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

IVAN PETER LAMB

A Multi-Tenant and Target-Independent Programmable Data Plane Virtualization Architecture

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

Advisor: Prof. Dr. Weverton Cordeiro
Coadvisor: Prof. Dr. José Rodrigo Azambuja

Porto Alegre
August 2026

CIP — CATALOGING-IN-PUBLICATION

Lamb, Ivan Peter

A Multi-Tenant and Target-Independent Programmable Data Plane Virtualization Architecture / Ivan Peter Lamb. – Porto Alegre: PPGC da UFRGS, 2026.

138 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2026. Advisor: Weverton Cordeiro; Coadvisor: José Rodrigo Azambuja.

I. Cordeiro, Weverton. II. Azambuja, José Rodrigo.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Prof[a]. Marcia Barbosa
Vice-Reitor: Prof. Pedro Costa
Pró-Reitora de Pós-Graduação: Prof[a]. Claudia Wasserman
Diretor do Instituto de Informática: Prof. Luciano Paschoal Gaspary
Coordenador do PPGC: Prof. Gabriel Luca Nazar
Bibliotecário-chefe do Instituto de Informática: Alexsander Borges Ribeiro

“The darker the night,
the brighter the stars.”
— FYODOR DOSTOEVSKY

ACKNOWLEDGEMENTS

Foremost, I give my thanks to both of my parents, Ivan Lamb and Marlene Nunes, for guiding me not only to the completion of this work, but also to at its very beginning by establishing a high quality education as a goal in my life.

This work was also made possible by my teachers, specially my advisors Weverton Cordeiro and José Rodrigo Azambuja , and my colleagues that supported me throughout my doctorate journey at UFRGS.

ABSTRACT

With the migration of traditional computer networks to the Software-defined Networking paradigm, flexibility is a core feature that novel technologies must provide. In this context, virtualization is gaining traction in Programmable Data Planes (PDP) as a means of achieving greater flexibility, with several solutions in the literature for emulating or instantiating virtual programmable switches on the same host device. Virtualization has numerous advantages in this context, enabling multi-tenancy in programmable data/research center networks and greater device resource utilization. Nevertheless, enabling a complete multi-tenant solution requires management and security considerations not yet approached in previous investigations. Previous works presented in the background of this thesis focus mainly on the core underlying technology necessary to deploy multiple devices in the same physical host. To further advance PDP virtualization towards an industry-adopted technology, this thesis presents a PDP virtualization architecture based on program composition and access control for securely managing virtual switches from different tenants. It describes how the core elements of PDP virtualization presented in the background are used to achieve a robust and modular solution. The proposed experiments highlight the ability to transparently manage virtual switches hosted in a single physical device in networking scenarios with multiple tenants and ascertain the low incurred overhead of the architecture in terms of control plane perceived delay. A novel approach to tackle the runtime reconfiguration of composition-based PDP virtualization is also presented. Concrete use cases of the virtualization solution, such as multi-tenant topology emulation in real hardware and modular data plane programming, are presented and discussed in this thesis. This thesis also hints at future work and how the modular design of the proposed architecture enables expansion of its functionalities.

Keywords: Software Defined Networking. Programmable Data Planes. Multi-Tenancy Virtualization.

Uma Arquitetura Multi-Tenant e Independente de Dispositivo de Virtualização de Planos de Dados Programáveis.

RESUMO

Com a migração de redes de computadores tradicionais para o paradigma de Redes Definidas por Software, a flexibilidade se tornou um recurso fundamental que novas tecnologias precisam prover. Neste contexto, virtualização está ganhando força em Planos de Dados Programáveis (PDP) como uma maneira de alcançar uma maior flexibilidade, com diversas soluções na literatura para emular ou instanciar switches programáveis virtuais no mesmo dispositivo hospedeiro. Virtualização possui diversas vantagens neste contexto, como a possibilidade de *multi-tenancy* em redes programáveis de data-centers ou instituições de pesquisa, além de um uso maior dos recursos dos dispositivos. Ainda assim, para alcançar uma solução *multi-tenant* completa se fazem necessárias considerações sobre segurança e genrenciamento que ainda não foram abordadas por trabalhos anteriores. Os trabalhos existentes apresentados na revisão da literatura desta tese focam nas tecnologias subjacentes necessárias para a implementação de múltiplos dispositivos virtuais em um mesmo hospedeiro físico. Para avançar a virtualização de PDPs em direção a uma tecnologia adotada pela indústria, esta tese apresenta uma arquitetura de virtualização de PDPs baseada em composição de programas e controle de acesso para gerenciar os dispositivos virtuais de diferentes *tenants* de forma segura. Descreve-se como os elementos principais dos trabalhos apresentados na revisão de literatura são utilizados para alcançar uma arquitetura robusta e modular. Também é apresntada uma abordagem inovadora para o problema de reconfiguração de switches virtuais programáveis baseados em composição em tempo de execução. Os experimentos propostos enfatizam o potencial de gerenciar os dispositivos virtuais hospedados em um único dispositivo físico de forma transparente em cenários com múltiplos *tenants* e atestam o baixo *overhead* da arquitetura em termos de atraso percebido pelo plano de controle.

Palavras-chave: Redes Definidas por Software, Planos de Dados Programáveis, Virtualização *Multi-Tenant*.

LIST OF FIGURES

2.1 Comparison between coupled (classical) and decoupled (programmable) network hardware. 21
2.2 Overview of the SDN architecture. 22
2.3 Integration of NFVs in the SDN paradigm. 24
2.4 A protocol-independent switch architecture (PISA). 27
2.5 Example of a header definition on multiple versions of the P4 language. 28
2.6 Example of a parser block in the P4 language and its associated FSM. 29
2.7 Example of a control block in the P4 language and its associated DAG. 30
2.8 Condensed view of the Tofino Native Architecture. 32
2.9 MicroP4 and Lyra code examples. 33
2.10 PDP runtime example scenario. 37
2.11 Programmable switch virtualization example. 40
2.12 Example of the lightweight merging of two parsers. 45
2.13 Comparison of the canonical NetFPGA and P4VBox designs. 46
2.14 Multi-tenant switch virtualization example. 48
2.15 International P4 Experimental Networks. 49
2.16 MTPSA Architecture. 50

3.1 Virtualization solution architecture. 57
3.2 Code Composition Module breakdown. 60
3.3 Example of the AST representation of a P4 code. 63
3.4 Example of the elimination of a nested parser. 64
3.5 Example of headers that can be unified. 65
3.6 Example header unification. 66
3.7 Example of the program-wide merging process. 67
3.8 Example of the merging of P4 tables. 69
3.9 Example of the merging of control *apply* blocks. 73
3.10 Runtime module operation scenario. 75
3.11 Brief overview of the TLS certificate signing process. 76
3.12 Example of modifications on the Packet I/O headers for runtime control. 80
3.13 Migration of TMS Elements. 83
3.14 Step-by-step hot-swap process. 85
3.15 Topology for the composition and latency experiment. 89
3.16 Topology for the isolation experiment. 91
3.17 Flows in the receptors on the isolation experiment. 92
3.18 Topology of the runtime deployment experiment. 93

4.1 Example of a tenant-defined topology and its text description. 99
4.2 Topology of the latency comparison experiment. 100
4.3 Topology of the resource usage experiment. 101

5.1 Example of annotations in header fields. 106
5.2 Example of the *VirtShared* annotation. 107
5.3 GNN Experiment Model. 112
5.4 Flowchart for the data plane GNN algorithm. 116
5.5 Topology of the experiments with the GNN P4 framework. 117
5.6 Flowchart for the data plane GNN algorithm. 119

C.1 Example of the merging of three ingress parsers. 138

LIST OF TABLES

2.1 Summary of PDP virtualization related works. 52

3.1 Delays measured on the latency experiment..... 90
3.2 Results for the runtime deployment experiment. 94

4.1 Comparison of the different targets used in the virtual topology experiment. 100
4.2 Resource usage of the individual programs and of the merged program. 102

5.1 Results for the simulated experiments 113
5.2 Results for the experiment considering random topologies 114
5.3 Total throughput for the TCP congestion control schemes..... 118

LIST OF ABBREVIATIONS AND ACRONYMS

AI	Artificial Intelligence
ANN	Artificial Neural Network
ARP	Address Resolution Protocol
ASIC	Application-Specific Integrated Circuit
bmv2	Behavioral-Model Version 2
CAP	Cross-App Poisoning
CDPI	Control-Data-Plane Interface
CLI	Command-Line Interface
CPU	Central Processing Unit
CSR	Certificate Signing Request
DAG	Directed Acyclic Graph
DCTCP	Data Center TCP
DNS	Domain Name System
DSL	Domain Specific Language
ECN	Explicit Congestion Notification
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
GNN	Graph Neural Network
HDL	Hardware Description Language
IDS	Intrusion Detection System
IFC	Information Flow Control
INC	In-network Computing
IP	Internet Protocol
JSON	JavaScript Object Notation

LAN	Local Area Network
MAC	Media Access Control
ML	Machine Learning
MLP	Multi Layer Perceptron
NAT	Network Address Translation
NBI	Northbound Interface
NFV	Network Function Virtualization
NS3	Network Simulator 3
NTMS	Non-Traffic-Modifiable State
ONF	Open Networking Foundation
ONOS	Open Networking Operating System
OSI	Open Systems Interconnection
OSPF	Open Shortest Path First
P4	Programming Protocol-Independent Packet Processors
PDP	Programmable Data Plane
PISA	Protocol-Independent Switch Architecture
POF	Protocol-oblivious Forwarding
RED	Random Early Detection
RFC	Request for Comments
RIP	Routing Information Protocol
RPC	Remote Procedure Call
RTT	Round Trip Time
SBI	Southbound Interface
SDN	Software Defined Networking
TCP	Transmission Control Protocol
TLS	Transport Layer Security

TMS	Traffic-Modifiable State
TNA	Tofino Native Architecture
TTL	Time to Live
UDP	User Datagram Protocol
VHDL	Very high speed integrated circuit Hardware Description Language
vIFC	Virtual Information Flow Control
VM	Virtual Machine
VSMA	Virtual Switch Management API

CONTENTS

1 Introduction **14**
- 1.1 Objectives 16
- 1.2 Contributions 16
- 1.3 Outline 17

2 Background and Related Works **18**
- 2.1 Sofware-Defined Networking 18
  - 2.1.1 Brief history of the IP protocol 19
  - 2.1.2 Efforts towards network flexibility 20
- 2.2 Network Function Virtualization 23
- 2.3 Programmable Data Planes 25
- 2.4 The P4 Programming Language 26
  - 2.4.1 Header Definitions 28
  - 2.4.2 Parser Blocks 29
  - 2.4.3 Control Blocks 30
- 2.5 Modular Data Plane Programming 32
- 2.6 PDP Runtime Interface 34
- 2.7 In-network Computing 38
- 2.8 Virtualization in PDPs 39
  - 2.8.1 Hyper4 41
  - 2.8.2 HyperVDP 43
  - 2.8.3 P4Visor 43
  - 2.8.4 P4VBox 46
- 2.9 Multi-tenancy in PDPs 47
  - 2.9.1 P4MT 49
  - 2.9.2 MTPSA 50
  - 2.9.3 SwitchVM 51
- 2.10 Summary of Related Works 52

3 Multi-Tenant Virtualization Architecture **55**
- 3.1 Design Goals 55
- 3.2 P4 Code Composition 59
  - 3.2.1 P4 Source Code Analysis 61
  - 3.2.2 Equivalent Element Unification 62
  - 3.2.3 Virtual Headers' Creation 66
  - 3.2.4 Parser and Control Blocks Merging 67
  - 3.2.5 Target-Specific Adjustments 72
  - 3.2.6 Runtime Mapping and Merged Program Output 73
- 3.3 Runtime Virtualization Management 74
  - 3.3.1 Controller Authentication 75
  - 3.3.2 Runtime RPC modifications 77
- 3.4 Runtime Deployment of Virtual Switches 79
  - 3.4.1 Migration of TMS Elements 81
  - 3.4.2 Step-by-step Hot-swap Process 85
- 3.5 Evaluation 88
  - 3.5.1 Composition and Latency Experiment 88
  - 3.5.2 Isolation Test 90

3.5.3 Runtime Deployment 92

4 Topology Emulation Integration **95**
4.1 Network Topology Simulation and Emulation 95
4.2 Recent Advances in SDN and PDP emulation 97
4.3 Multi-Tenant Hardware-Based Topology Emulation 98
4.4 Experimental Results 99
4.4.1 Performance Test 99
4.4.2 Resource Usage Test 101
4.5 Planned Features 102

5 Sharing Virtual Switch Resources **104**
5.1 Augmenting the PDP programs to implement resource sharing 105
5.2 Application in Modular Programming 108
5.2.1 Graph Neural Networks 108
5.2.2 Integration of GNNs in PDPs 109
5.2.3 Evaluation with Simulations 110
5.2.3.1 Case-Study: DCTCP ECN markings 110
5.2.3.2 Dataset Generation and Model Training 111
5.2.3.3 Implementation 112
5.2.3.4 Results 113
5.2.4 P4 Framework 114
5.2.5 Advantages of Modular Programming 118
5.2.6 Conclusion 118
5.3 Final Considerations 119

6 Conclusion and Future Work **121**
6.1 Contributions 121
6.2 Future Work 123

References **125**

APPENDIX A — Resumo Expandido em Português **132**
A.1 Introdução 132
A.2 Motivações 133
A.3 Contribuições 133
A.4 Resultados Experimentais 134
A.5 Trabalhos Futuros 135

APPENDIX B — Publications **137**

APPENDIX C — Example of the Merging of Parser Blocks **138**

1 INTRODUCTION

The computer networks area has significantly evolved since its early conceptions in the 1960s (ABBATE, 1994), to the advancements that enabled the World Wide Web (KUROSE; ROSS, 2010) and to the modern age of Software-Defined Networking (SDN) (OPEN-NETWORK-FOUNDATION, 2013). Throughout those years, researchers and operators realized that network flexibility was a key objective to strive for. Many limitations of today's network protocols come from the underlying hardware infrastructure, which was not designed for agile updates. The mass deployment of those fixed-functionality hardware devices has led to slow-evolving network cores.

Aiming for a less "ossified" computer networks paradigm with flexibility-driven design philosophy, many advancements were made in favor of Programmable Data Planes (PDP) (BOSSHART et al., 2014), which essentially allow operators to fully define the forwarding behavior of their hardware components. This leads to configurable network architectures that can be tailored to each operator's necessities. Furthermore, researchers in the computer networks area also benefit from this technology, as prototypes of network protocols and hardware become much more agile, avoiding the need to design and craft specialized hardware. This approach is also highly attractive from a performance point of view, as it leverages modern programmable devices' line-rate packet processing speed (currently in the order of terabits per second) to offload part of the data processing that typically occurs in traditional servers or end-host devices. In summary, programmability is a must-have in our ever-changing network infrastructure environment.

In this new era of networks, the complexity of the operations performed by the core devices has increased, as they can implement almost any traditional network algorithm that the operator defines (KIANPISHEH; TALEB, 2023) and new non-standard functionalities on top of that. This scenario creates a need for appropriate testing, prototyping, and debugging environments for innovative network designs.

Recently, PDP virtualization has emerged as a tool to enable, among other solutions, these kinds of development environments (HANCOCK; MERWE, 2016). It allows for hardware slicing of the data plane devices similar to traditional virtual machines on commodity servers (TANAEMBAUM; BOS, 2014). That way, a single physical programmable switch, which can cost tens of thousands of dollars depending on its specifications (excluding maintenance costs during operation), could be used to implement multiple virtual switches simultaneously.

However, existing PDP virtualization solutions have yet to achieve the maturity necessary for industry and market adoption. Much work has been done on the core technologies that enable the solutions, but as of today, they still lack proper abstractions to manage the deployed virtual devices and to deal with multi-tenant scenarios, in which each virtual device belongs to different "users". Previous works in this direction overlimit the individual capacity of each tenant in terms of performance of virtual devices, the available features of the programming language, the possibility of runtime adhoc reconfiguration, among other limitations.

This also includes security considerations that are not necessary on systems with a single user. Recent surveys have identified malicious applications and controllers as severe threats to SDNs (DACIER et al., 2017), but no current solution has abstractions that protect the virtual devices against those threats. Also, most of the early works on virtualization are evaluated at software switches such as bmv2 (HANCOCK; MERWE, 2016; ZHANG et al., 2019; ZHENG; BENSON; HU, 2018), which are not intended for deploying PDP solutions. A solution that aspires to become more than just an idea must be designed to be implemented and evaluated at real industry-level hardware. That imposes challenges that cannot be abstracted as "out-of-scope" at the prototype level, such as the maximum number of packet processing stages and minimum throughput guarantees. Those are the next steps that must be taken for PDP virtualization to escape the academic research phase and reach broad adoption.

Additionally, target-independency is a key feature to strive for in this context. It makes solutions easily portable to multiple network infrastructures and fosters research reproducibility by sharing research artifacts. As discussed further in this thesis, many works on PDP research are not allowed to publish those artifacts (such as source codes) due to non-disclosure agreements with hardware vendors.

In order to tackle those limitations, the following research questions are raised:

1. Is it possible to provide PDP virtualization without hardware support?

This thesis presents a complete multi-tenant and target-independent virtualization architecture alongside a reference implementation. The design principles of the solution aim to address the current limitations in terms of security, management, and scalability of the current approaches. Alongside the architecture design, a reference implementation is also provided and discussed to foster both research reproducibility and use in larger projects, expanding the usage of PDP virtualization technology. The proposed architecture allows (i) the composition of distinct switch programs at source-code level to create

a single program that implements the logic of all network functions defined on the individual programs belonging to different tenants, (ii) the mapping of the elements of the composed program, such as match-action tables and registers to the elements of the virtual switches, (iii) an access control scheme for the tenants to securely and transparently access the elements of their virtual switches, and (iv) runtime logical reconfiguration for the composition-based virtual switches.

1.1 Objectives

The specific objectives established for this thesis to take the next step toward the broader adoption of PDP virtualization technologies are the following:

- The study of current PDP virtualization solutions and their limitations.
- The design of an architecture for PDP virtualization that addresses those limitations.
- The implementation of the architecture, which shall be made publicly available.
- The study of use-case scenarios that benefit from the architecture and the implementation and evaluation of solutions for those scenarios.

1.2 Contributions

The following are the principal contributions presented with this thesis:

- **A multi-tenant and target-independent PDP virtualization architecture:** The main contribution of this thesis is the design of a virtualization architecture that addresses the current limitations to enable large-scale integration of PDP virtualization. The architecture is designed to operate in multi-tenant environments and to be compatible with different hardware targets.
- **A novel runtime deployment scheme for composition-based virtual switches:** The tenants' capacity to deploy virtual switches at will, just as they would reconfigure a traditional programmable device, is one of the biggest challenges of composition-based PDP virtualization. The architecture proposed in this thesis provides a strategy to perform logical deployment of virtual switches, with expected results from the point-of-view of each individual tenant.

- **A publicly available implementation of the architecture:** The proposed virtualization architecture was implemented to be evaluated with multiple targets, such as the commercial Tofino Switches from Intel and popular SDN controllers such as ONOS (Open Network Operating System). The implementation is made publicly available in a GitHub repository[1].

- **A solution based on the architecture for a PDP prototyping environment:** The implementation of a multi-tenant solution for topology emulation based on the presented architecture is also evaluated in this thesis. The solution allows simultaneous prototyping with real hardware of multiple programmable virtual forwarding devices organized in a user-defined topology with limited resources. The users of the system (tenants) can share the same physical substrate to speed up the development of network protocols and algorithms.

- **An explicit resource sharing scheme for multi-tenant scenarios:** The virtualization architecture provides the option for tenants to share switch elements. This enables further resource savings on the target hardware and modular data plane programming. This thesis also provides a case study of in-network graph neural network inference leveraging modular data plane programming enabled by the virtualization architecture.

1.3 Outline

The remainder of this thesis is organized as follows. Chapter 2 presents the background for the topics of this thesis, mainly PDP virtualization, along with the related works of the area. Chapter 3, presents the design of an architecture for PDP virtualization that tackles the limitations of the state-of-the-art and its implementation insights. Chapter 4 presents a use-case scenario of the proposed architecture focused on the development of innovative PDP algorithms and protocols. Chapter 5 discusses the explicit resource sharing features of the virtualization architecture and provides a case study of a modular PDP graph neural network inference engine enabled by this technology. Finally, Chapter 6 concludes the thesis by summarizing the contributions and discussing possible directions for further work.

[1] Available at: <https://github.com/Ivanatorion/VirtualizationArchThesis>

2 BACKGROUND AND RELATED WORKS

This Chapter describes the main concepts of the objects of study of this thesis, mainly Software Defined Networking and Programmable Data Planes and their applications. This Chapter also relates the ideas with the principal works of the area.

