platform-independent tool for OpenFlow controller analysis," in *2014 IEEE Fifth International Conference on Communications and Electronics (ICCE)*. IEEE, 2014, pp. 182–187. 26, 53

[21] M. Jarschel, F. Lehrieder, Z. Magyari, and R. Pries, "A flexible OpenFlow-controller benchmark," in *2012 European Workshop on Software Defined Networking*, 2012, pp. 48–53. 25

[22] A. Nguyen-Ngoc, S. Raffeck, S. Lange, S. Geissler, T. Zinner, and P. Tran-Gia, "Benchmarking the ONOS controller with OFCProbe," in *2018 IEEE Seventh International Conference on Communications and Electronics (ICCE)*. IEEE, 2018, pp. 367–372. 26

[23] Z. K. Khattak, M. Awais, and A. Iqbal, "Performance evaluation of OpenDaylight SDN controller," in *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2014, pp. 671–676. 26

[24] O. Salman, I. H. Elhajj, A. Kayssi, and A. Chehab, "SDN controllers: A comparative study," in *2016 18th Mediterranean Electrotechnical Conference (MELECON)*, 2016, pp. 1–6. 26

[25] A. Shalimov, D. Zuikov, D. Zimarina, V. Pashkov, and R. Smeliansky, "Advanced study of SDN/OpenFlow controllers," in *Proceedings of the 9th Central Eastern European Software Engineering Conference in Russia*. ACM, 2013. 26

[26] F. Alencar, M. Santos, M. Santana, and S. Fernandes, "How software aging affects

SDN: A view on the controllers," in *2014 Global Information Infrastructure and Networking Symposium (GIIS)*, 2014, pp. 1–6. 26

[27] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in Software-Defined Networks," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*. USENIX Association, 2012, p. 10. 26

[28] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow *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. 17

[29] 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. 33, 70, 79

[30] H. T. Dang, H. Wang, T. Jepsen, G. Brebner, C. Kim, J. Rexford, R. Soulé, and H. Weatherspoon, "Whippersnapper: A p4 language benchmark suite," in *Proceedings of the Symposium on SDN Research*, 2017, pp. 95–101. 24, 65, 66, 71, 178

[31] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," in *Proceedings of the Internet Measurement Conference (IMC)*. ACM, 2015, pp. 275–287. 28, 30, 31, 43, 77, 167

[32] D. Scholz, H. Stubbe, S. Gallenmüller, and G. Carle, "Key Properties of Programmable Data Plane Targets," in *Teletraffic Congress (ITC 32), 2020 32nd International*, Osaka, Japan, 2020. 24, 65, 66, 71, 112, 178

[33] A. Mohammadkhan, S. Panda, S. G. Kulkarni, K. K. Ramakrishnan, and L. N. Bhuyan, "P4NFV: P4 Enabled NFV Systems with SmartNICs," in *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, 2019, pp. 1–7. 24, 102, 103

[34] 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. 70

[35] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu, "Stateless datacenter load-balancing with beamer," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, pp. 125–139. 24

[36] G. Grigoryan, Y. Liu, and M. Kwon, "iload: In-network load balancing with programmable data plane," in *Proceedings of the 15th International Conference on emerging Networking EXperiments and Technologies*, 2019, pp. 17–19. 24

[37] T. Kohler, R. Mayer, F. Dürr, M. Maaß, S. Bhowmik, and K. Rothermel, "P4cep: Towards in-network complex event processing," in *Proceedings of the 2018 Morning Workshop on In-Network Computing*, 2018, pp. 33–38. 24

[38] P. Bressana, N. Zilberman, D. Vucinic, and R. Soule, "Trading latency for compute in the network," in *ACM SIGCOMM 2020 Workshop on Network Application Integration/CoDesign (NAI)*. ACM, 2020, pp. 1–6. 24

[39] Y. Qiao, X. Kong, M. Zhang, Y. Zhou, M. Xu, and J. Bi, "Towards in-network acceleration of erasure coding," in *Proceedings of the Symposium on SDN Research (SOSR)*, 2020, pp. 41–47. 24

[40] D. Scholz, A. Oeldemann, F. Geyer, S. Gallenmüller, H. Stubbe, T. Wild, A. Herkersdorf, and G. Carle, "Cryptographic hashing in p4 data planes," in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, 2019, pp. 1–6. 24, 112