As some of the concepts are closely related to the works that originated the study area, this thesis is structured to keep those together in the same Chapter for a better reading experience. For example, it is easier to understand Programmable Data Plane Virtualization while reading about the solutions that implement the concept differently (e.g., Hyper4, P4Visor, etc.).

One of the primary objects of study in this thesis is network forwarding devices. Those devices can have a plethora of different names depending on the context. For example, the term "router" is typically attributed to a forwarding device that operates in the third layer (network) of the Open Systems Interconnection (OSI) model (KUROSE; ROSS, 2010). Another case is the term "switch" which commonly refers to a second-layer (link) device in the same model. Still, in the modern paradigms, it may refer to any forwarding device.

To avoid confusion with the terminology, this thesis uses the terms "switch" and "forwarding device" to refer to any kind of network device. Nothing is assumed *a priori* about the behavior of said devices other than that they forward network packets according to a certain logic and can perform certain procedures on said packets.

2.1 Sofware-Defined Networking

This Section describes the central concept behind modern computer network innovations. It is no overstatement to say that since the change of paradigm provided by SDNs, research in the area of computer networks has the intrinsic objective of giving operators greater flexibility in their software and hardware artifacts.

To illustrate the differences between classical computer networks and justify the need for this new paradigm, this Section begins with a short example of the difficulties associated with "inflexible" hardware focused on the Internet Protocol (IP). After that, the concept of SDN itself is presented along with references to the main works in the area.

2.1.1 Brief history of the IP protocol

The Internet is one of the most ubiquitous pieces of technology in the day-to-day life in the field of computer networks. Through it, people can access an ever-growing catalog of services such as social networks, literature research, online shopping, and media streaming, among others; a list so vast it would not be practical to be exhaustive. On the Internet, traffic is routed by the IP protocol[1]. As of the time of the writing of this thesis, most of the internet traffic is routed by the IPv4 version (ORDABAYEVA et al., 2020), formally specified in Request for Comments (RFC) 791 as early as 1981.

Based on these statements, one might think that the advancements of the technologies that support today's global computer network have stagnated for over 40 years, which is a half-truth. Not so long after the initial deployment of the IPv4-based network infrastructure in the 1980s, researchers and operators identified problems with that protocol version. The most concerning of which was very practical: we will run out of unique IPv4 addresses[2]. To address the issues, updated protocol versions were designed, with IPv6 being the most promising one, formally specified in RFC 1752 in 1994.

However, there was (and still is) a significant problem: the deployed hardware operates with IPv4 only. This issue means that new hardware would need to be deployed and ready to operate with the latest version of the protocol. One can imagine that retiring "perfectly working" hardware because of a new version of a protocol and having to buy and deploy new devices to operate in their place did not appeal to network operators.

As a result, more involved solutions had to be proposed. One such **workaround** is the well-known IP Network Address Translation (NAT). The emphasis on the term "workaround" is to reinforce the idea that such a thing should not exist in a utopic, ideal world where hardware deployment has no cost. The very RFC that defines NAT, 1631, states that it was supposed to be a short-term solution to the problem. That is not to say that NAT is a bad solution to the problem. It is an ingenious mechanism that sustains the modern Internet.

The root of the problem is that the hardware deployed in the core of the Internet, and of networks in general, needed to be more flexible (ANDERSON et al., 2005). That gave origin to the term **"ossification"**, which denotes the difficulty of changing and adapting the underlying workings of network protocols.

[1]The expression "IP protocol" will be used sometimes in this thesis for clarity, even though the "P" stands for Protocol.

[2]That was the concern in the 1990s, as of 2023 we have run out already.

2.1.2 Efforts towards network flexibility

Dealing with the issue of network ossification was necessary for a more agile environment for the development of computer network innovations. Attempts to address the problem gave origin to the general concept of *network programmability*. The core idea of network programmability is to separate the control logic of how network packets should be processed and forwarded from the underlying hardware responsible for the operations. This core principle extends to SDNs but with more formal, open, and standardized interfaces in the latter.

Although some authors may consider the concept of SDNs as new as of the 2000s, multiple surveys acknowledge that one of its fundamental principles (network programmability) dates from as early as the 1960s (ANEROUSIS et al., 2021). The acknowledgment of the problem predates even the Internet itself, and programmable network solutions have been employed in specific scenarios since then.

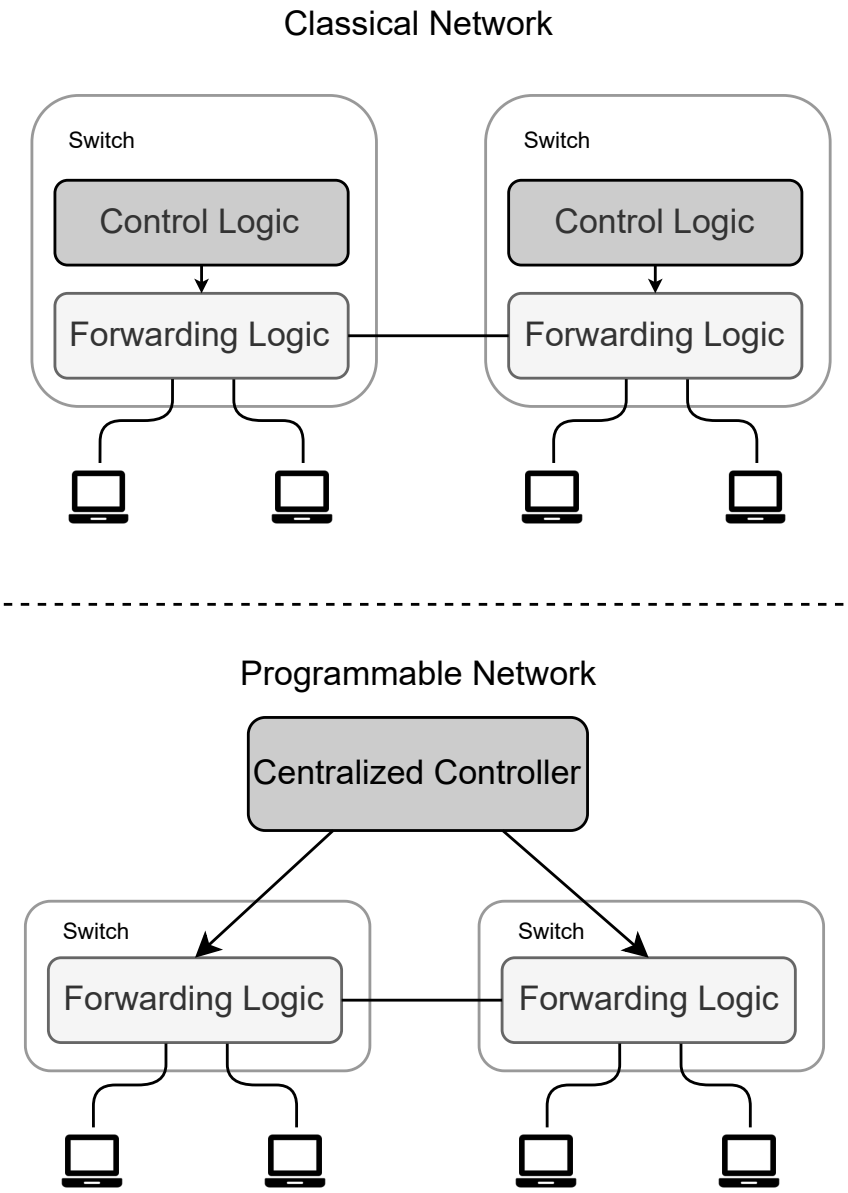
Figure 2.1 illustrates this concept of network programmability. The hardware of the Classical Network (upper half) has fixed functions and network protocols implemented in its control logic. This logic includes the implementation of traditional algorithms such as shortest-path computation, and protocols such as OSPF, RIP, IPv4, Ethernet, etc. The forwarding logic consists of the elements responsible for the forwarding itself, such as match action tables, registers, and counters. In contrast, a Programmable Network (bottom half) employs a separated, logically centralized controller, which can run in a traditional server. That has numerous advantages compared to the classical approach, such as:

- **Less Complex Hardware:** With the external control logic implementation, the forwarding devices' hardware can be less complex, as they do not need to implement sophisticated algorithms. That incurs the reduction of hardware costs for the network operators.

- **Adaptable Control Logic:** With a decoupled architecture, vendors can more easily adapt their control logic to employ different algorithms. This translates to a finer grain of tuning for the vendors.

- **Reduction of In-Band Control Data:** Many routing decisions in networks depend on information about the whole network. With the classical approach, the devices must obtain this information by exchanging data "in-band" (that is, using the same

Figure 2.1: Comparison between coupled (classical) and decoupled (programmable) network hardware.



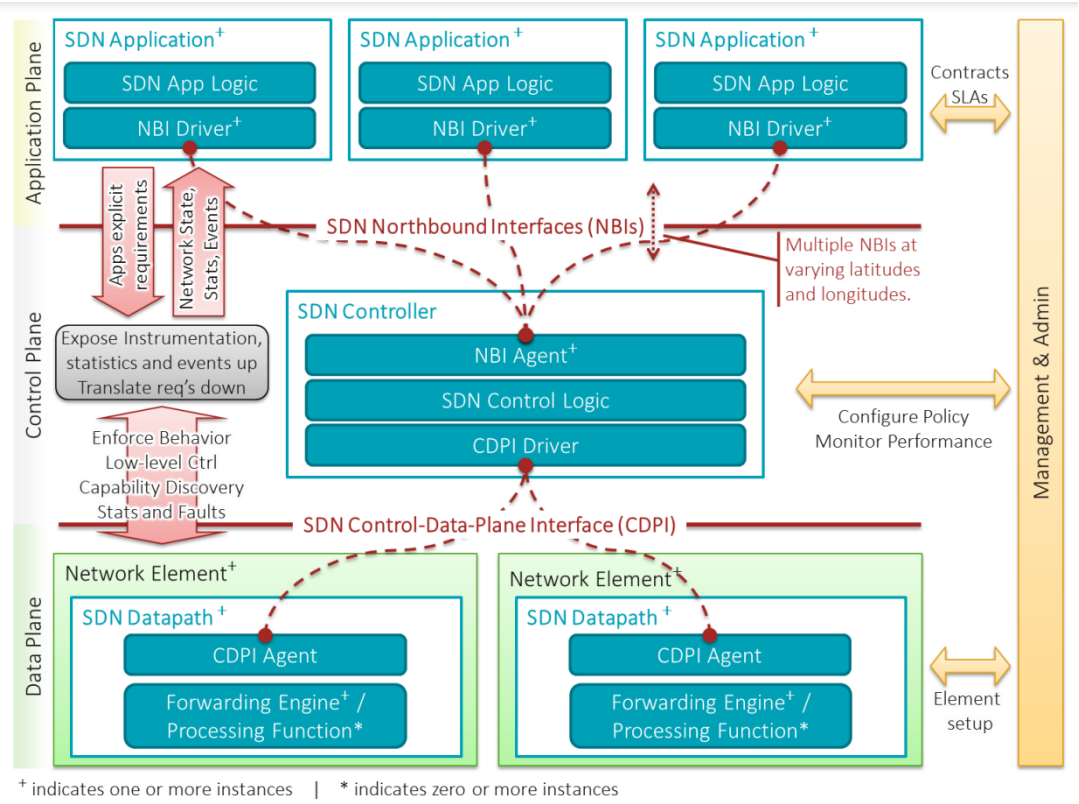
Source: The Author.

links as user data). With a centralized control logic, a broader vision is available without the need for this data exchange.

One may ask what differentiates modern SDNs from those early network programmability attempts. The answer is the other core principles of SDNs, open interfaces, and operator flexibility. While the early iterations of programmable network solutions would start to address the issues caused by inflexible hardware with tightly coupled control logic, they would still have vendor-specific interfaces and offer little space for the operators themselves. The reason for this paradigm change can be attributed to the

rapidly growing network technologies, such as virtualization, Web APIs, application marketplaces, etc., that demanded a more agile development environment (TAN et al., 2016).

Figure 2.2: Overview of the SDN architecture.



Source: (OPEN-NETWORK-FOUNDATION, 2013)

Many researchers relate the transition to modern SDN to the OpenFlow Protocol (MCKEOWN et al., 2008) from 2008. To better understand this protocol, it is useful to look at the SDN architecture depicted in Figure 2.2. It divides the network into three distinct planes, which are:

- **Application Plane:** Consists of applications that implement a specific network functionality. Operators can deploy multiple applications, such as a routing application, a firewall application, a monitoring application, etc. Those applications communicate only with the control plane through the Northbound Interface (NBI).

- **Control Plane:** A centralized entity[3] that translates the requests from the applications in the plane above to the devices in the plane below. It may also provide other functionalities, such as authentication and security. It provides the applications with an abstract view of the forwarding elements in the data plane.

[3]The controller can be viewed as a logically centralized entity, but its actual implementation may be distributed (OKTIAN et al., 2017).

- **Data Plane:** This is the place where the forwarding elements are located. They receive the forwarding logic from the control plane through the Southbound Interface (SBI), also known as the Control-Data-Plane Interface (CDPI). One of the advantages of the SDN architecture is that the data plane devices can originate from different vendors and even be implemented with various technologies, such as FPGAs, ASICs, software emulation, and others. The CDPI standardization is what enables these different elements to be deployed in the same environment.

The OpenFlow Protocol was designed to operate on the CDPI of the SDN. It was an effort to create a standard way in which the controllers communicate with forwarding devices, regardless of the model or vendor. That would allow operators, for example, to insert a set of forwarding rules on an IP table of different routers with the same controller program. The OpenFlow protocol defines a set of network packet headers (Ethernet, ARP, IP, etc.) and allows operators to insert rules into the forwarding devices' tables that match the fields of these headers. The specification needs to be updated to accommodate new network protocols constantly and is currently maintained by the Open Networking Foundation (ONF) (OPEN-NETWORK-FOUNDATION, 2015).

2.2 Network Function Virtualization

The concept of Network Function Virtualization (NFV) derives directly from the flexibility of SDNs. It aims to provide an alternative approach to the "ossified" implementation of many functionalities traditionally delegated to the core of the networks (LI; CHEN, 2015). As previously mentioned, many useful network functionalities, such as access control, DNS caching, firewalls, etc., are traditionally implemented in a "hardcoded" manner in the forwarding devices.

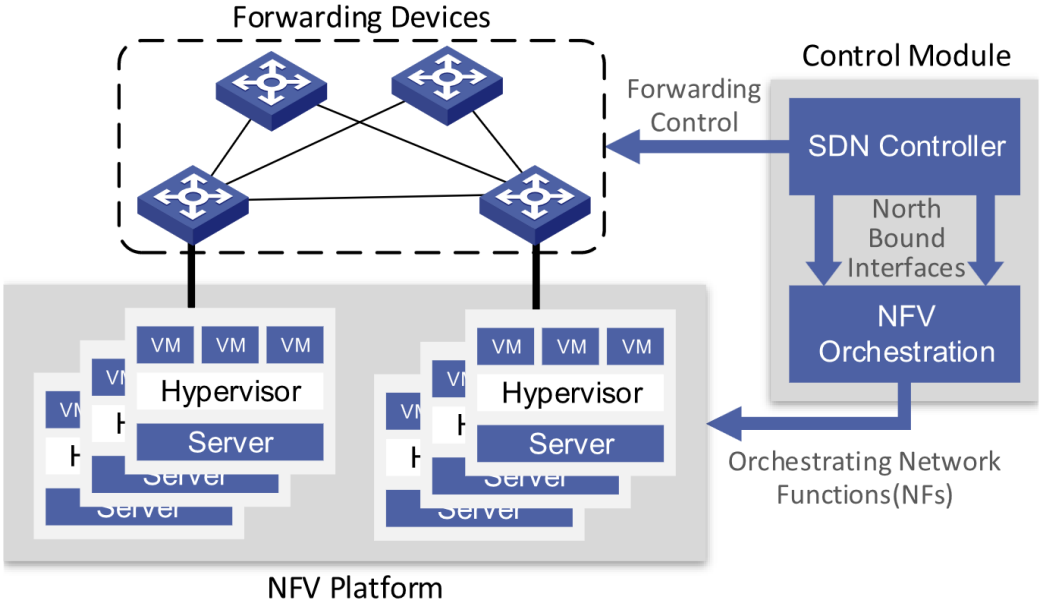
Essentially, NFVs are implementations of those functions using traditional general-purpose servers. In the SDN paradigm, each application (in the application plane) would be responsible for one of those functions. The forwarding hardware would only need to define tables and registers to be used by the external control logic. The end goal is to give operators more precise control of the algorithms and services that the network is providing. Doing so allows fine-grain customizations suited for each environment and reduces the need for network middleboxes.

In that way, a chain of proprietary hardware from multiple vendors deployed on

the network infrastructure could be replaced by NFVs. Those middleboxes' functionalities include NAT (EGEVANG; FRANCIS, 1994), load balancing (BOWMAN-AMUAH, 2003), and Intrusion Detection Systems (IDS) (LIAO et al., 2013), among others.

In fact, the chaining of those functionalities is a hot topic of research and is formally specified as "service-function chaining". RFC 7498 (QUINN; NADEAU, 2015) describes the problem as "the definition and instantiation of an ordered list of instances of such service functions." With NFVs, the whole process becomes more flexible, from the algorithm specifications to the "chaining" itself.

Figure 2.3: Integration of NFVs in the SDN paradigm.



Source: (LI; CHEN, 2015).

Figure 2.3 shows the integration of NFVs in the SDN architecture. It presents high-speed links between four forwarding devices, allowing the offloading of the aforementioned functions (e.g., NAT, firewall, etc.) to commodity servers. It also emphasizes the possibility of these functions being deployed in an isolated manner through VMs (or containers). Also, as previously stated, NFVs are implemented in the application plane of SDNs. One might expect a control plane layer between the devices and the servers to provide the abstractions to the applications. This control layer can be omitted or replaced by application-plane logic (i.e., the NFV Orchestration module) when the NFVs are specific to a single device's traffic and the speed of the communication required prevents a more complex control structure. That is to say, the SDN architecture (depicted in Figure 2.2) is not "set in stone", as flexibility is a core principle of SDN.

2.3 Programmable Data Planes

Programmable Data Planes is another concept enabled by SDNs that is extremely relevant to this thesis. The foundations for the communications between the SDN layers were established with OpenFlow, but it still maintained a relatively high degree of ossification due to it having a set of supported headers and protocols. Even if the OpenFlow specification were ever-changing to accommodate the newest innovations in terms of protocols, it would, by design, disallow the immediate prototyping of protocol ideas by researchers. Due to these limitations, there was a collective effort to develop alternatives that would allow greater flexibility.

As an example, to better understand this limitation, consider the development of a protocol that changes the way packets are processed by a switch in a higher layer, such as SCION (ZHANG et al., 2011), which operates in the network layer, similar to the IP protocol. Before PDPs, prototyping for such a protocol would be a complex process, as it would need hardware support, which entails the development of new switch models. Moreover, the interfacing with this hardware for this protocol would need to be approved by an organization in charge of a control plane management interface such as OpenFlow, which, again, takes its due time.

With PDPs, one can quite literally implement and test such protocols in a matter of hours or a couple of days. One definition of a PDP is: **A data plane infrastructure that allows the operators to freely customize the processing logic of the devices in a complete protocol-agnostic manner**. The goal of PDPs is to allow network operators to define an arbitrary number of packet header fields to be matched and to define the actions to be performed when those fields are matched. To that end, Domain Specific Languages (DSLs) were introduced as a standard way of describing the packet processing logic and to allow data plane programming. Those languages allow the specification of header fields, match table definitions, implementation of actions performed, checksum verification, security protocols, etc.

The initial idea of PDPs comes with the DSL from POF (Protocol-oblivious forwarding) (SONG, 2013). As the name suggests, it aims to provide decoupling of the forwarding logic of the network devices from the protocol definitions that should come from the users. This work paved the way for more complex DSLs, defining a whole programmable switch architecture that justifies its design principles.