[41] P. B. Viegas, A. G. de Castro, A. Lorenzon, F. D. Rossi, and M. C. Luizelli, "The actual cost of programmable smartnics: Diving into the existing limits," in *Advanced Information Networking and Applications (AINA)*, vol. 225. Springer, 2021, pp. 1–6. 24

[42] D. Erickson, "The beacon openflow controller," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 13–18. 26

[43] P. Benáček, V. Puš, H. Kubátová, and T. Čejka, "P4-to-vhdl: Automatic generation of high-speed input and output network blocks," *Microprocessors and Microsystems*, vol. 56, pp. 22–33, 2018. 34

[44] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, 2013. 32

[45] L. Linguaglossa, S. Lange, S. Pontarelli, G. Rétvári, D. Rossi, T. Zinner, R. Bifulco, M. Jarschel, and G. Bianchi, "Survey of performance acceleration techniques for network function virtualization," *Proceedings of the IEEE*, vol. 107, no. 4, pp. 746–764, 2019. 2, 14

[46] M. He, A. Basta, A. Blenk, N. Deric, and W. Kellerer, "P4NFV: An NFV architecture with flexible data plane Reconfiguration," in *Proceedings of the International Conference on Network and Service Management (CNSM)*. IEEE, 2018, pp. 90–98. 2, 14

[47] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, and M. Yu, "Hpcc: High precision congestion control," in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '19. Association for Computing Machinery, 2019, p. 44–58. [Online]. Available: https://doi.org/10.1145/3341302.3342085 12

[48] J. Hyun, N. Van Tu, and J. W.-K. Hong, "Towards knowledge-defined networking using in-band network telemetry," in *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, 2018, pp. 1–7. 12

[49] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '16. Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2890955.2890968 12

[50] S. K. Singh, C. E. Rothenberg, G. Patra, and G. Pongracz, "Offloading virtual evolved packet gateway user plane functions to a programmable asic," in *Proceedings of the 1st ACM CoNEXT Workshop on Emerging In-Network Computing Paradigms*, ser. ENCP '19. Association for Computing Machinery, 2019, p. 9–14. [Online]. Available: https://doi.org/10.1145/3359993.3366645 12

[51] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H. Acharya, K. Zarifis, and S. Shenker, "Troubleshooting blackbox sdn control software with minimal causal sequences," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, p. 395–406, Aug. 2014. 25

[52] G. Antichi and G. Rétvári, "Full-stack sdn: The next big challenge?" in *Proceedings of the Symposium on SDN Research*, ser. SOSR '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 48–54. [Online]. Available: https://doi.org/10.1145/3373360.3380834 1

[53] O. Arouk and N. Nikaein, "5g cloud-native: Network management amp; automation," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, 2020, pp. 1–2. 3

[54] M. Harchol-Balter, *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013. 79, 81

[55] K. Mahmood, A. Chilwan, O. N. Sterb, and M. Jarschel, "On the modeling of OpenFlow-based SDNs: The single node case," *Computer Science & Information Technology*, 2014. 70, 81

[56] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," vol. 38, no. 2. Association for Computing Machinery, 2008, p. 69–74. 87

[57] F. Wamser, R. Pries, D. Staehle, K. Heck, and P. Tran-Gia, "Traffic characterization of a residential wireless internet access," *Special Issue of the Telecommunication Systems (TS) Journal*, vol. 48: 1-2, 2010. 87

[58] Y. Goto, H. Masuyama, B. Ng, W. K. G. Seah, and Y. Takahashi, "Queueing analysis of Software Defined Network with realistic OpenFlow–based switch model," in *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2016, pp. 301–306. 70

[59] K. Mahmood, A. Chilwan, O. Østerbø, and M. Jarschel, "Modelling of OpenFlow-based software-defined networks: the multiple node case," *IET Networks*, vol. 4, no. 5, pp. 278–284, 2015. 70

[60] J. Ansell, W. K. G. Seah, B. Ng, and S. Marshall, "Making queueing theory more palatable to SDN/OpenFlow-based network practitioners," in *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, 2016, pp. 1119–1124. 70

[61] D. Lukács, G. Pongrácz, and M. Tejfel, "Performance guarantees for P4 through cost analysis," in *2019 IEEE 15th International Scientific Conference on Informatics*. IEEE, 2019, pp. 000 305–000 310. 71