2.4 The P4 Programming Language

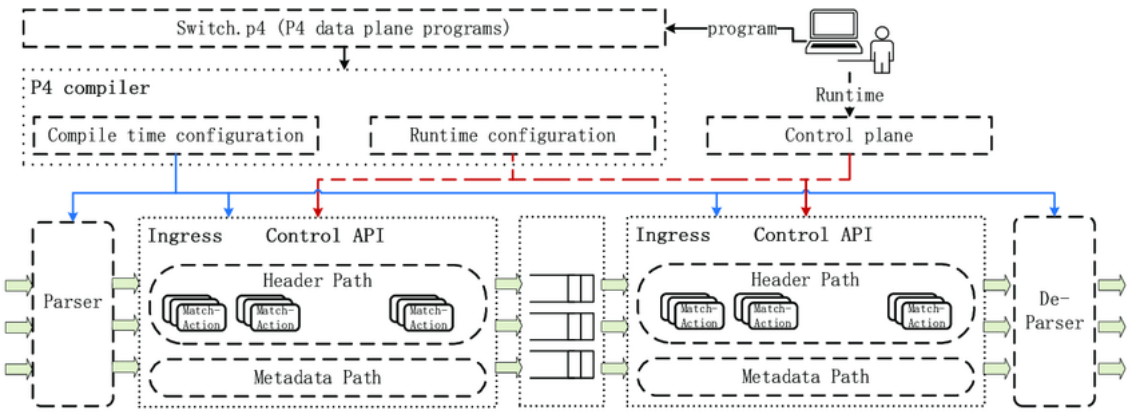
One other DSL designed to describe the behavior implemented by PDPs is P4 (Programming Protocol Independent Packet Processors) (BOSSHART et al., 2014). While there were (and still are) many other attempts to create DSLs for PDPs, such as OpenState (BIANCHI et al., 2014) and Domino (SIVARAMAN et al., 2016), P4 is by far the most broadly present DSL for this application both in industry and in academia nowadays. It was adopted as the language of choice for multiple *targets* (products that can be programmed by the language) from different vendors available in the market, such as Nvidia, Xilinx, and Intel, to cite a few. Other DSLs are not even close to the market adoption that P4 has at the time of writing this thesis.

Inspired by POF, P4 is an effort to formalize a language that provides the maximum possible amount of freedom to the network operators to define the behavior of their devices. Naturally, one of its design goals is that vendors would be able to develop programmable hardware capable of efficiently implementing the programs described with it to achieve high bandwidth and low latency. To that end, the language must be domain-specific to network forwarding devices and can not be as generic as general-purpose languages such as C (P4 does not allow loops, for instance). Citing Bosshart et al. (2014), the design principles of the language are:

- **Reconfigurability.** The controller should be able to re-define the packet parsing and processing in the field.
- **Protocol independence.** The switch should not be tied to specific packet formats. Instead, the controller should be able to specify (i) a packet parser for extracting header fields with particular names and types and (ii) a collection of typed match+action tables that process these headers.
- **Target independence.** Just as a C programmer does not need to know the specifics of the underlying CPU, the controller programmer should not need to know the details of the underlying switch. Instead, a compiler should consider the switch's capabilities when turning a target-independent description (written in P4) into a target-dependent program (used to configure the switch).

Figure 2.4 shows an example of a P4-defined protocol-independent switch architecture (HANG et al., 2019). In the switch, the parser and deparser are configured to

Figure 2.4: A protocol-independent switch architecture (PISA).



Source: (HANG et al., 2019)

process user-defined packet headers. The ingress and egress pipelines implement match-action tables to process the packets in arbitrary user-defined stages, matching the packet headers with rules to perform the corresponding actions[4]. Those actions use language primitives to modify the packet's metadata and headers.

It is also depicted in Figure 2.4 the process to deploy the source code (Switch.p4) to the switch, which involves the compilation (P4 Compiler) and configuration (blue lines). The runtime configuration (red lines) allows the Control Plane to insert match-action rules on the switches after the initial deployment. Section 2.6 describes the runtime configuration of the switches in more detail.

From the programmer's point-of-view, a P4 program consists of:

- **Headers:** Headers define the structure of a set of fields, including width and constraints on the values of these fields.

- **Parsers:** A definition of a parser specifies how to identify headers and valid header sequences within packets.

- **Tables:** The main mechanism for packet processing. Match-action tables apply match actions on the packet's defined headers and perform the specified action for the match.

- **Actions:** Complex actions can be built from the P4 protocol-independent primitives. Those actions are available within match-action tables.

[4]Actions may be decreasing TTL, calculating checksum, dropping invalid packets, sending the packet to the control plane, etc.

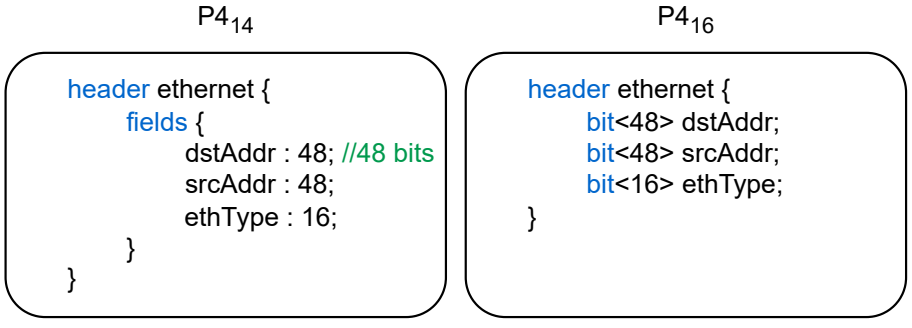
- **Control Programs:** The control program determines the order of match-action tables to be applied to the packets. It is a simple imperative program to describe this flow of control.

Those definitions, cited directly from the P4 introductory paper (BOSSHART et al., 2014), are crucial to fully understand a fundamental part of this thesis contribution described later in Section 3.2, which is the composition of P4 programs. Specifically, it is important to understand which components of a P4 program are fully *target-independent* and which are not. As such, they are described in more detail in the following subsections with that emphasis.

2.4.1 Header Definitions

Headers are the main data structures extracted from packets to direct the control flow according to a specific protocol logic. P4 programs declare the headers as a list of fields with their specific size (number of bits) for use. Figure 2.5 shows an example of an Ethernet header definition in multiple versions of the P4 language. As of the moment of the writing of this thesis, $P4_{16}$ is the current version of the P4 language and is largely incompatible with its predecessor $P4_{14}$. All of the examples of P4 code in this thesis are from the $P4_{16}$ version unless stated otherwise.

Figure 2.5: Example of a header definition on multiple versions of the P4 language.



Source: The Author

Even though the P4 language specification does not impose any constraints on the total size of each header, specific hardware targets may require, for example, that the total header bit length be a multiple of 8. Those restrictions pose an obstacle to creating target-independent P4 code, as they must be taken into account when creating custom headers for user-defined protocols. The most efficient way to circumvent these restrictions also varies

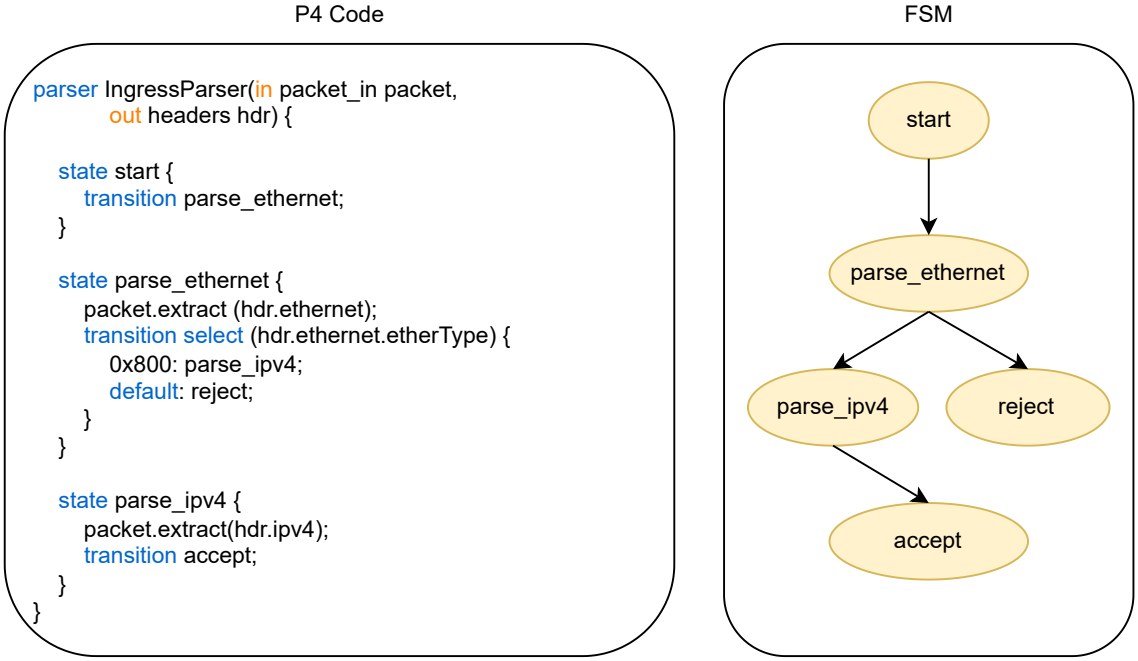
from target to target. On the Tofino Native Architecture employed by Intel®Tofino™ switches, for example, a P4 annotation is used to denote fields that are added only to comply with the required bit length of headers which the compiler can optimize. For other targets, the mechanism for doing so may be different.

2.4.2 Parser Blocks

A P4 program, syntactically, is comprised of many definitions of language-specific elements. Among these elements, **parser and control blocks** are the ones where the actual packet processing logic, in terms of control flow, is defined. Those blocks are then combined in an architecture-specific way[5] to form the *main* P4 program declaration.

Parser blocks specify how the defined headers are extracted from the packets that arrive in the switch. Formally, a parser can be described as a Finite State Machine (FSM) with an initial state (*start*) and two different final states (*accept* and *reject*). The semantics for the final states are yet another architecture-specific definition, but a common definition is that packets that reach the *accept* state are forwarded to the next block of the architecture pipeline, and packets that reach the *reject* state are dropped.

Figure 2.6: Example of a parser block in the P4 language and its associated FSM.



Source: The Author

[5] In the v1model architecture, for example, a switch is comprised of 6 blocks: 1 parser and 5 control blocks with different functionalities.

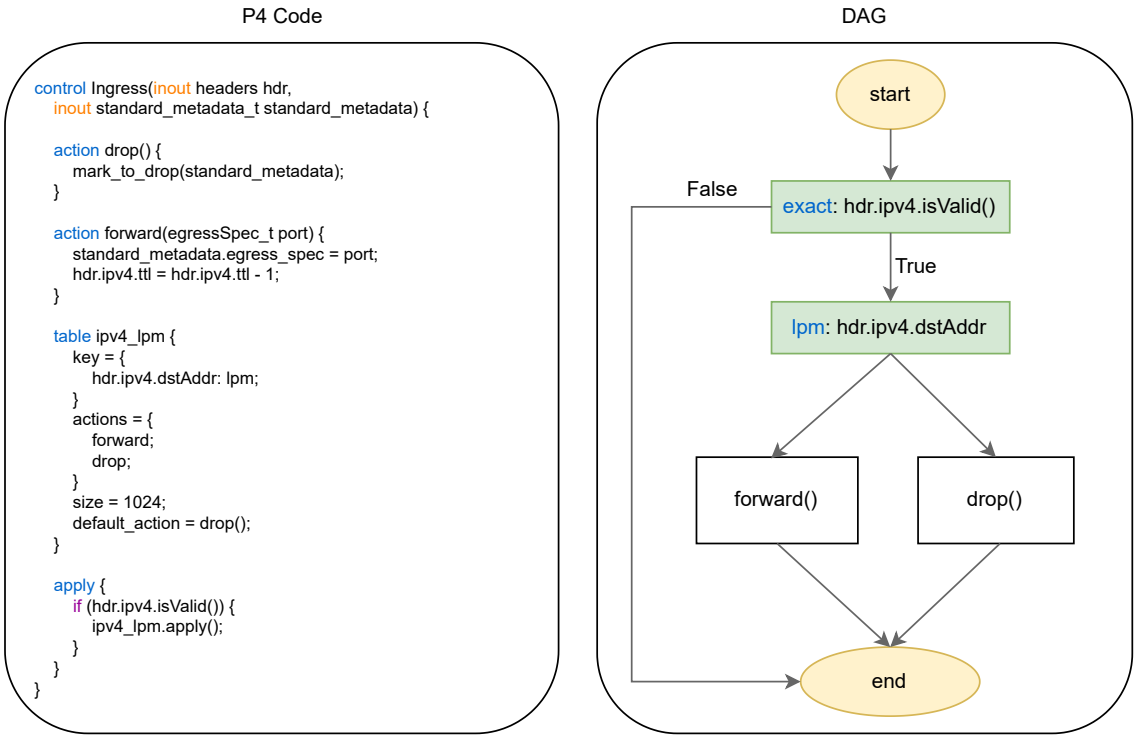
Figure 2.6 shows an example of a parser block that extracts layer 2 (Ethernet) and layer 3 (IPv4) of common internet packets. It shows the P4 code of such a parser along with the FSM associated with it. An important concept of parser blocks is that while they may contain loops in their FSM definitions (such as an MPLS stack parser of arbitrary size), hardware targets are likely to unroll these loops. This way, we can also treat parser blocks as Directed Acyclic Graphs (DAGs) for these targets.

2.4.3 Control Blocks

The control blocks of the P4 language are where the core of a forwarding device's processing logic is defined. The matching **tables** and associated **actions** cited earlier from the P4 paper are all defined inside control blocks.

Those blocks may declare tables that match a *key*, which could have been extracted from the packet headers in a previous parser block. The logic in these control blocks may also be defined with control structures such as conditional statements. Usually, hardware targets implement all control structures as tables in the compiled code, so these structures are typically translated to tables by the compilers.

Figure 2.7: Example of a control block in the P4 language and its associated DAG.



Source: The Author

The control logic of these blocks, similarly to parser blocks, can be viewed as a DAG with *start* and *end* states. The language does not allow loops in the processing logic. It is designed this way so that all instructions may be translated to a pipeline structure of table matching and actions to be implemented in the programmable hardware so that it can guarantee high-speed processing and throughput. From the DAG, the compiler extracts the maximum time that a single packet will take to go through this pipeline. Therefore, it can emit a warning or prevent compilation if it detects that the hardware will not be able to accommodate the structures without compromising throughput guarantees.

Figure 2.7 shows an example of a control block defined in P4 code and a possible DAG that represents this block. In the DAG, green blocks represent tables with their match keys, and white blocks represent actions (comprised of statements). As previously explained, the *if* statements are typically implemented with static tables that match the condition. Table keys can have different match types, such as *exact* and *longest prefix matching* (lpm) depicted in the figure.

Each table has a maximum number of entries that are either statically defined in the P4 code or dynamically inserted during runtime. In the example of Figure 2.7, the first table (derived from the *if* statement) contains static entries for *True* and *False*. For the second table, a controller is responsible for inserting entries at runtime, which specifies the key to be matched and the appropriate action. An example of a rule that could be inserted in this table to match packets that have a "10.0.1.X" 24-bit IPv4 prefix and forward them to port 3 would be:

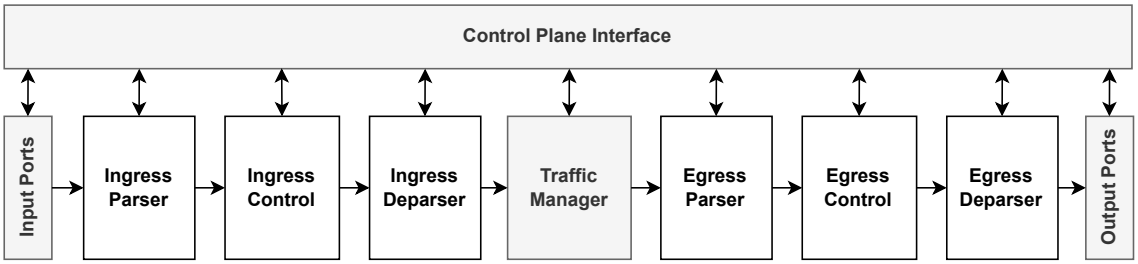
$$key = (10.0.1.0, 24), action = (forward, 3)$$

Tables also specify a *default* action, which is taken in case no entry is matched (that is, a table-miss). In the example of Figure 2.7, the *lpm* table defines "drop" as its default action, which simply drops the packets.

Each forwarding device's architecture also imposes restrictions on the set of operations that the control blocks are able to perform. As an example, the *v1model* architecture specifies that 5 control blocks are used to compose the complete switch program. Blocks 1 and 4 implement checksum verification and computation; therefore, they are only allowed to call functions that verify or compute the checksums on these blocks. Block 2 and 3 are the *ingress* and *egress* control blocks, and the programmer can freely define their behavior. Block 5 is a *deparser*, which re-assembles the packet's headers that were pre-

viously extracted or inserts additional headers according to the device's operation logic; thus it is only allowed to call the *emit* function.

Figure 2.8: Condensed view of the Tofino Native Architecture.



Source: The Author, adapted from (Intel, 2021).

The ingress and egress control structure of the *v1model* architecture is typically used by architecture that employs a non-programmable processing block between the P4 blocks, usually called a *traffic manager*. As another example, in the Tofino Native Architecture, the packet processing is divided into 6 programmable P4 blocks: 3 ingress blocks, the traffic manager, and 3 egress blocks. This schema is depicted in Figure 2.8. The traffic manager implements multiple functionalities, including packet replication, control plane forwarding, multicasting, and egress queue scheduling, among others. Depending on the architecture, the control plane can also manage these non-programmable elements during runtime, such as in the TNA shown in the figure.

2.5 Modular Data Plane Programming

As previously mentioned, the P4 language is the market's most widely adopted PDP programming language. Even so, it is not without its limitations in terms of code development design patterns.

P4 programs are monolithic by nature, offering little flexibility for the developers to separate functionalities into modules. The programs are wholly defined in a single translation unit. That is, the compilation process generates a binary configuration file from a single input "file" after pre-processing directives. No linking is performed across multiple "object" files or other pre-compiled libraries.

One might want, for instance, to develop a P4 "module" that handles firewall management, and that could be composed with another "module" that performs NAT. This modular structure allows for better reuse of program code when operators want to change

the components of their forwarding devices. The only standard way to implement this sort of structure is with “include” pre-processor directives. Large programs that implement multiple network functionalities, such as *fabric.p4* (ONOS, 2021), heavily utilize these directives in a tightly coupled manner to compose these functionalities, making it difficult to reuse components in other programs.

Additionally, C pre-processor directives are also overused in large programs to manage target-specific versions of these programs. In fact, the P4 files that define the *v1model* architecture utilize multiple “ifdef” directives to perform conditional compilation based on the version supported by each compiler. While these structures are suited for minor compilation control, their heavy usage is known to be error-prone and lead to difficult-to-track-down bugs (ERNST; BADROS; NOTKIN, 2002).

Higher-level languages were designed to address these limitations. Two of the most prominent languages are MicroP4 (SONI et al., 2020) and Lyra (GAO et al., 2020). Essentially, these languages allow for the definition of higher-level constructs across multiple translation units to achieve a modular programming paradigm for PDPs. Figure 2.9 shows an example of excerpts of code of the MicroP4 (left) and Lyra (right) languages. MicroP4’s syntax is noticeably similar to P4, while Lyra aims to be even more decoupled from a specific DSL.

Figure 2.9: MicroP4 and Lyra code examples.

MicroP4

```
L3 (pkt p, im_t im, out bit<16> nh, inout bit<16> type);

program ModularRouter : implements Unicast<> {
  parser P(extractor ex , pkt p, out hdr_t h) {
    state start {
      ex.extract (p, h.eth);
      transition accept;
    }
  }

  control C( pkt p , inout hdr_t h , im_t im ) {
    // p is partial pkt without L2 header
    bit<16> nh;
    L3() l3_i;
    action drop () {}
    action forward (bit<48> dmac, bit<48>smac, bit<8> port) {
      h.eth.dstMac = dmac;
      h.eth.srcMac = smac;
      im.set_out_port(port); //setting metadata
    }
    table forward_tbl { // L2 forwarding , needs nh
      key = { nh : exact; }
      actions = { forward; drop; }
    }
    apply { //invoke L3 to get nh
        l3_i.apply (p , im , nh , h.eth.etherType );
        forward_tbl.apply(); // use nh in forward_tbl
    }
  }
  control D( emitter em , pkt p , in hdr_t h) {
    apply { em . emit (p , h. eth ); }
  }
}
ModularRouter (P, C, D) main;
```

Lyra

```
>HEADERS:
  header_type int_probe_hdr_t { // Define header type
    bit[8] hop_count;
  }

  packet in_pkt { fields { ... } }

>PIPELINES:
  pipeline[INT]{int_in -> int_transit -> int_out};

  algorithm int_in { // Ingress INT
    global bit[32][1024] packet_counter; // global variable
    int_filtering();
    if (int_enable) {
      add_int_probe_header();
      add_int_md_hdr();
    }
  }
  algorithm int_transit { ... } // Transit INT
  algorithm int_out { ... } // Engress INT

>FUNCTIONS:
  func add_int_md_hdr() {
    extern dict<bit[8] msg_type, bit[30] switch_id>[128] add_int_md_hdr_filter;
    if (int_probe_hdr.msg_type in add_int_md_hdr_filter) {
      add_header(int_md_hdr);int_md.queue_len = get_queue_len();
      int_info(int_info);
    }
  }
```

Source: The Author, adapted from Soni et al. (2020) and Gao et al. (2020).