[62] S. Wang, Z. Meng, C. Sun, M. Wang, M. Xu, J. Bi, T. Yang, Q. Huang, and H. Hu, "SmartChain: Enabling high-performance service chain partition between SmartNIC and CPU," in *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, 2020, pp. 1–7. 102

[63] S. Nik, S. John, G. Hans, P. Isaac, and H. Zhaoyang, "Flightplan: Dataplane disaggregation and placement for P4 programs," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Apr. 2021. 102, 103

[64] G. P. Sharma, W. Tavernier, D. Colle, and M. Pickavet, "VNF-AAPC: Accelerator-aware VNF Placement and Chaining," Mar. 2020, working paper or preprint. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02517141 102, 103

[65] F. Bari, S. R. Chowdhury, R. Ahmed, R. Boutaba, and O. C. M. B. Duarte, "Orchestrating virtualized network functions," *IEEE Transactions on Network and Service Management*, vol. 13, no. 4, pp. 725–739, 2016. 102, 103

[66] W.-H. Chen, X. Yin, Z. Wang, and X. Shi, "Placement and routing optimization problem for service function chain: State of art and future opportunities," *ArXiv*, vol. abs/1910.02613, 2019. 102

[67] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys Tutorials*, vol. 18, no. 1, pp. 236–262, 2016. 102

[68] P. Bressana, N. Zilberman, D. Vucinic, and R. Soulé, "Trading latency for compute in the network," in *Proceedings of the Workshop on Network Application Integration/CoDesign*, ser. NAI '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 35–40. [Online]. Available: https://doi.org/10.1145/3405672.3405807 112

[69] L. Kleinrock, "Queueing systems: Volume i-theory," 1975. 83, 84

[70] W. S. LIU, H. Li, O. Huang, M. Boucadair, N. Leymann, Z. Cao, Q. Sun, C. Pham, C. Huang, J. Zhu, and P. He, "Service Function Chaining (SFC) Use Cases," Internet Engineering Task Force, Internet-Draft draft-liu-sfc-use-cases-06, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-liu-sfc-use-cases-06 134

[71] W. Haeffner, J. Napper, M. Stiemerling, D. Lopez, and J. Uttaro, "Service Function Chaining Use Cases in Mobile Networks," Internet Engineering Task Force, Internet-Draft draft-ietf-sfc-use-case-mobility-09, Jan. 2019, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-sfc-use-case-mobility-09 134

[72] D. A. Joseph, A. Tavakoli, and I. Stoica, "A policy-aware switching layer for data centers," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, p. 51–62, aug 2008. [Online]. Available: https://doi.org/10.1145/1402946.1402966 134

[73] B. Briscoe, "The native AQM for L4S traffic," *arXiv preprint arXiv:1904.07079*, 2019. 151

[74] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," *ACM SIGCOMM computer communication review*, vol. 41, no. 4, pp. 63–74, 2011. 152, 158

[75] Y.-W. Chen, L.-H. Yen, W.-C. Wang, C.-A. Chuang, Y.-S. Liu, and C.-C. Tseng, "P4-enabled bandwidth management," in *2019 20th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE, 2019, pp. 1–5. 142, 153, 156

[76] A. Sivaraman, K. Winstein, S. Subramanian, and H. Balakrishnan, "No silver bullet: extending SDN to the data plane," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in networks*. ACM, 2013, p. 19. 142

[77] N. K. Sharma, M. Liu, K. Atreya, and A. Krishnamurthy, "Approximating Fair Queueing on Reconfigurable Switches," in *USENIX Symposium on Networked Systems Design and Implementation*, 2018, pp. 1–16. 142

[78] N. K. Sharma, C. Zhao, M. Liu, P. G. Kannan, C. Kim, A. Krishnamurthy, and A. Sivaraman, "Programmable calendar queues for high-speed packet scheduling," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 685–699. 142

[79] C. Cascone, N. Bonelli, L. Bianchi, A. Capone, and B. Sansò, "Towards approximate fair bandwidth sharing via dynamic priority queuing," in *Local and Metropolitan Area Networks (LANMAN), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 1–6. 142

[80] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown, "Programmable packet scheduling at line rate," in *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 2016, pp. 44–57. 142

[81] A. G. Alcoz, A. Dietmüller, and L. Vanbever, "SP-PIFO: approximating push-in first-out behaviors using strict-priority queues," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 59–76. 142