The catch is that no target supports these languages directly, as they were not designed for this purpose. Both MicroP4 and Lyra "compile" their source codes to various target-specific P4 source codes. These P4 codes can, in turn, be compiled by the vendor's compilers to generate the binary configuration files for the devices. This pipeline allows for MicroP4 and Lyra to be used with hardware devices that operate with the P4 language. Actually, Lyra has compilers that are able to generate code for other DSLs such as NPL (BROADCOM, 2023), a vendor DSL from Broadcom that has its fair share of hardware targets such as Broadcom's Trident-4.

There are works that aim to provide modular data plane programming directly through P4 code composition. PRIME (PARIZOTTO et al., 2020) defines a mechanism to compose modules, which are complete P4 programs, into a single program that is deployed in the target device. During runtime, the operator can define an order in which the modules are applied to each packet according to an identifier (e.g. TCP 5-tuple). The packets are recirculated through the device once for each module that must be applied.

The goal of this kind of high-level abstractions (Lyra, MicroP4, PRIME...) is similar to PDP virtualization (which will be introduced later in Section 2.8), and that is the primary object of study of this thesis: providing greater flexibility over the operator's hardware. Here, the focus is on the programmer of the device's behavior, whereas in virtualization, the focus lies on the flexibility of the hardware to implement multiple logical devices simultaneously. Combining these two concepts allows for an end-to-end solution that can be completely agnostic of target and even language-specific details (using high-level languages) and efficiently and simultaneously utilize this hardware for different purposes (virtualization).

Additionally, the architecture described in this thesis enables modular dataplane programming via PDP virtualization. In contrast with MicroP4 and Lyra, this is achieved with an extension to the P4 programming language, without the need to define a novel PDP DSL. In contrast to PRIME, the approach presented in this thesis does not enforce a module application order and does not require packet recirculation. This feature is explored in Chapter 5.

2.6 PDP Runtime Interface

Forwarding devices deployed in the SDN data plane, whether programmable or not, must provide an interface for the control plane to configure and manage them during

operation. This involves straightforward actions such as inserting table entries in a forwarding table, as previously mentioned in Section 2.4.3, to more complex actions such as configuring priority queues in the traffic manager or setting up the multicast rules for packet replication.

This notion was previously shown in Figure 2.2 with the Southbound Interface (or CDPI). This interface is usually implemented by protocols such as OpenFlow on a non-programmable data plane. PDPs, on the other hand, require a more flexible interface due to the flexibility of the behavior of the devices themselves.

There are different concrete implementations of the CDPI for programmable devices depending on the specific device model. Most P4 targets implement P4Runtime (The P4.org API Working Group, 2019), a standard interface designed and maintained by the *P4 API Working Group*. This interface allows for the configuration of the elements of the deployed switch during its operation, such as inserting a matching rule in a match-action table, for instance.

The advantage of P4Runtime is that its open specification allows for the reuse of control plane software regardless of the specific targets, so long as they implement this open interface. Nevertheless, it is not uncommon for targets to implement P4Runtime along with another target-dependent interface. On the Tofino switches from Intel, for example, both the open P4Runtime interface and a proprietary BFRuntime interface are implemented. The advantage of the proprietary interfaces is that they can be used to access components of the forwarding device that are not generic enough to be contemplated in the P4Runtime specification, such as the traffic manager or the physical port configurations.

Since the goal of this thesis is to provide a generic architecture for programmable forwarding device virtualization, the rest of this Section will describe in greater detail the workings of the P4Runtime interface, as it is a widely adopted interface among different hardware targets. The reference implementation of the architecture of this thesis is targeted for devices that implement it[6]. Still, the architecture described in the following Chapters of this thesis is not tightly coupled to the P4Runtime interface, as it can be interchanged according to the operator's needs.

It is also worth reinforcing that, even though the name might suggest it, P4Runtime is not exclusive to targets that have their forwarding behavior described by a program written in the P4 language. It utilizes concepts that are indeed elements of a P4 program, such

[6]The architecture is not limited to a specific implementation. For hardware experiments, support for the BFRuntime CDPI was also implemented. These additional implementations are not discussed in detail in this thesis.

as tables and actions, but these concepts are mostly generic to forwarding devices. If another DSL for PDPs implements similar concepts, and if the vendors of the targets of such DSLs want to support P4Runtime, it is certainly possible. In fact, even non-programmable data plane devices may support P4Runtime to interface with the non-programmable data plane-elements, such as fixed tables and registers.

P4Runtime uses gRPC, an open-source Remote Procedure Call (RPC) framework from Google (GOOGLE, 2015). Essentially, RPC is a strategy to implement a client-server inter-process communication between processes that may be executing on different machines. The clients interact with the server through procedure (function) calls implemented by it using a layer of abstraction provided by the framework. RPCs are a thoroughly studied paradigm that stood the test of time, being discussed as far back as the 1970s (WHITE, 1976).

One of the main advantages of gRPC is that it can be integrated with many different programming languages. The "services" (i.e., the remote procedures, arguments, and return values) are defined in the Protocol Buffers description language, and a transpiler converts them to the language of choice of the service programmer. At the moment of writing this thesis, gRPC has official support for 11 programming languages, including C++, Java, and Python, among others.

This flexibility allows for the decoupling of the implementations of the client and the server of the services. P4Runtime, for instance, defines its specification in a gRPC description language file[7], and the programmable forwarding devices' vendors implement a server based on this description using the language of their choice. The SDN controllers can, in turn, implement clients using their own language of choice. The RPC framework interface enables all of the servers from different vendors to be compatible with all the clients from different controllers.

P4Runtime defines a series of RPCs the controllers may call to interact with the data plane. These are:

- **Write:** Allows to update an entity in the target. This includes inserting table entries and updating registers.

- **Read:** Returns the values of entities in the target. Similar to write, table entries and registers can be read.

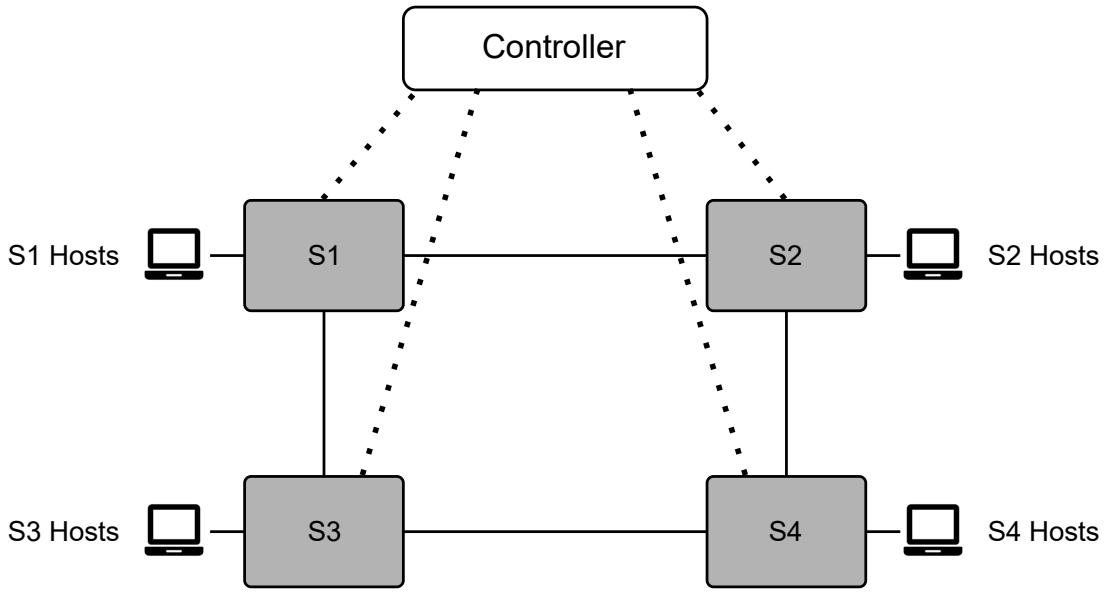
[7]The protocol buffers description of P4Runtime is available at: <https://github.com/p4lang/p4runtime/blob/main/proto/p4/v1/p4runtime.proto>

- **SetForwardingPipelineConfig:** P4Runtime provides this powerful RPC that can change the "program" that is implemented in the forwarding device. The argument to this RPC is target-dependent, but it is typically the artifacts generated by the P4 compiler. In the bmv2 (behavioral-model version 2) software switch, for instance, the output of the compilation process is a JSON file.

- **GetForwardingPipelineConfig:** This RPC returns a description of the current implemented program. The values are usually the same as the ones used as arguments to the SetForwardingPipelineConfig RPC.

- **StreamChannel:** This is a special RPC, as it opens a bi-directional persistent channel between the client and the server. It provides authentication, packet I/O (exchange of network packets between the data plane and control plane), and other notifications, such as table timeout.

Figure 2.10: PDP runtime example scenario.



Source: The Author.

As an example of a scenario that uses most of the RPCs defined in the specification, except for the program configuration ones, consider the topology depicted in Figure 2.10. S1-4 are programmable switches that can implement, for instance, IPv4-based L3 forwarding. To manage these devices, the controller would start multiple connections to the P4Runtime servers running on each switch and open a **StreamChannel** to send an authentication message.

Considering that, initially, the IPv4 tables on each device would be empty, all of the packets would result in table misses. A possible approach to establishing the necessary connections between the network hosts would be to program the devices (in the P4 code) to send packets to the controller if they result in table misses for them to be analyzed. Essentially, (i) the table-miss packets are sent with the packet I/O functions of the **StreamChannel** RPC, (ii) the controller executes an algorithm that determines a series of table rules to be inserted on the devices with the **Write** RPC, (iii) and the packets are sent back from the controller to be forwarded appropriately. This approach is known as *reactive forwarding*. Its main advantage is that it allows for the tables implemented in the devices to be smaller since only entries related to active flows need to be kept. Inactive entries that did not match any packets in a determined time interval can be removed (notifications for these entries are another feature of P4Runtime).

With PDPs, it is also possible to implement more flexible and sophisticated routing algorithms than the classical link-state ones such as OSPF (MOY, 1998). Intuitively, flows would be routed in the network through the shortest paths in terms of number of hops (links). It is possible, though, to use registers of the programmable forwarding devices to keep track of the link usage in the network. This allows for the controller to use this additional information obtained with the **Read** RPC to avoid overloaded links. For example, if multiple hosts connected to S1's subnet want to communicate with hosts from S2's subnet, the link between these switches may be saturated. The controller could instead redirect flows using the routing S1-3-4-2 to avoid this link.

2.7 In-network Computing

In the context of SDNs, two concepts that are closely related, even though they are antagonistic, are *Network Function Virtualization* and *In-network Computing* (INC). These concepts are enabled by the aforementioned decoupling of the processing logic of the forwarding devices from their hardware, which provides a greater degree of flexibility on how (and where) to implement network services.

As described in Section 2.2, NFV's main idea is to offload part of the network's core processing tasks to commodity servers. The main reasons for this decision are (i) to give operators better control of those functionalities that are traditionally hardcoded in the network devices' hardware and (ii) to reduce the use of different middleboxes in the system and provide uniform service-chaining.

The inverse can also be attractive in some scenarios. In some cases, it would be interesting to offload services performed by commodity servers to the forwarding devices and take advantage of their increasing processing capabilities. For instance, the network devices between the hosts and their first-layer DNS provider could perform DNS caching directly to speed up the queries (DANG, 2019).

This approach is enabled by PDPs, as they provide the freedom for the operators to define the algorithms executed by the devices. Modern hardware allows for those arbitrary functionalities to be executed at a line rate without penalties to the device's throughput and latency performance. Kianpisheh and Taleb (2023) provide a recent survey on the many applications of INC, including the aforementioned DNS example, distributed consensus in the data plane (WOODRUFF, 2020), smart data storage (TOKUSASHI; MATSUTANI; ZILBERMAN, 2018), among others.

Naturally, the development of those solutions calls for robust testbeds that ascertain their viability in large-scale production environments. Research institutions, such as universities, have a limited budget for hardware devices, and it is not uncommon for research groups to share those resources. Virtualization of PDPs, introduced in the next Section (2.8), is a possible solution to speed up the development of INC functionalities. The contributions of this thesis include a multi-tenant solution that enables INC testing in parallel, described in Chapter 4.

2.8 Virtualization in PDPs

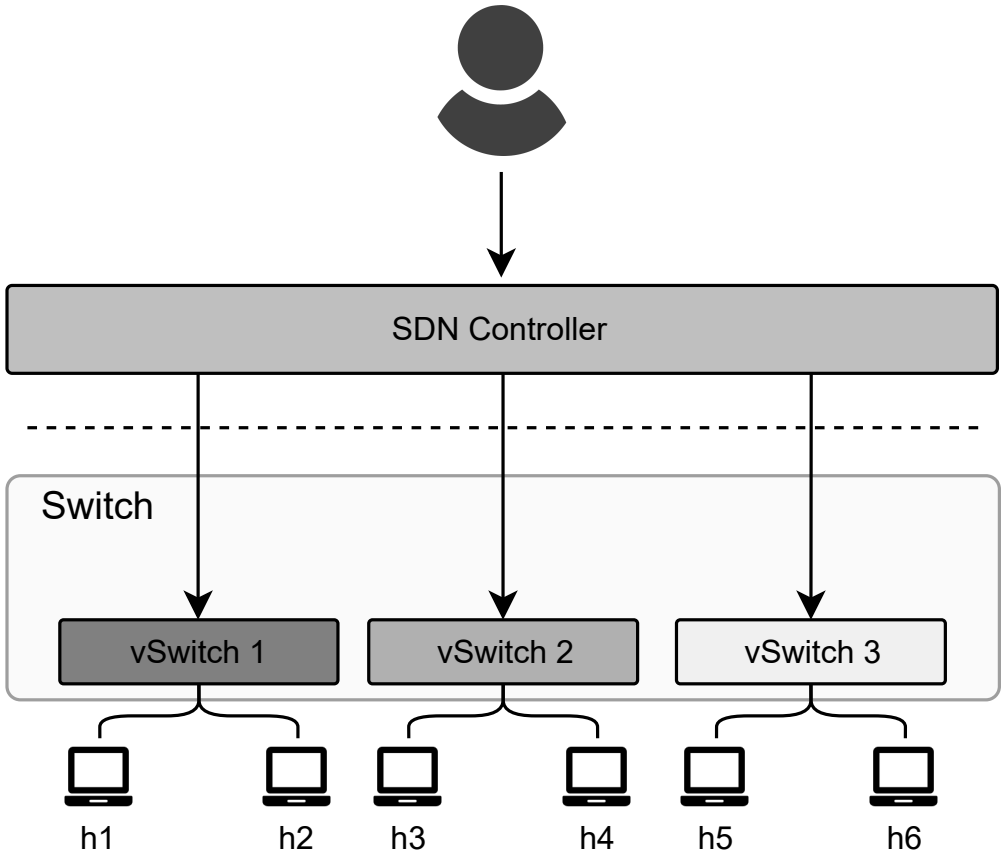
Virtualization is a technique to split a resource (in most cases, hardware) of a host between multiple tenants (guests). It is a broadly discussed topic by many authors. Tanaembaum and Bos (2014), for instance, lists some of the advantages of Virtual Machine[8] (VM) virtualization, such as service isolation to increase fault tolerance without incurring additional costs of independent hardware. An entity called "hypervisor" is responsible for this division of the host's resources between the guests.

A couple of years after the introduction of the P4 language, researchers started to study the possibility of applying virtualization in the context of PDPs (HANCOCK; MERWE, 2016). The main idea is that multiple devices, described by different P4 codes, could "run" in the same physical target simultaneously. This would be equivalent to a

[8] One of the most common types of virtualization employed are Virtual Machines, in which many logical hosts may operate sharing the same physical hardware.

conventional server running multiple isolated VMs.

Figure 2.11: Programmable switch virtualization example.



Source: The Author.

Figure 2.11 depicts this core idea in a scenario with three virtual switches. A single data plane programmable target that is managed by the SDN controller can implement the behavior of multiple different devices. This way, it is possible to benefit from hardware slicing, saving resources that would be required to run a similar setup with multiple hardware targets.

The rest of this subsection describes in chronological order the initial efforts employed to enable this concept. Those works are essentially all proof-of-concepts that introduce the main ideas and provide incomplete implementations that are not intended to be maintained. In fact, most of the works cited in this section had their prototypes implemented in an outdated version of the P4 language, making it hard to reproduce them with the currently available hardware.

2.8.1 Hyper4

The first solid attempt to provide an environment for PDP virtualization was Hyper4 (HANCOCK; MERWE, 2016). It introduced the notion of multiple P4 programs running simultaneously in a single target, citing among other advantages:

- **Network slicing:** Just as the previously mentioned VMs slice their host server's hardware, Hyper4's virtual switches could be able to slice the targets.
- **Network snapshotting:** Multiple logical configurations of the network could be "stored" in a database that could be used to provide quick transition when required by the system operators.
- **Modular data plane development:** Similar to the higher level languages described in Section 2.5 that aims to provide modularity through code management, virtual switches could also be able to achieve it by means of composing multiple virtual devices under Hyper4's management.

The fundamental idea of Hyper4 is to have a **generic P4 program** (called *hypervisor*) that can be configured at runtime to implement the behavior of any other P4 program. In fact, this hypervisor program would be able to emulate the workings of multiple P4 programs at the same time.

This would be achieved by defining a series of carefully designed tables in the hypervisor that had actions that implement the operations of the P4 language, from simple addition to conditional statements and table indexing. The hypervisor would then be compiled and deployed on the programmable device, and be ready to be configured to implement other programs.

To implement the virtual devices, an SDN controller would take any P4 program "P" and, through a runtime interface, would insert a set of carefully formulated table entries based on "P" on the hypervisor and that would result, from an outside perspective, in the same behavior as if the target were running program "P". Additionally, as previously mentioned, it would also be possible to repeat this process for other programs that would run in parallel by adding additional table rules on the hypervisor. Naturally, the hypervisor should allocate those tables with the largest possible size supported by the hardware in order to accommodate the maximum amount of virtual devices at the same time.

Moreover, the programs implemented through the hypervisor would naturally need runtime management from the control plane for updating the table rules, setting values to

registers, and other functionalities. Those elements of the virtual programs (tables, registers...), however, are not actually defined in the program that is running in the target (the hypervisor), so a schema for "translating" runtime requests from the control plane crafted for the virtual devices to requests to the hypervisor would also be required.

So far, the proposed solution could be viewed as ambitious, but nothing absurd. Unfortunately, it is not clearly specified how a concrete implementation of those modules could be achieved, and as of the moment of the writing of this thesis, Hyper4's public repository has not had any updates for over 7 years. The proof-of-concept implementation of the authors uses the following workarounds to achieve a minimal functional solution:

- **P4 Hypervisor:** The code of the hypervisor, written in $P4_{14}$, is not complete as to allow any program to be emulated through its tables. The program can only be used to implement programs with a small subset of the P4 language features, such as the ones used in Hyper4's evaluation (layer 2 forwarding, a firewall, and a router).
- **P4 Programs Translation:** As mentioned, a P4 program must be "translated" to table entries that are inserted in the hypervisor so that its behavior can be emulated. It is not trivial to design a tool that performs this process. For the proof-of-concept of Hyper4, manually crafted entries are inserted by the authors.
- **Runtime Management of the P4 virtual devices:** Yet another challenge left for future works is the management of virtual switches through a southbound interface such as P4Runtime. The examples used in the proof-of-concept do not require runtime management.

The results in terms of performance are also another downside discussed by the authors, with experiments that show that the throughput and latency of Hyper4 is 80% to 90% worse than the "native" P4 code deployed without the virtualization solution. Hyper4 was not evaluated or tested in real hardware, only in a software switch known as *bmv2* - Behavioral Model Version 2 (P4LANG-CONSORTIUM et al., 2019). This software device is intended to be used as a quick prototyping testbed for P4 programs. Even with an environment not suited for performance evaluation, the authors justify that the analysis could be extended to real hardware, which is largely controversial.

In summary, Hyper4 was paramount to introduce the first concepts that allow for the design of a PDP virtualization solution, but it lacked a concrete implementation of those concepts to achieve a broader audience. These missing key points need to be addressed by later solutions to increase the adoption of the technology.

2.8.2 HyperVDP

Following the work of Hyper4, another group of researchers took an interest in PDP virtualization and its limitations at the time, particularly the performance and hardware usage inefficiency. Their solution, HyperVDP (ZHANG et al., 2019), follows the same strategy as Hyper4, which leverages a hypervisor P4 program that can be configured in runtime to mimic the behavior of several other P4 programs.

Using clever optimizations of the hypervisor, the authors were able to achieve a better performance than its predecessor. Still, their implementation is heavily target-dependent on the bmv2 software switch, which is the platform used for their evaluation. As it was stated, bmv2 is not an appropriate platform for evaluating the performance of these kinds of solutions. One cannot expect the real forwarding device's hardware to behave the same way as the CPU-based prototyping switch. Current high-end P4 hardware, such as the Tofino switches, will not even allow for programs to be loaded if it is not possible to guarantee its advertised throughput and latency.

Unfortunately, it is also not possible to reproduce the implementation of HyperVDP on modern hardware, as the tool needed to convert arbitrary P4 programs to table rules of the hypervisor was never publicly available. Moreover, the hypervisor program was implemented in the outdated $P4_{14}$ version of the P4 language.

2.8.3 P4Visor

The initial approaches to PDP virtualization, Hyper4 and HyperVDP, were based on the same underlying technology: **emulation**. The goal of the approach was to provide flexibility in exchange for hardware efficiency and network performance. Alas, that design proved to be unscalable for real hardware targets, where efficiency is paramount.

With the lack of viable compromise, it became clear that a different virtualization technology was needed to overcome the limitations of the time. The next big advancement of the area came with a novel approach: **composition**. P4Visor (ZHENG; BENSON; HU, 2018) was the work that introduced this new concept.

The main idea of P4Visor and composition-based PDP virtualization is to perform most of the work at compilation time, in contrast with the runtime-focused paradigm of emulation-based approaches. Essentially, composition solutions take advantage of the observation that multiple P4 programs can be "merged" into a single program that imple-

ments the behavior of the composed programs. Afterward, the resulting P4 program goes through the target's compilation process just as a "normal" P4 program to be deployed.

This eliminates the need for the P4 program translator discussed in Hyper4's design requirements (Section 2.8.1), which converts whole P4 programs into a set of table rules for the hypervisor. As stated, it is not trivial to implement this module, as it requires a profound knowledge of the inner structures of the hypervisor program. In fact, the module was manually simulated for the evaluation of both Hyper4 and HyperVDP proof-of-concept implementations.

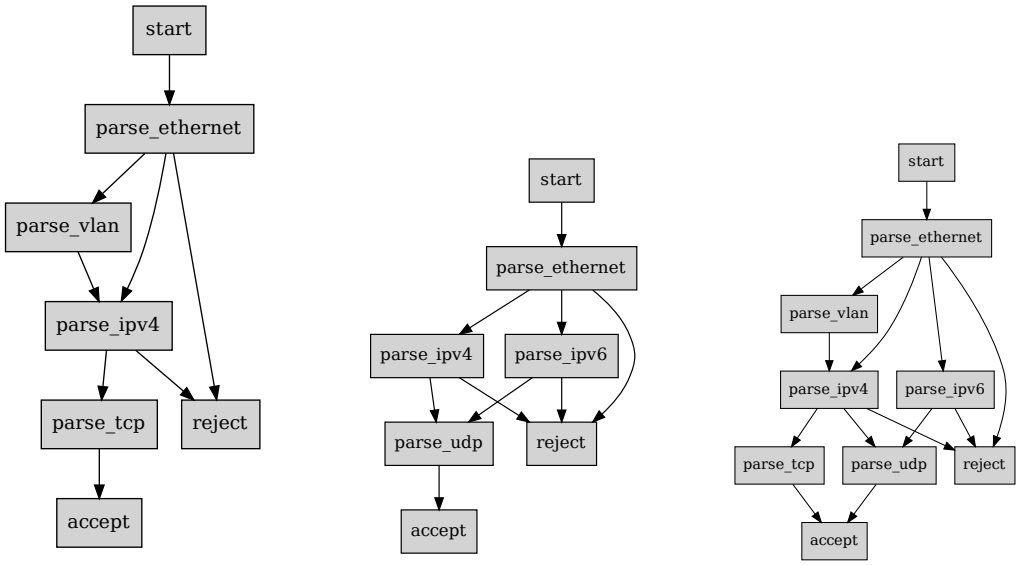
The other modules discussed are still required for a composition-based solution, but their design can be simplified. The "hypervisor" can be viewed as a combination of the composition process and the merged program. A runtime module is still needed to translate requests, which are crafted for the virtual devices. More details about the implementation of those modules are discussed in Chapter 3 of this thesis, which presents a composition-based virtualization architecture.

P4Visor specifically is heavily focused on P4 program debugging and testing. It was designed to merge different versions of the same P4 program. The main use case is to take a current, stable version of a program and a test version with minor differences and compose them into a single program that can behave as both. P4Visor provides a framework that marks packets to define which of the virtual programs is going to process them, and testing primitives to compare the results of the processing of those programs.

This use case scenario, where the programs are mostly similar due to being different versions of the same processing logic, provides opportunities for merging optimizations. Similar or even equal structures, such as table definitions, can be unified in the resulting P4 program. This approach is known as **lightweight virtualization**, as the virtual devices are logically isolated from an outside view, but the internal structure executing in the hardware is shared. It contrasts with the previous emulation-based approaches that implement **hard virtualization**, where the programs are fully independent and do not share those resources.

Lightweight virtualization leads to better hardware usage efficiency. Figure 2.12 depicts an example of the merging process of two parser blocks of different P4 programs. The figure shows the resulting DAGs that represent parsers of two different P4 programs ("A" and "B") and also the DAG of a merged program. The first program (A) processes headers of the Ethernet, VLAN, IPv4 and TCP protocols, and the second program (B) processes headers of the Ethernet, IPv4, IPv6 and UDP protocols. Considering that the parser

Figure 2.12: Example of the lightweight merging of two parsers.



(a) Program A. (b) Program B. (c) Merged Program.

Source: The Author.

stages of each of those protocols represented by the nodes of the DAGs are mostly similar when processing the same headers, the merging process can unify them in the resulting program. The merged program contains the processing logic for all of the protocols of the input programs taking advantage of the unification opportunities.

P4Visor was evaluated in a real hardware target, ONetSwitch, which implemented a subset of the P4 language features at the time. The authors were able to achieve up to 5Gbps of aggregated throughput on the tested programs, which was a significant improvement compared to the results of the emulation-based approaches in the order of Mbps.

The main tradeoff of the composition-based approach, in theory, is the loss of flexibility provided by the emulation approaches in terms of virtual switch deployment. In that paradigm, since the insertion/removal of a virtual device is essentially the insertion/removal of table entries, the devices can be inserted/removed from the hypervisor in runtime without disrupting the processing of the other virtual switches. This is not possible in composition-based approaches. To insert a new virtual switch in the target, it is necessary to generate a new merged program that includes this new virtual switch and reconfigure the whole forwarding pipeline in the target. This briefly disrupts the operation of all virtual devices and, more critically, destroys the stateful memory (registers and table entries) of those devices. P4Visor leaves those limitations for later works. A solution to

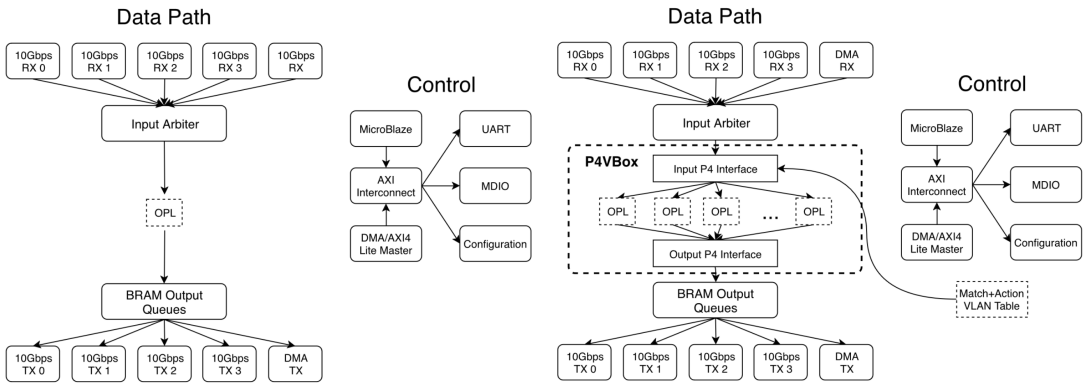
diminish the impacts of the redeployment of the merged program based on state migration (LUO; YU; VANBEVER, 2017) is discussed in Chapter 3 of this thesis.

2.8.4 P4VBox

A different approach to achieve PDP virtualization, leveraging modern field programmable gate array (FPGA) hardware designed for network scenarios, was introduced with P4VBox (Saquetti et al., 2020). It is an alternative to the emulation and composition mechanisms that take advantage of the underlying hardware support.

It is designed based on the NetFPGA-SUME boards from Xilinx, which started to support P4 high-level synthesis (HLS) in addition to other hardware description languages (HDL) such as Verilog and VHDL in the late 2010s (IBANEZ et al., 2019). Figure 2.13 shows the canonical NetFPGA reference design (a) and how P4VBox is inserted on it (b). The canonical design (a) is composed of input structures (ports and Input Arbiter), output structures (BRAM Output Queues and ports), and an Output Port Lookup (OPL) that is responsible for the implementation of the processing logic of the packets. The authors modified the P4-NetFPGA HDL codes to implement P4VBox (b) by replacing the OPL. P4VBox allows for as many OPLs (which represent the virtual switches) to be deployed as there are available hardware resources. Input and Output P4 Interfaces were also added to determine which OPL receives the incoming packets following a certain scheme.

Figure 2.13: Comparison of the canonical NetFPGA and P4VBox designs.



(a) Canonical NetFPGA design. (b) P4VBox design.

Source: Saquetti et al. (2020).

For a virtual switch to be implemented, its P4 code goes through an HLS process using the toolkit provided by Xilinx SDNet to generate configuration bitstreams. Those

bitstreams can, in turn, be loaded into P4VBox OPLs. This process allows the decoupling of the virtualization solution into separate modules.

The result is a powerful virtualization platform that is capable of achieving a competitive throughput (in the order of tens of Gbps) and latency (in the order of decimals of microseconds) while still retaining the flexibility associated with emulation-based approaches. P4VBox design makes it possible to deploy and remove virtual switches in runtime due to the hardware support for partial reconfiguration. This process does not disrupt the network flow of the other virtual devices.

Of course, the design of P4VBox makes it inherently target-dependent, as the hardware target is required to have an FPGA substrate. Moreover, the dependency of the Xilinx toolkit for the HLS process to generate the bitstreams aggravates this issue.

2.9 Multi-tenancy in PDPs

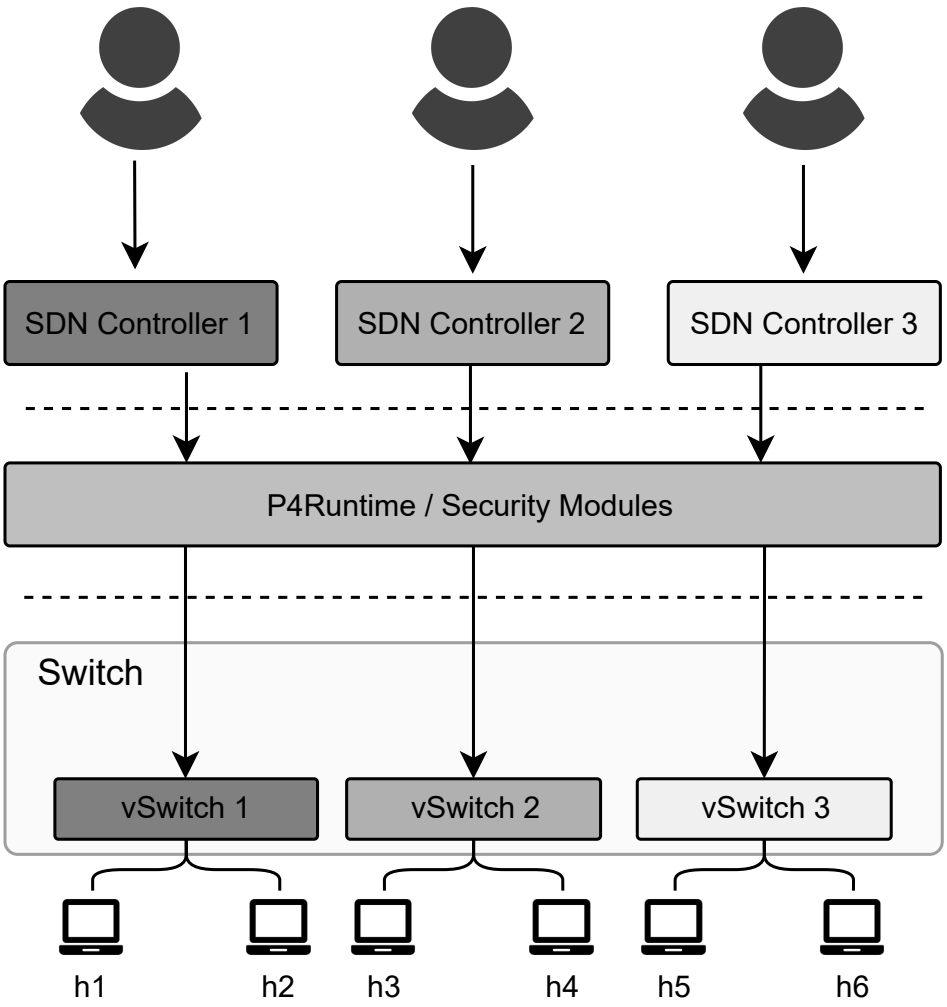
The previous Section focused on the initial efforts made to enable PDP virtualization in order to run multiple devices in parallel within the same target. After those initial advancements, it was possible to design solutions that would consider that the multiple virtual switches running simultaneously could belong to different *tenants*. That entails that not only would the switches be processing traffic independently, but they also must be logically isolated to the point of not interfering at all with the others during runtime.

As one can imagine, this area of research is still early in its development, with the earliest works starting in the 2020s. The published works that are described here mainly focus on the conceptual ideas about the restrictions that must be enforced to guarantee virtual switch security in an environment with multiple tenants.

The main concept of a multi-tenant PDP virtualization scheme is depicted in Figure 2.14. In contrast to the earlier single-tenant model shown in Figure 2.11, multiple controllers, owned by different tenants, are responsible for managing each virtual device. As such, additional modules, such as the runtime module shown in the picture, are necessary to orchestrate this kind of solution. Those modules' main goals are to provide management abstractions and security features.

It is critical to realize that many of the classical security challenges in multi-user environments have equivalents in PDP virtualization with multiple tenants. As an example, Cross-App Poisoning (CAP) attacks are a class of network attacks against programmable devices that involve privilege escalation (UJCICH et al., 2018). As a pre-

Figure 2.14: Multi-tenant switch virtualization example.



Source: The Author.

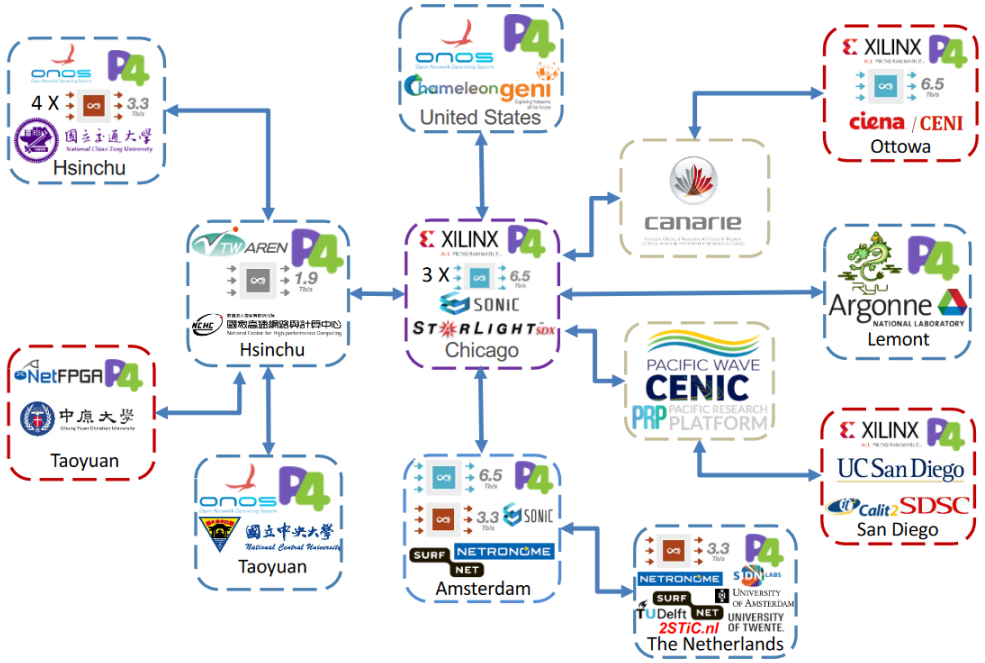
vious work of this thesis, it was shown that additional information flow control policies are necessary to protect virtual programmable switches from those kinds of attacks in a multi-tenant environment (LAMB et al., 2022).

Similarly to the previous section on virtualization, the rest of this section introduces the main initial works to enable some kind of multi-tenancy in PDPs. As of the moment of the writing of this thesis, there are few works that focus on this aspect of PDP virtualization due to the novelty of the concept. However, the potential of industry-level applications and market adoption is encouraging more work in this direction, setting up a promising field of research at present times.

2.9.1 P4MT

The authors of P4MT (CHUNG et al., 2021) recognize that prototyping and testing is a critical step in the development of innovative network protocols and services. They reinforce the importance of a complete testbed to ensure that most problems are identified and fixed before those innovations are deployed in production.

Figure 2.15: International P4 Experimental Networks.



Source: (CHUNG et al., 2021).

With the goal of increasing the support for this part of development, they proposed i-P4EN (International P4 Experimental Networks) shown in Figure 2.15. It is an international collaborative effort to create a P4 testbed in which multiple institutions can share hardware resources between their projects.

This collaborative infrastructure calls for a management solution that can deal with multiple tenants interacting with the system. The authors designed P4MT to be this management solution to run multiple P4 packet processing pipelines in parallel. Essentially, P4MT is able to allocate specific hardware pipelines for a tenant in devices that support multi-pipeline programmability (such as the Intel®Tofino™ family of switches, which has models that can support up to 4 different pipelines in parallel). This way, the logical isolation of the packet flows for each tenant is provided by the device's hardware itself, without the need for a sophisticated emulation or composition solution.

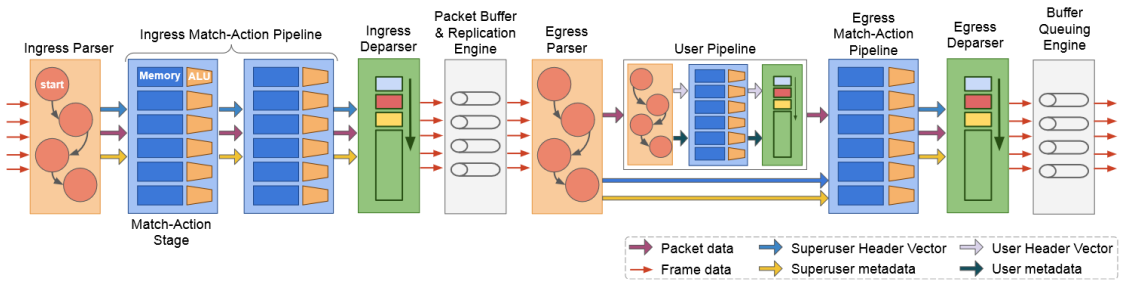
The limitation of this approach is the heavy target dependency of the solution. It needs hardware support for multiple pipelines to accommodate multiple tenants. This limitation would also imply that the maximum number of tenants supported by the system is the number of pipelines available in the target hardware. To circumvent this limitation, P4MT adopts a *shared table* design in which the NFVs cannot be freely designed by them. The tenants must choose from a set of available functions to compose their programs.

Also, P4MT is not an open-source solution, and many of the implementation details about how the composition of the multiple pipelines in a single P4 program is performed or how the P4 program handles Packet-In and Packet-Out metadata for each tenant are not discussed. As such, it is not possible to compare it to other management solutions.

2.9.2 MTPSA

Another work that aims to provide multi-tenant virtualization is MTPSA (Multi-Tenant Portable Switch Architecture) (STOYANOV; ZILBERMAN, 2020). The authors emphasize the need for programs belonging to different tenants to not interfere with each other's processing.

Figure 2.16: MTPSA Architecture.



Source: (STOYANOV; ZILBERMAN, 2020).