[82] V. Shrivastav, "Fast, scalable, and programmable packet scheduler in hardware," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 367–379. 142

[83] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker, "Universal packet scheduling," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016, pp. 501–521. 142

[84] R. Kundel, J. Blendin, T. Viernickel, B. Koldehofe, and R. Steinmetz, "P4-CoDel: Active Queue Management in Programmable Data Planes," in *Proceedings of the IEEE 2018 Conference on Network Functions Virtualization and Sofwtare Defined Networks*. IEEE, 2018, pp. 27–29. 142

[85] C. Papagianni and K. De Schepper, "PI2 for P4: An active queue management scheme for programmable data planes," in *Proceedings of the 15th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 84–86. [Online]. Available: https://doi.org/10.1145/3360468.3368189 142

[86] M. Menth, H. Mostafaei, D. Merling, and M. Häberle, "Implementation and Evaluation of Activity-Based Congestion Management Using P4 (P4-ABC)," *Future Internet*, vol. 11, no. 7, p. 159, 2019. 142

[87] S. Laki, P. Vörös, and F. Fejes, "Towards an AQM Evaluation Testbed with P4 and DPDK," in *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, ser. SIGCOMM Posters and Demos '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 148–150. [Online]. Available:

https://doi.org/10.1145/3342280.3342340 142

[88] R. Shah, V. Kumar, M. Vutukuru, and P. Kulkarni, “Turboepc: Leveraging dataplane programmability to accelerate the mobile packet core,” in *Proceedings of the Symposium on SDN Research*, ser. SOSR ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 83–95. [Online]. Available: https://doi.org/10.1145/3373360.3380839 140

[89] S. K. Singh, C. E. Rothenberg, G. Patra, and G. Pongracz, “Offloading virtual evolved packet gateway user plane functions to a programmable asic,” ser. ENCP ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 9–14. [Online]. Available: https://doi.org/10.1145/3359993.3366645 141

[90] Y.-B. Lin, C.-C. Tseng, and M.-H. Wang, “Effects of transport network slicing on 5g applications,” *Future Internet*, vol. 13, no. 3, 2021. [Online]. Available: https://www.mdpi.com/1999-5903/13/3/69 141

[91] R. MacDavid, C. Cascone, P. Lin, B. Padmanabhan, A. ThakuR, L. Peterson, J. Rexford, and O. Sunay, *A P4-Based 5G User Plane Function*. New York, NY, USA: Association for Computing Machinery, 2021, p. 162–168. [Online]. Available: https://doi.org/10.1145/3482898.3483358 141

Cited websites

[92] “P4Runtime Specification v1.2.0,” https://opennetworking.org/wp-content/uploads/2020/10/P4Runtime-Specification-120-wd.html, accessed: 2022-01-11. 6, 16, 83

[93] “Protocol Buffers,” https://github.com/protocolbuffers/protobuf, accessed: 2022-01-11. 16

[94] “P4runtime.proto,” https://github.com/p4lang/p4runtime/blob/main/proto/p4/v1/p4runtime.proto, accessed: 2022-01-11. 16, 54

[95] “gRPC,” https://grpc.io/, accessed: 2022-01-11. 16

[96] “P4C,” https://github.com/p4lang/p4c, accessed: 2022-01-11. 18

[97] “P4lang Behavioral Model (bmv2),” last accessed: 2022-01-11. [Online]. Available: https://github.com/p4lang/behavioral-model 19

[98] “Mininet,” last accessed: 2022-01-11. [Online]. Available: http://mininet.org/ 19

[99] Xilinix. Virtex-7 FPGAs. https://www.xilinx.com/products/silicon-devices/

fpga/virtex-7.html. Accessed: 2022-01-11. 28

[100] Netronome. Mapping P4 to SmartNICs. https://p4.org/assets/p4_d2_2017_nfp_architecture.pdf. Accessed: 2022-01-11. 20

[101] ——. Netronome smartnic. https://www.netronome.com/products/agilio-cx/. Accessed: 2022-01-11. 20, 28, 44

[102] Intel. Intel Programmable Ethernet Switch. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html. Accessed: 2022-01-11. 20

[103] D. Project. DPDK (Data Plane Development Kit). https://www.dpdk.org/. Accessed: 2022-01-11. 2, 14, 19, 28, 40, 41