This architecture, however, is also heavily target-dependent. The authors's description of the architecture is heavily coupled with the reference software switch implementation of P4, bmv2 (which cannot achieve industry-level throughput), and with FPGA-based P4 switches from Xilinx, similar to P4VBox. In fact, their open-source implementation is completely focused on those FPGA's as the targets[9], with a limited demo also available for the bmv2 software switch.

Moreover, the architecture defines the entity of a "Superuser", as seen in Figure 2.16, which is a program that is appended at the ingress pipelines of the switches and

[9]Available at: <https://github.com/mtpsa>

defines which headers the users (i.e. tenants) programs are able to parse, such as Ethernet, IPv4, TCP, UDP, etc. This design decision limits the flexibility and creativity of user P4 programs. Ideally, the tenants should be able to **fully** design their programs without even considering the virtualization platform that it will run on.

2.9.3 SwitchVM

One of the most recent works towards multi-tenant data plane virtualization is SwitchVM (KHASHAB; RASHELBACH; SILBERSTEIN, 2024). The authors take a different approach compared to the other works by advocating for a separate language that serves as an abstraction between the user-defined processing logic and the hardware's native language (such as P4). A hypervisor manages this abstraction similarly to the emulation-based approaches.

The authors provide a quick survey of the previous works cited in this thesis, explaining the limitations of each approach with an emphasis on multi-tenancy. They mainly acknowledge that the earlier emulation-based approaches' advantage of being able to deploy and remove virtual switches in runtime was shadowed by the disadvantages of management and performance. As such, they propose an enhanced emulation-based approach that mixes static packet processing functions with user-defined functions with code that is encapsulated in the packet headers (similar to the concept of active networks (TENNENHOUSE et al., 1997)).

The main disadvantage of SwitchVM is the constraints imposed on the data-plane programming logic due to the designed programming language. The most notorious is the restriction of the set of protocols that the virtual switches must operate with (Ethernet, IP, etc.). As mentioned in Section 2.3, the flexibility for the users to define their protocols at will is one of the main motivations for the change towards PDPs. Additionally, a new programming language implies rewriting all of the code for existing algorithms to adapt to this kind of solution, as the authors were required to do for their experimental evaluation. Ideally, the same code that runs on a physical switch should be able to be deployed on a virtual switch. Moreover, the control plane code to configure them in runtime should also be the same.

Another considerably weak aspect of this solution is the lack of open-source code or any other material to enable the reproducibility of the results and insights of the prototype's implementation. It is impossible for independent researchers to fully grasp the

inner workings of the hypervisor without access to this material and to compare its performance to other similar solutions.

2.10 Summary of Related Works

After reviewing the main concepts of this thesis, mainly PDP virtualization, we can synthesize the main features of the presented PDP virtualization architectures. We compared the solutions considering the aspects of a complete multi-tenant virtualization architecture and the desirable features that it should provide, which are: (a) the virtualization scheme employed (e.g., emulation, composition, etc.); (b) the performance impact; (c) the possibility of swapping virtual devices in runtime; (d) the support for multi-tenancy; (e) the limitations of the P4 language imposed by the solution; (f) the availability of the source code or a prototype for independent testing/usage; and (g) the target-independency of the solution.

Table 2.1: Summary of PDP virtualization related works.

	Hyper4	**HyperVDP**	**P4Visor**	**P4VBOX**
a	Emulation	Emulation	Code Composition	FPGA Composition
b	High	High	Low	Low
c	Yes	Yes	No	Yes
d	No	No	No	Partial, no access control
e	P4 Language Subset	P4 Language Subset	P4 Language Subset	Hardware Area Constrain
f	Prototype (Old P4 version)	Prototype (Old P4 version)	Prototype (Old P4 version)	Available
g	Target-Dependent (bmv2)	Target-Dependent (bmv2 / DPDK)	Target-Dependent (bmv2 / ONetSwitch)	Target-Dependent (NetFPGA)

	P4MT	**MTPSA**	**SwitchVM**
a	Multiple Pipelines	FPGA Composition	Emulation
b	Low	Low	Low
c	No	Yes	Yes
d	Yes	Yes	Yes
e	Specific Functions	Specific Headers	Different Language
f	Not Available	Available	Not Available
g	Target-Dependent (Multi-Pipeline Support)	Target-Depedent (bmv2 / NetFPGA)	Target-Dependent (Intel Tofino 1)

Table 2.1 shows how each solution relates according to the listed characteristics.

One can observe that no single current solution contemplates all of the desirable features established for a complete virtualization architecture. Current architectures strike a balance of those features to produce a use-case-specific solution.

The earlier works, Hyper4 and HyperVDP, are not viable to be implemented on real hardware targets. Their virtualization scheme, emulation, induces a heavy memory and performance overhead, leading to a loss of throughput on the software switch in which they were tested. Additionally, even for software-based switch scenarios, the solutions only ever got to a prototyping phase without complete implementations due to the high complexity of the emulation paradigm. The main issues are the unclear program translation process and unspecified runtime management interfaces.

P4Visor made a breakthrough with the introduction of a different virtualization strategy and program composition, which led to significantly lower performance impacts and the possibility of hardware deployment. Still, it was focused on a specific use case (program version testing), which limited the P4 language features available. Moreover, the change in the virtualization strategy prohibits virtual switch swapping during runtime.

Yet another strategy was introduced with P4Vbox, composing the compiled virtual switch's FPGA configuration bitstreams directly in the target. It showed that it was possible to fully isolate the virtual devices directly in the hardware and allowed for agile swapping with partial FPGA reconfiguration. P4Vbox design opened up the possibility of high-performance multi-tenant virtualization solutions. Still, it is inherently target-dependent, relying on the NetFPGA's capabilities.

P4MT and MTPSA took steps in the direction of multi-tenant virtualization solutions. Both of them take advantage of target-specific hardware features. Still, those works are proof-of-concept architectures, as many gaps remain to be filled to achieve an industry-ready solution. For example, those solutions have no runtime modules to integrate with open CDPIs such as P4Runtime, which is implemented by most PDP controllers (ONOS, 2021). SwitchVM also advances the multi-tenant runtime deployment of virtual switches with a different approach to emulation but requires a completely different programming paradigm inspired by active networks and introducing a new programming language (requiring rewriting both data and control plane code).

Among the discussed works, none has a fully functional implementation available for any modern hardware. Hyper4, HyperVDP, and P4Visor's proof-of-concept implementations are designed for an older version of P4 tightly coupled with bmv2. P4VBox and MTPSA have available implementations for their target platform only, NetFPGA.

P4MT and SwitchVM have no publicly available implementations.

Finally, another interesting conclusion that we may come to is that a feature that was previously cited as a core design principle of the P4 language, target-independency, is not fully achieved by any existing solution. This drawback is intrinsic to the design of their architecture. They either consider a specific switch architecture (*v1model*, bmv2), modify the hardware design of a specific vendor (NetFPGA), or take advantage of features available only on certain targets (multiple pipelines).

3 MULTI-TENANT VIRTUALIZATION ARCHITECTURE

This Chapter describes the proposed virtualization architecture that is the core of this thesis and how it is able to go beyond the previous works to achieve a complete target-independent multi-tenant virtualization solution. It also gives insight into the current implementation of the architecture, publicly available on GitHub[1].

The results of a preliminary evaluation with two test cases are also discussed at the end of this Chapter. The implementation of the architecture was tested both in a software switch and a hardware switch to ascertain its target independence and competitive performance on terabits-per-second devices.

3.1 Design Goals

Currently, there is no readily available virtualization solution that can be deployed on an industry-level scale. That is mainly due to the previous works on virtualization described in Chapter 2 not being focused on the practical aspects of implementing a complete multi-tenant solution. They are heavily focused on scenarios where a single tenant would like to deploy multiple virtual switches under its own control on a single target.

The result is that those previous works focus on ideas about how to make the "core" of virtualization possible, such as the emulation of programs through a generic program or the composition of programs into a single program. However, none of those works cover the full picture. For instance, the runtime abstraction mechanism to interact with a virtual switch in an isolated manner is not described in any of the related works, and no source code of prototypes that works on real hardware is available. In fact, no solution has a complete source code available that operates on the newest version of P4 (P4 16), which is the one supported by most hardware P4 switches. Most of the prototypes source codes are also not updated since their initial commits.

As for multi-tenant PDP solutions, P4MT (CHUNG et al., 2021) does not support completely tenant-designed P4 programs. The tenants must choose from some predetermined NFVs to be deployed on their switches. Also, this solutions is neither target-independent or open-source, which makes it impossible to be evaluated and compared with other management solutions. In a way, this goes against the design principles of PDPs itself to eliminate network ossification by allowing network operators to freely de-

[1] Available at: <https://github.com/Ivanatorion/VirtualizationArchThesis>

termine the processing logic of their devices, but it is the current tradeoff for a multi-tenant data plane implementation.

The architecture described in this thesis is designed to tackle these problems and finally take the steps necessary to reach practical multi-tenant virtualization solutions, building upon the raw ideas described in the previous works. The main objective is for the virtualization solution to be the most transparent as possible to the tenants, mainly:

- **The architecture must be able to support an arbitrary number of tenants in an isolated and transparent manner:** That is, the architecture must provide each tenant the perception of the underlying system as it would if its virtual switch was actually a "real" programmable switch, and it must enforce that no tenant will be able to interfere with other tenant's switches in a harmful manner.

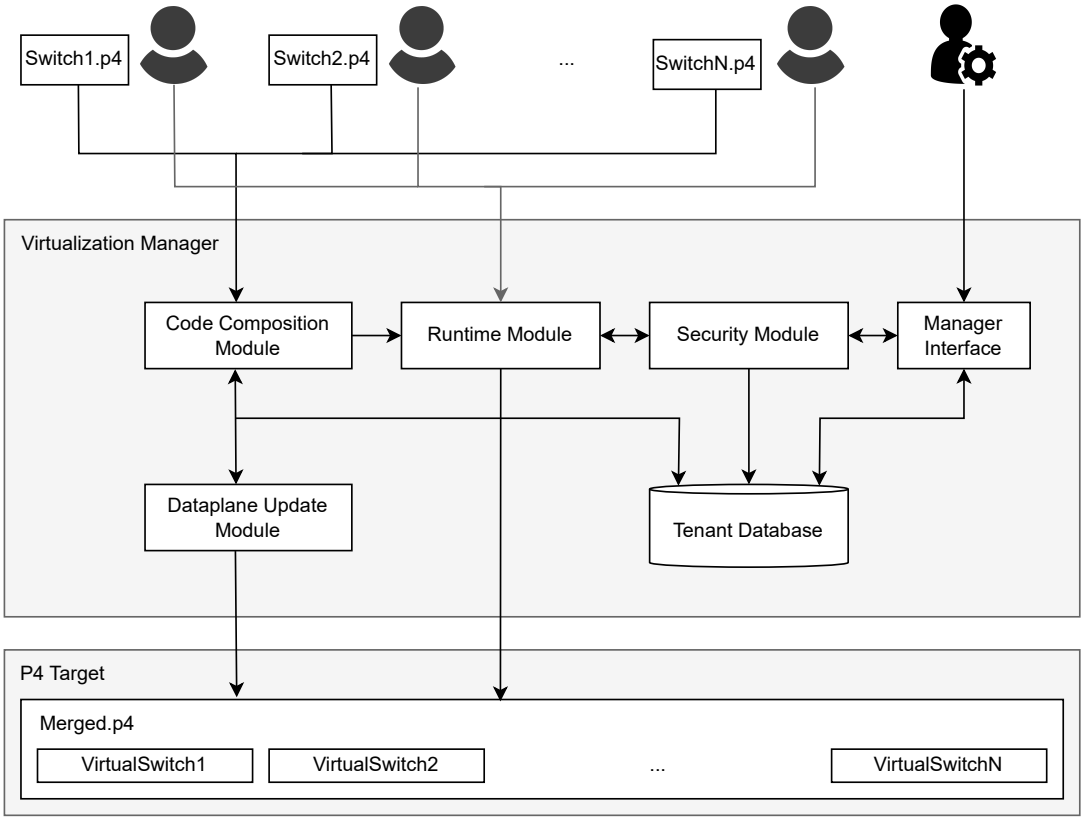
- **The tenants should create P4 code to their virtual switches as if it was a code meant to run on the real target:** That implies not restricting the P4 language to a subset of features whenever possible, taking into consideration the restrictions implied by the target's hardware itself.

- **The tenants should be able to deploy their code at will**: The same way that a tenant would go through the process of implementing its P4 code in its target (compilation and deployment), a virtualization architecture that aims to surpass its prototype phase must supply a clear chain to achieve this goal without major inconveniences (such as complex processes and high wait times).

- **The tenants should be able to transparently manage their virtual switches in runtime:** That is perhaps the most crucial point to the wide adoption of a virtualization solution. Every piece of application-plane and control-plane software developed to interact with programmable switches during runtime does so through standard interfaces (such as P4Runtime, or vendor-specific APIs). As such, the virtualization solution must have well-defined interfaces as well for the tenants and their applications to interact with their virtual switches.

To be clear, the goal of the architecture is to allow for a tenant that would normally deploy their program in a specific target "*X*" to deploy it on an implementation of the architecture running with the same target "*X*" as the host. Cross-target virtual switch deployment is trivial and is not in the scope of this thesis.

Figure 3.1: Virtualization solution architecture.



Source: The Author

Figure 3.1 illustrates the complete architecture of the virtualization solution. It allows for multiple tenants to implement a virtual switch on the system by uploading their own P4 programs to the code-composition module and the runtime module. The system administrator has an interface to populate the security policies and other data for each tenant registered in the system. The modules have the following functions:

(A) Code Composition Module: This module is responsible for composing multiple P4 programs belonging to different tenants. This module generates a new P4 program that implements the logic of all programs supplied in parallel. The module passes the composite program to the update module of the data plane and also generates additional information that will be used by the runtime module later. Section 3.2 describes this module in greater detail.

(B) Runtime Module: The virtualization solution should provide the vision to each tenant that their P4 program is the only program running on the target to ensure the isolation and security of the virtual switches. To achieve this, the Runtime module implements a translation layer for the specific CDPI of the target (e.g. P4Runtime) that behaves as if it were the CDPI implementation on the tenant's "target". In effect, the module intercepts

the messages sent to the target and modifies them to be compatible with the composite code that is actually executing on the target. This translation is performed using information about the shared elements in the composite code (provided by the composition module) and additional restrictions imposed by the security module. Section 3.3 goes into more detail on the workings of this module.

(C) Security Module: Responsible for ensuring that the tenant's requests comply with the security policies and for detecting other possible attacks on virtual switches. For example, this module can implement an Information Flow Control enforcement policy to mitigate Cross-App Poisoning attacks on the virtual switches (LAMB et al., 2022).

(D) Manager Interface: Interface for the system administrator to define the security policies and to populate the database containing the tenant's data.

(E) Data Plane Update Module: This module performs the implementation of the P4 program in target. This module simply "changes" the program that is implemented when requested, which involves procedures dependent on the target architecture, such as calls to specific compilers and drivers. Depending on the available resources of each system, this module can be expanded to allow for a more complex solution that is able to update the virtual switches without interrupting the target's operation, which involves disrupting the execution of all virtual devices. This thesis provides an approach to this problem, which provides each tenant with a logical view of an update while not disrupting the other tenant's traffic by employing multiple devices. This approach is described in Section 3.4.

(F) Tenant Database: Database containing tenant's information, such as available physical resources (number of ports, memory size, etc.), access permissions to entities of the composite program, among others.

Throughout the rest of this thesis, the *reference implementation* of the architecture and its modules will be cited when useful. Currently, the Code Composition, Runtime, Security, Data Plane Update, and Manager Interface modules are implemented. The modules were implemented in C++ with over 19000 lines of code. As mentioned, this implementation is provided publicly along with this thesis[2].

The remainder of this Chapter provides more details on the program composition strategy of the code composition module (3.2) and the abstraction during device operation provided by the runtime module (3.3).

[2]Available at: <https://github.com/Ivanatorion/VirtualizationArchThesis>

3.2 P4 Code Composition

The code composition module of the proposed architecture is the core of the virtualization solution. It is what enables multiple P4 programs to execute simultaneously on a single target in an isolated manner. The composition is inspired mainly by the strategies proposed by Lyra (GAO et al., 2020) and P4Visor (ZHENG; BENSON; HU, 2018).

As mentioned in Chapter 2, Lyra is a complete programming language that is designed to offer modularity to P4 programs, which are intrinsically monolithic. The Lyra framework is able to divide the programs into isolated modules (such as an IPv4 processing module and another module that processes ARP requests and replies, for example) and merge them into a single P4 program. However, the whole Lyra pipeline assumes a single tenant, that is, all of the modules are under the control of a single person or organization, the same for the resulting P4 program, and the same for whichever target they intend to deploy this P4 program. Similarly, but with the end goal of a multi-tenancy solution, the architecture described in this thesis focuses on how to take multiple complete P4 programs (that could be generated by Lyra or any other method) from different tenants and merge them together into a new P4 program before the deployment in the target.

From the mentioned works on code composition, PRIME is focused on modular data plane programming and packet steering through recirculations. Generating code compatible with controllers from different tenants at runtime while providing logical isolation is outside the scope of PRIME.

P4Visor, on the other hand, works in a similar way to the proposed architecture, merging two different P4 programs into a single new program. However, it has limitations that need to be addressed in a complete virtualization architecture, such as:

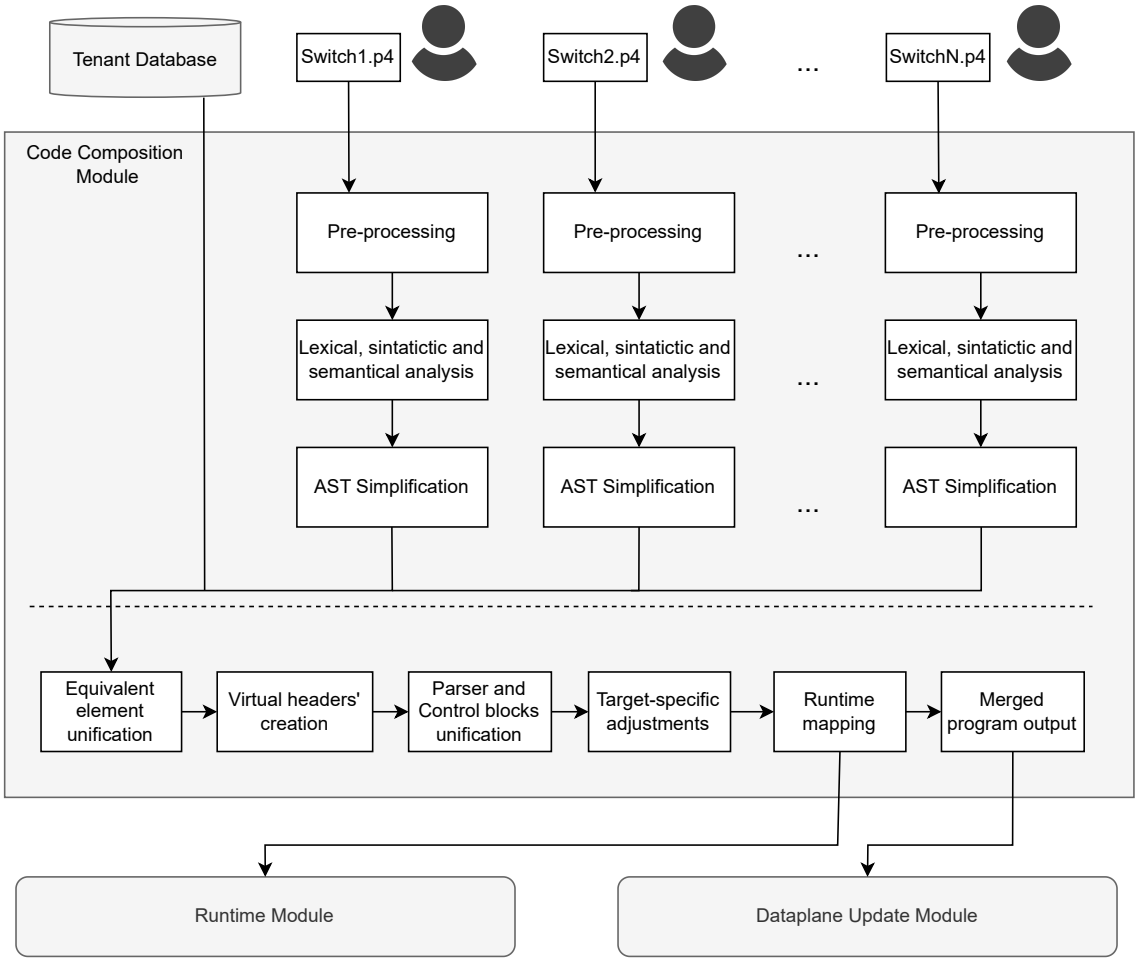
- **Merging completely different programs:** P4Visor mainly focuses on merging two different versions of the same P4 program for testing purposes. Merging programs from different tenants that might have entirely different processing pipelines is more challenging.

- **Merging the source code of the programs in a target-independent way:** P4Visor was designed to merge *json* representation of the P4 programs generated by the P4C compiler for the bmv2 software switch. This software switch is merely a prototype tool meant to emulate real hardware switches. For a solution to reach industry adoption, it must be target-independent so that it can be deployed in any hardware

switch with maximum performance.

- **Enforcing isolation and generating access information for runtime management:** Another consequence of the single-tenant design of the previous virtualization solution is the lack of proper isolation between the switches. A multi-tenant solution needs to isolate each virtual switch resource (tables, registers, variables, etc.) so that they retain ownership of those resources.

Figure 3.2: Code Composition Module breakdown.



Source: The Author

This Section explains in detail the workings of the code composition module and how it tackles the mentioned limitations of previous solutions. Figure 3.2 breaks down all of the submodules (or steps) involved in the process of merging multiple P4 programs. Those submodules are described in detail in the rest of this Section.

3.2.1 P4 Source Code Analysis

The main inputs from the tenants to the system are their virtual switch definitions in the form of raw P4 programs. By working directly with the source codes, we can leverage the target-independent properties of the P4 language in the virtualization architecture. This approach requires the code composition module to work on top of those codes itself, as opposed to higher-level intermediate representations. As previously mentioned, this contrasts with the aforementioned code composition works.

With that in mind, we can refer to Figure 3.2 and notice that the first (upper) part of the code composition process is performed individually and independently for each of the tenants' input programs. Those steps prepare the programs to be merged by the code composition module. There is no limitation on the number of programs that can be simultaneously merged.

In a similar way to a compiler, the code composition module must create a higher-level abstraction of the source programs in order to compose them. In fact, the implementation of this module performs many traditional programming language compilation steps: (i) it performs a lexical analysis (i.e., tokenization) of the input programs, (ii) generates an Abstract Syntax Tree (AST) of each program through syntax analysis and (iii) performs a simplified type inferring on expressions through semantic analysis. The first two steps allow the module to work on the AST representation of the programs, which can be later translated back to source code after the merging of multiple ASTs into the final P4 program. The last step, the type inferring, is necessary for the changing of expressions that must occur on specific user-defined types (such as headers or structs). Strictly, the steps performed are:

- **Source file pre-processing:** This is the first step taken for each tenant source file. The P4 language defines some preprocessing directives that are evaluated by compilers before the actual analysis. Those directives are a subset of the ones processed by the C preprocessor (GNU Project, 2023b). Namely, the directives that are processed are **#include**, **#define** and **#if, #elif, #else** directives. Vendor-specific compilers may employ additional directives for their targets. In an implementation of the code composition module, one could employ the C preprocessor in this step. However, the public reference implementation for the architecture of this thesis does not employ it and opts to provide its own P4 preprocessor to reduce coupling with external sources.

- **Source file analysis:** This step takes the preprocessed P4 file and generates an AST representation of it that is going to be used in the next steps. Essentially, this step removes all of the "boilerplate" language elements (commas, parenthesis...) and creates the most concise structure necessary to represent the program. This is also a typical step in the compilation process of many programming languages. The reference implementation of the composition module provided along with this thesis utilizes, in conjunction with tools such as flex (GNU Project, 2023c) and bison (GNU Project, 2023a), the very grammar defined in the P4 specification to perform this step (The P4.org API Working Group, 2017). Figure 3.3 shows an excerpt of a P4 program along with the resulting AST, as an example of the result of this step. The representation used by this thesis emphasizes elements that can be later merged by the composition module, such as the properties of a table declaration, as shown in the figure.

- **AST Simplification:** The resulting AST for each of the programs may contain structures that are interesting for the code composition module to eliminate, as it will lead to a more efficient merging in the next steps. For example, nested parser and control blocks. Those structures are also typically eliminated by real P4 compilers when generating target-specific code. Figure 3.4 shows a typical usage of a nested parser and the resulting code after its elimination in this step. The figure shows the input and results in P4 source code form for clarity, but the actual process occurs entirely in the AST form of the programs.

After the source code analysis is performed for each of the input programs, the resulting ASTs are passed to the part of the Code Composition module that performs the actual program merging (the lower part of Figure 3.2). As depicted in the figure, the merging process also requires information stored in the tenant database, such as the physical port allocation for each program, to perform the next steps.

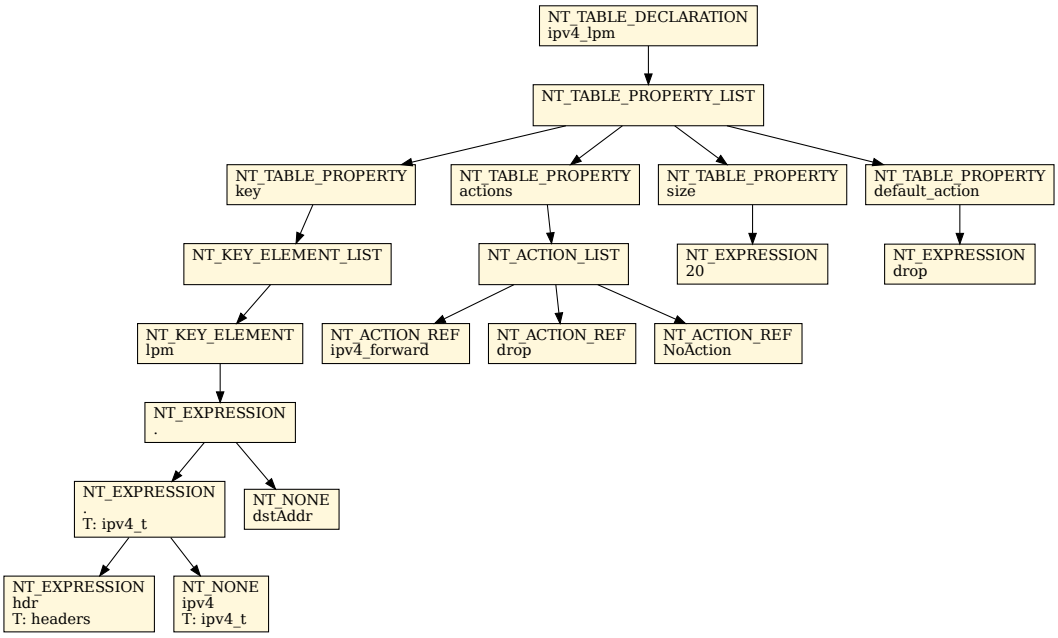
3.2.2 Equivalent Element Unification

The first step in the merging process involves identifying elements that are equivalent across the multiple programs so that they can be unified. This reduces the sizes of the programs, which in turn reduces the hardware resources required to deploy them. Sharing structures is the core of lightweight virtualization strategies and this step illustrates the

Figure 3.3: Example of the AST representation of a P4 code.

```
table ipv4_lpm {
    key = {
       hdr.ipv4.dstAddr: lpm;
    }
    actions = {
       ipv4_forward;
       drop;
       NoAction;
    }
    size = 20;
    default_action = drop();
}
```

(a) P4 Code.



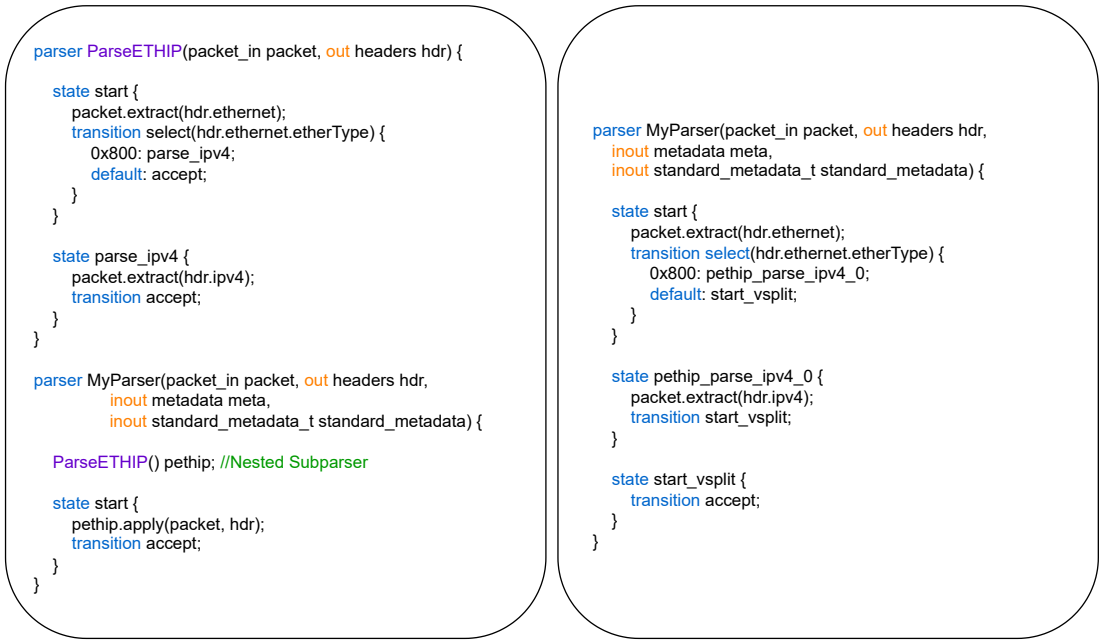
(b) AST.

Source: The Author.

advantages it brings to the table. In summary, this submodule's goal is to accommodate different programming styles in defining structures with the same behavior.

Namely, the elements that the code composition module analyses to try to unify in this step are (i) headers and structs, (ii) parser blocks, and (iii) control blocks. Unifying header definitions is useful to merge elements from different programs that use a member of those headers for some operations, such as tables that match those members, and unifying parser and control blocks straightforwardly saves hardware resources, removing the need for those structures to be duplicated. Of those, the headers are the elements that are

Figure 3.4: Example of the elimination of a nested parser.



```
parser ParseETHIP(packet_in packet, out headers hdr) {

    state start {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            0x800: parse_ipv4;
            default: accept;
        }
    }

    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition accept;
    }
}

parser MyParser(packet_in packet, out headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {

    ParseETHIP() pethip; //Nested Subparser

    state start {
        pethip.apply(packet, hdr);
        transition accept;
    }
}
```

(a) Original Code.

```
parser MyParser(packet_in packet, out headers hdr,
    inout metadata meta,
    inout standard_metadata_t standard_metadata) {

    state start {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            0x800: pethip_parse_ipv4_0;
            default: start_vsplit;
        }
    }

    state pethip_parse_ipv4_0 {
        packet.extract(hdr.ipv4);
        transition start_vsplit;
    }

    state start_vsplit {
        transition accept;
    }
}
```

(b) Modified Code.

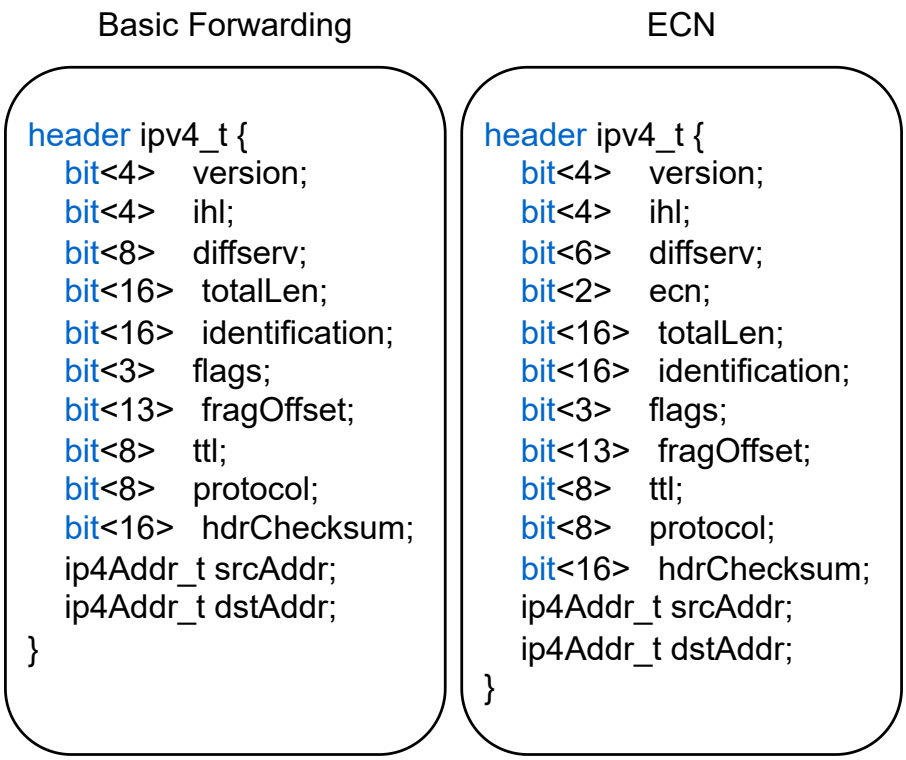
Source: The Author.

likely going to be unified the most, since many standard protocol headers are defined in multiple network scenarios of each tenant (e.g Ethernet and IP headers).

For the headers and structs, the submodule checks the list of declared fields in each of the compared inputs to determine if they are equivalent. The lists are checked in terms of the types of the declared fields, abstracting their names in each program. It is important to note that the fields must be in the exact order in the lists for the headers to be considered equivalent due to the way that the P4 parsers extract data from the incoming packets and place them into the headers.

In order to perform an efficient unification of the declared headers, the submodule may need to "rewrite" some of the user-defined headers so that they can be unified using this notion of equivalent field lists. Consider, for example, two possible definitions of an IPv4 header found in different real examples from the P4 tutorials about basic forwarding and ECN (Explicit Congestion Notification) shown in Figure 3.5. At the time of execution of the submodule, the user-defined types (*ip4Addr_t*) would have already been substituted by their actual base-type definitions (bit<32>), so the submodule would identify that the headers have the same size. Both definitions are similar, except that on the second one (right) the *ecn* field is separated from the *diffserv* in order to be accessed later in the code. Without deeper analysis, those headers would not be eligible for unifications since their

Figure 3.5: Example of headers that can be unified.



Source: The Author

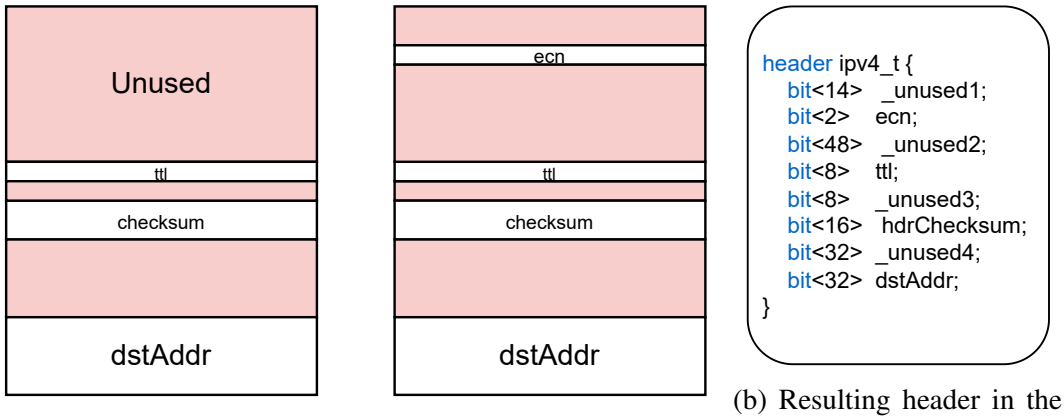
field lists are different. But the submodule is able to identify that the *diffserv* field on the first code is never actually accessed and can then safely divide it in the same way as the second case in order to unify them.

With this process, it is possible to unify headers regardless of their original purpose, provided that (i) they contain the same size and (ii) the **used** fields in each header can be mapped to a bit interval in the other header, which is not split into multiple fields. Figure 3.6 depicts this verification for the multiple versions of the IPv4 header of the previous example. In (a), it shows the used bit intervals of each header and their respective fields. We can verify that both conditions (i) and (ii) are satisfied, therefore the headers can be unified. The resulting header for the merged program is shown in (b)[3].

The process of unification is similar in parser and control blocks. Those blocks essentially can be divided into local declarations and state lists in the case of parser blocks and local declarations and a series of statements in the case of control blocks. The submodule compares the blocks to determine if they are the same piece by piece without considering the naming of the local declarations. For those blocks, one of the advantages

[3]In this Chapter, some examples may contain structures with the same identifiers (e.g "ttl", "checksum") across different programs for ease of reading, but those names are opaque to the virtualization solution, as they are all renamed.

Figure 3.6: Example header unification.



(a) Identification of compatible headers.

(b) Resulting header in the composed program.

Source: The Author.

of the previous step in which the ASTs are simplified is clearly apparent. As previously mentioned, Figure 3.4 depicts the removal of a nested subparser. Note that the result is also a possible programming style that one of the tenants could employ in their programs. Due to the simplification, a parser in the program that uses the nested parser style and one that does not can be unified, as both are going to be in the simplified form when this submodule processes them.

3.2.3 Virtual Headers' Creation

This step creates the new structures that are necessary for the merging. Mainly, it creates virtual headers for the *packet-in* and *packet-out* runtime messages (explained in detail in Section 3.3) and a virtual structure for metadata. The virtual metadata contains information such as which virtual switch the packet belongs.

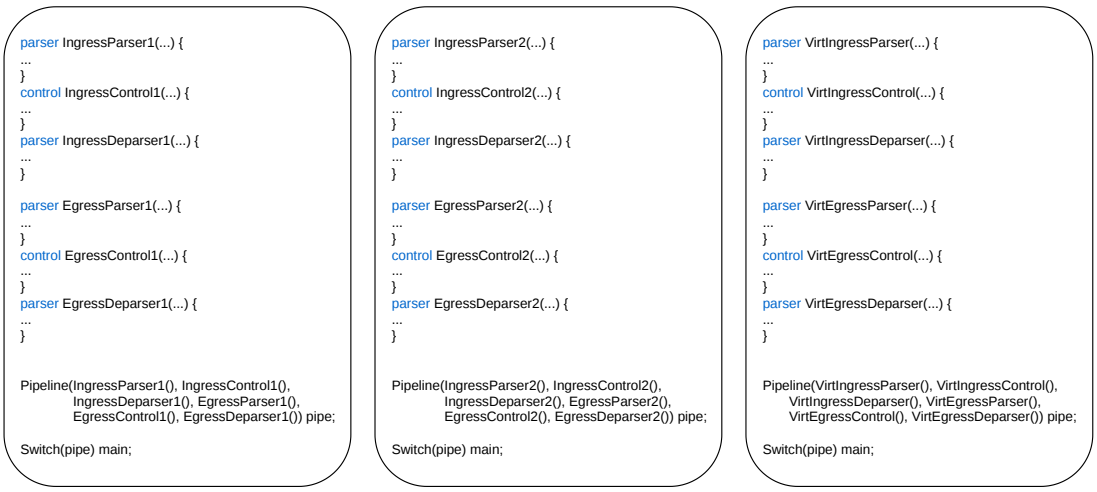
Additionally, this step looks at the **main** program declaration to identify which parser and control blocks will need to be merged. Each architecture defines the specific blocks used to compose this *main* instantiation (recall Section 2.4 and Figure 2.8, which displays a simplified view of the TNA architecture). The submodule then determines which user-defined structures need to be merged before the merging of those blocks by looking at the declaration of the blocks. It then merges those structures by appending their field lists, creating a new structure that will be used in the declaration of the merged blocks that will compose the *main* merged program declaration.

3.2.4 Parser and Control Blocks Merging

When necessary, parsers and control blocks that are different in the merged programs can be unified to implement the behavior of both. This is usually induced by the target-specific architecture and the *main* declaration in the P4 program, which determines how those blocks are chained in the resulting binary configuration program.

Consider the TNA switch architecture just mentioned in the previous Section, depicted earlier in Figure 2.8. Along with some non-P4 programmable components, it defines six P4 blocks that compose the main program declaration: (i) Ingress Parser, (ii) Ingress Control, (iii) Ingress Deparser, (iv) Egress Parser, (v) Egress Control and (vi) Egress Deparser. Those elements are going to be present in every program for this architecture, and their behaviors can vary. It is the goal of this submodule to create virtual blocks that unify each of the corresponding blocks of the merged programs.

Figure 3.7: Example of the program-wide merging process.



(a) Program A. (b) Program B. (c) Merged Program.

Source: The Author.