[104] Moses. spider_plot. https://github.com/NewGuy012/spider_plot/releases/tag/17.8. Accessed: June 2, 2022. 113

[105] "eBPF (extended Berkeley Packet Filter)," https://ebpf.io/, accessed: 2022-01-11. 11, 173

[106] "OpenFlow Switch Specification v1.5.1," https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf, accessed: 2022-01-11. 13

[107] "What is NFV," https://www.etsi.org/images/files/etsitechnologyleaflets/networkfunctionsvirtualization.pdf, accessed: 2022-01-11. 13

[108] Floodlight SDN Controller. https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/overview. Accessed: 2022-01-11. 25

[109] Onos SDN Controller. https://opennetworking.org/onos/. Accessed: 2022-01-11. 8, 26, 57

[110] NOX SDN Controller. https://github.com/noxrepo/nox. Accessed: 2022-01-11. 25

[111] Maestro: A System for Scalable OpenFlow Control, author=Cai, Zheng, Cox, Alan L. and Ng, T. S. Eugene howpublished = https://hdl.handle.net/1911/96391, year=2010, note = Accessed: 2022-01-11. 25

[112] Cbench: an OpenFlow Controller Benchmarker. https://github.com/trema/cbench. Accessed: 2022-01-11. 25

[113] OpenDayLight SDN Controller. https://www.opendaylight.org/. Accessed: 2022-01-11. 26

[114] P. Vörös, D. Horpácsi, R. Kitlei, D. Leskó, M. Tejfel, and S. Laki. (2017) t4p4s repository. https://github.com/P4ELTE/t4p4s. Accessed: 2022-01-11. 34, 45

[115] (2017) Simple Sume Switch Architecture. https://github.com/NetFPGA/P4-NetFPGA-public/wiki/Workflow-Overview. Accessed: 2022-01-11. 44

[116] TCPDUMP Public Repository. https://www.tcpdump.org. Accessed: 2022-01-11. 49

[117] OpenFlow Switch Specification. https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf. Accessed: 2022-01-11. 13

[118] P4 V1model Architecture. https://github.com/p4lang/p4c/blob/main/p4include/v1model.p4. Accessed: 2022-01-11. 14, 15, 155, 174

[119] *MATLAB version (R2020b)*, The Mathworks, Inc., 2020. 82, 87

[120] Infinera, "Low Latency – How Low Can You Go?" Tech. Rep., 2020. 96

[121] "NetFPGA-SUME™ Reference Manual," https://digilent.com/reference/_media/sume:netfpga-sume_rm.pdf, 2016. [Online]. Available: https://digilent.com/reference/_media/sume:netfpga-sume_rm.pdf 113

[122] "NFE-3240 Appliance Adapters," https://www.netronome.com/media/redactor_files/PB_NFE-3240_App_Adapter.pdf, 2016. [Online]. Available: https://www.netronome.com/media/redactor_files/PB_NFE-3240_App_Adapter.pdf 112

[123] "Intel® Xeon® Platinum 8156 Processor," https://ark.intel.com/content/www/us/en/ark/products/120499/intel-xeon-platinum-8156-processor-16-5m-cache-3-60-ghz.html. 112

[124] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2021. [Online]. Available: https://www.gurobi.com 114, 126

[125] J. Heinanen and R. Guerin. (1999) RFC2698: A two rate three color marker. 154

[126] Netronome. (2020) Traffic class configuration. https://groups.google.com/g/open-nfp/c/kNqO8mSTupE. Accessed: 2020-08-01. 155

[127] 3GPP, "5G; System architecture for the 5G System (5GS) (3GPP TS 23.501 version 16.6.0 Release 16)," ETSI 3rd Generation Partnership Project (3GPP), Technical Specification (TS) 123.501, 10 2020, version 16.6.0. [Online]. Available: https://www.etsi.org/deliver/etsi_ts/123500_123599/123501/16.06.00_60/ts_123501v160600p.pdf 139

[128] R. Kundel, A. Rizk, J. Blendin, B. Koldehofe, R. Hark, and R. Steinmetz. (2020)

P4-codel: Experiences on programmable data plane hardware. 142

[129] Zhao, “upf_p4_poc,” https://github.com/801room/upf_p4_poc, 2020. 141

[130] B. Reselman, “5 design principles for microservices,” https://developers.redhat.com/articles/2022/01/11/5-design-principles-microservices, 2022. 143