Figure 3.7 shows a simplified example of this unification process. Two P4 programs, “A” and “B”, designed to the TNA switch architecture, are composed in the merged program “C”. The code composition module analyses the *main* switch declaration of each program to identify which blocks need to be composed, regardless of their declaration order in the multiple programs.

The parameters of those block declarations may contain variables of (i) P4 base types, (ii) architecture-specific types, or (iii) user-defined structure types. Of those, (i)

and (ii) do not require special treatment, as they will be the same in all merged programs. It is for (iii) that the importance of the previous step, which creates virtual structures containing the fields of the multiple user-defined types, becomes apparent.

Typically, an architecture will define a parser as the first programmable block that composes the *main* switch declaration, as the headers need to be extracted to be operated upon. Parsers in the P4 language contain a list of local declarations (usually variables) and a list of parsing states. The local declaration lists are merged by simply appending them to the merged program. The merging of the states, which is where the actual parser logic is implemented, is more involved. As described in Section 2.4.2, the parser logic can be seen as an FSM, with each state of the parser associated with a state in the FSM.

The merging process uses the *start* state of this first parser to populate a field in the virtual metadata, which identifies the virtual switch that will process the packet. This is done by verifying the ingress port of the packet, which determines the virtual switch that it belongs to, defined in the tenant database. It also creates new starting states for each virtual switch, which the packets are then forwarded to.

By utilizing the input port as a mechanism to identify which virtual switch to process a packet by the input port, we eliminate the need for an external mechanism to perform this function. Some earlier works, for instance, utilize tags appended to the packets through an external server or network middlebox (ZHENG; BENSON; HU, 2018). With the scheme proposed in this Thesis, there is no need for those external systems and, therefore, the deployment of the virtualization solutions does not incur major changes to the network infrastructure. This approach, however, may limit the maximum number of tenants for a given host with X ports to X. Therefore, the architecture also supports for virtual switch identification by external mechanisms which appends the program ID to each incoming packet. This way, the system administrator may employ other slicing strategies, such as bandwidth limitation.

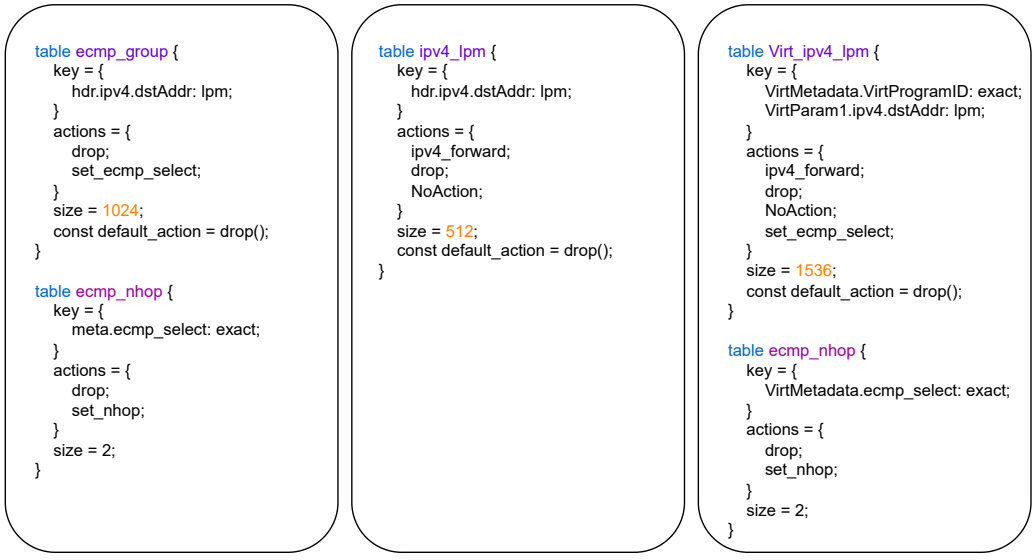
A naive strategy to be employed would be to emplace each state of the original programs into the merged program, which would result in a parser with a number of states equal to the sum of the states in each input program. With lightweight composition virtualization, however, it is possible to greatly reduce this number. If some states are equal across the merged programs, they can be unified in the merged program. This was mentioned in Chapter 2, when discussing composition-based virtualization (Section 2.8.3). States in the P4 language are syntactically divided into a list of *parser statements* followed by a *transition* statement. The architecture presented in this thesis only requires

the *parser statements* to be equal, removing the need for temporary states, such as the ones employed by P4Visor (ZHENG; BENSON; HU, 2018). A complete example of this composition strategy is shown in Appendix C, with the merged ingress parser of three distinct programs (*basic forwarding*, *calculator*, and *load balancer*).

For the control blocks, there is also a syntactical division into *local declarations* (actions, tables, and variables) and the *apply* block. The merging process works on them separately. First, local elements are merged, which includes (i) copying all the variable declarations from the input programs to the resulting program (including *externs* such as registers), (ii) the same with the actions, but unifying them when they are equal, and (iii) merging tables whenever possible.

The most important part of this step (and arguably of the whole merging process) is the merging of the declared tables. Tables are compatible to be merged if their match key is the same and if their default actions (taken in cases of table misses) are equal and constant. The remaining elements can be unified in different ways, such as summing their sizes (maximum number of entries) and using the union of the sets of actions.

Figure 3.8: Example of the merging of P4 tables.



(a) Load Balancer. (b) L3 Forwarding. (c) Merged Elements.

Source: The Author

Take, for instance, the example in Figure 3.8, which shows the resulting tables of the merging of control blocks from a load balancer and an L3 forwarding device, adapted from the P4 tutorials. In the example, tables *ecmp_group* (a) and *ipv4_lpm* (b), are compatible. The purposes of the tables on the input programs are completely different, but that is irrelevant to the merging process, as it only verifies the compatibility criteria. The

resulting table *Virt_ipv4_lpm* (c) contains all the actions necessary, its size is equal to the sum of the original tables' sizes, and, most importantly, a new field in the *key* property is added. This new field is used to determine to which program each entry belongs. Assume, for instance, that both tenants need to add an entry for $10.0.1.0/24$ (likely with different actions). The actual rules that would be inserted in the merged program must contain their program IDs to differentiate them. Table *ecmp_nhop* (a) is not compatible with any other table, so it is simply copied to the resulting program.

It is worth reinforcing that the tenants **do not need, at any point, to acknowledge any aspect of the merged program**. From their point of view, the only program that is executing is their own. As such, rules originating from the tenant of the program (a) to table *ecmp_group* must actually be changed to rules to table *Virt_ipv4_lpm*. This process is detailed in the next session, which discusses runtime management (3.3).

Not all tables that are compatible can be merged in the resulting program. That is because dependencies (the order in which the tables are applied) must be taken into account. P4 programs do not allow for tables to be applied twice (if one wishes to do that, it requires duplicating the table). Suppose, for instance, that control block A containing tables A_1 and A_2 (applied in that order) must be merged with control block B with tables B_1 and B_2 (also applied in that order) and A_1 is compatible with B_2 and A_2 is compatible with B_1. In this scenario, we cannot merge both pairs of compatible tables, as the order of the apply statements would be different in the resulting program (even though control flow would never allow for more than two tables to be applied for a single packet).

Note that in the example, there are two possible choices of merges to be made, but depending on the number of tables and their compatibilities, there may be many more. This problem was described in the work of P4Visor (ZHENG; BENSON; HU, 2018), but considering the merging of two programs, which is their focus. Here is presented a different formulation of the problem that generalizes to an arbitrary number of programs.

Let T be the set of all tables from the control blocks that are being merged. Let the boolean function $D(t_1, t_2)$ denote dependencies between the tables:

$$D(t_1, t_2) = \begin{cases} 1, & \text{if } t_1 \text{ must be applied before } t_2 \\ 0, & \text{otherwise.} \end{cases}$$

We can construct a **directed** graph $G = (T, E)$ where the set of nodes is T and the set of edges is defined by the function D:

$$\forall t_1, t_2 \in T, (t_1, t_2) \in E \leftrightarrow D(t_1, t_2) = 1$$

By the aforementioned compatibility criteria, we can define a relation R in T where tables are related if they are compatible. It is clear that this relation is symmetric and transitive, and we can consider that tables are compatible with themselves so that R is an equivalence relation.

We can then define a new **undirected** graph $G' = (T', E')$ where T' is the set of equivalence classes in R (that is, the set of sets of compatible tables). Let $[t]$ be the equivalence class of $t \in T$ in R, we can define E' as:

$$\forall t_1, t_2 \in T, ((t_1, t_2) \in E \wedge (t_2, t_1) \in E) \leftrightarrow ([t_1], [t_2]) \in E'$$

With this construction, choosing a set of tables to be merged is equivalent to choosing an independent set of nodes in G'. Choosing the set of merges that leads to the maximum amount of table merges in the resulting program is equivalent to solving the Maximum Independent Set (MIS) problem. It is shown in the literature that MIS $\in$ NP-complete (KARP, 1972).

Still, simply taking the set with the most number of merges is not always desirable, as those sets only represent the equivalence classes (i.e. the resulting merged tables) without taking into account how important they are. It is possible to define a value for each vertex $t \in T'$ to represent its importance in terms of hardware usage. The authors of P4Visor (ZHENG; BENSON; HU, 2018) performed a brief evaluation that shows that hardware usage scales in a sub-linear fashion on their test platform when the complexity of the tables increases, but it is unclear if it is a target-specific optimization. For instance, merged tables that are comprised of more tables of T may receive higher values. The sizes of those tables may also affect this evaluation. The complexity of the actions of those tables (which must be duplicated if they are referenced in more than one table) is another such factor. A deeper analysis of those factors is left for future work.

Selecting the independent set of G' that maximizes the values of the chosen merged tables is equivalent to solving the Maximum Weight Independent Set (MWIS) problem. This problem is also known to be NP-complete (BOMZE et al., 1999).

Currently, the reference implementation of the architecture selects a sub-optimal

set using a simple algorithm that selects vertices while no dependency is created. A possible future work related to this thesis may involve testing different heuristics for selecting an independent set. Also, a more rigorous verification of the necessary complexity of the merged programs (in terms of the number of compatible tables in T) for an exact solution of the MWIS problem to be impracticable must be conducted.

Once a set of tables to be merged is selected, the merging of the *local declarations* portion of the control block is finished. The merging of the remaining portion, the *apply* block, is theoretically simple to define with *switch* statements. We would verify the program ID metadata and apply the appropriate command block. However, merged tables complicate this logic, as they can only ever be applied once in the whole block of the merged program. Additional control logic is required to achieve the desired behavior.

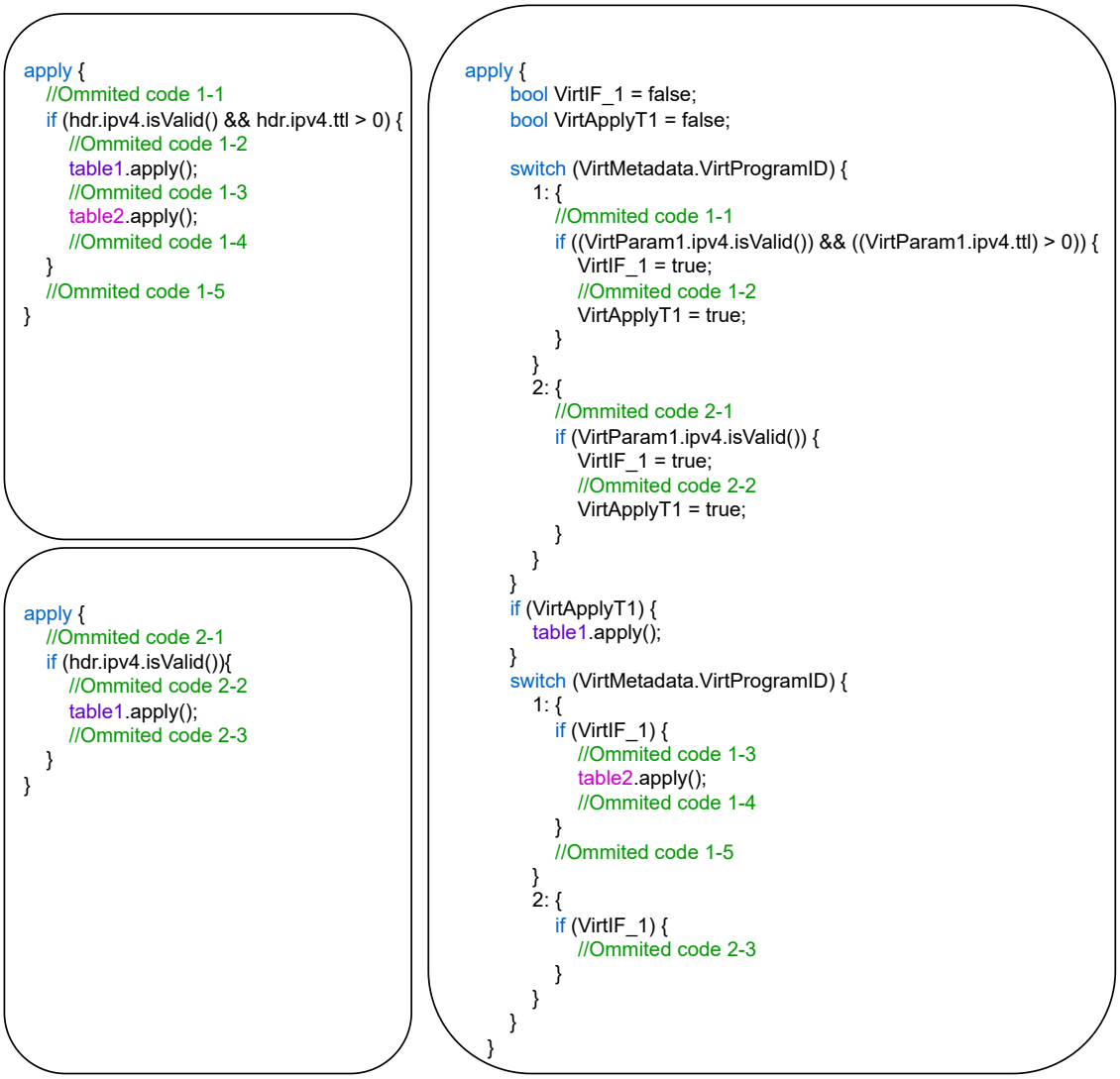
Figure 3.9 depicts a simple scenario as an example of such a case. In the merged program, *table1* is merged from the inputs, therefore it must be accessed by both. However, it can only be applied once in the block. Therefore, the *apply* blocks of the inputs must be split. Additional control variables are used to return to the conditional context of the programs before the table *apply* statements in each of them. This is not necessary for non-merged tables, such as *table2* from the top program.

After the *apply* blocks have been adjusted, the merging of the control blocks is complete. Additionally, for the last control block of the ingress/egress switch pipelines, a new table is defined and applied at the end of the block to verify the output port of the packet. This ensures that a tenant will not be able to send packets to a port belonging to a different tenant. The control block in which this table is placed is target-dependent, as the structure of the pipelines varies.

3.2.5 Target-Specific Adjustments

This submodule performs the target-specific portion of the merging process. The process cannot be fully target-independent since different architectures may have restrictions on the implemented subset of the P4 language. For example, as discussed in Section 2.4.1, different targets may employ incompatible efficient ways to circumvent hardware restrictions. Another example is in which control block the verification tables (mentioned at the end of the last section) must be placed.

Figure 3.9: Example of the merging of control *apply* blocks.



(a) Input Programs. (b) Merged Program.

Source: The Author

3.2.6 Runtime Mapping and Merged Program Output

Those final submodules generate the resulting outputs that will be used by other modules. Namely the runtime mapping (used by the runtime module) and the merged program (used by the data plane update module).

The runtime mapping includes information about how the structures of the input programs relate to the ones in the merged program. For instance, what table in the merged program represents each table in the input programs, what tables are merged, what actions are merged, etc. The output P4 program contains the merged structures of the previous steps. The submodule performs the inverse process of the P4 source code analysis step to

generate a P4 file based on the resulting AST.

3.3 Runtime Virtualization Management

The resulting merged program produced by the code composition module and deployed by the data plane update module on the host device simultaneously implements the behavior of all the tenant's programs. As such, an abstraction layer is needed for the tenants to manage their structures simultaneously.

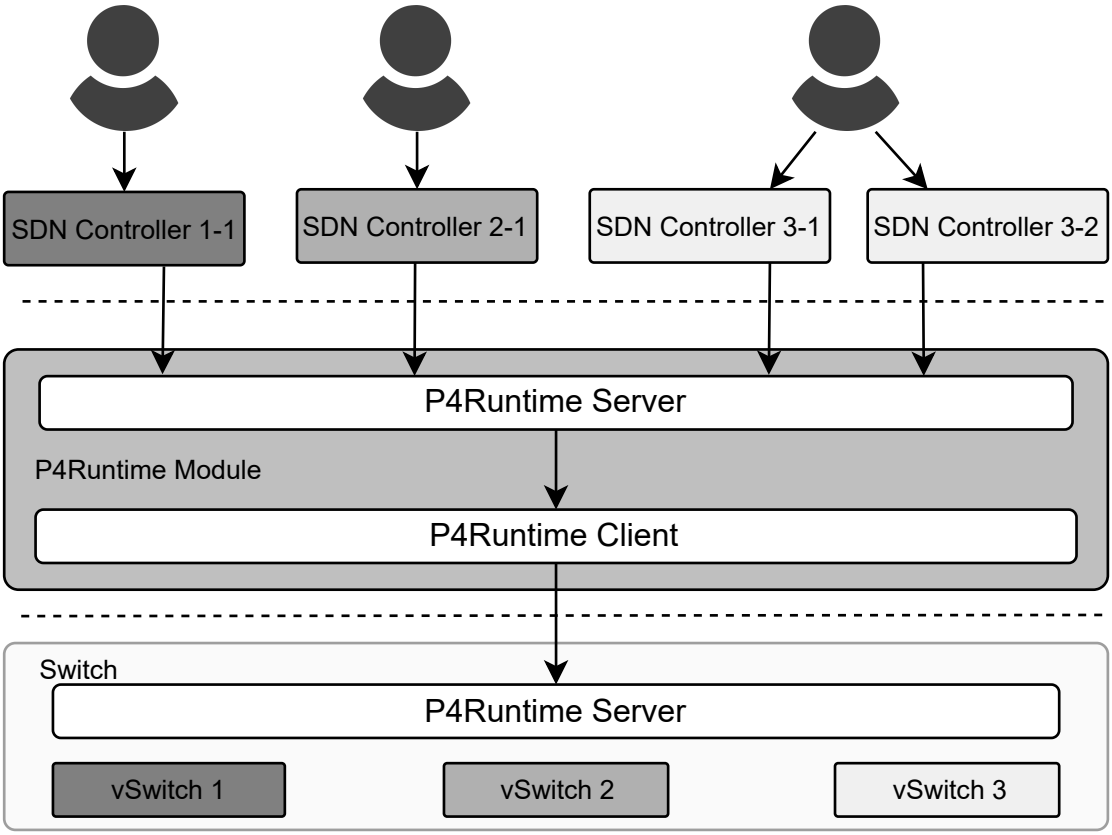
This abstraction is provided by the Runtime module (Fig. 3.1). It offers a logical view for each tenant that their virtual device is deployed on the host device just as it would if no virtualization solution were being used.

By providing this logical view, the controller that manages virtual devices can be identical to that used to manage "normal devices". This is critical to enable tenants to use standard control-plane software such as ONOS (ONOS, 2021) to manage their virtual switches. It is unreasonable for the tenants to rewrite control plane software to comply with the virtualization architecture requirements.

One of the design principles of the proposed architecture is *target-independency*. As such, this module is not restricted to a specific CDPI implementation, such as P4Runtime. Still, as mentioned in Section 2.6, P4Runtime is an open interface implemented by most commercial P4 hardware targets. Even the ones that provide target-specific interfaces usually provide P4Runtime alongside it as an alternative. Therefore, to achieve a broad range of targets, the reference implementation of the architecture implements this module focused on P4Runtime controllers. Nevertheless, the ideas presented here can be utilized to implement the module with another CDPI protocol.

Figure 3.10 shows the operation of the runtime module. In a traditional P4Runtime operation scenario, the P4Runtime clients (SDN controllers) connect to the P4Runtime server implemented in the target switch. The runtime module implements this interface, instantiating its own P4Runtime server, which is the one the controllers will connect to. Essentially, the module acts as a proxy between the controllers and the switch server. The internal logic of the module will modify the request from the controllers, made by assuming their virtual device is deployed on the target, to requests to the merged program, which is the one that is actually deployed. Those modified requests are then carried out through a P4Runtime client, also instantiated in the module, which is the only client that connects to the host target's server.

Figure 3.10: Runtime module operation scenario.



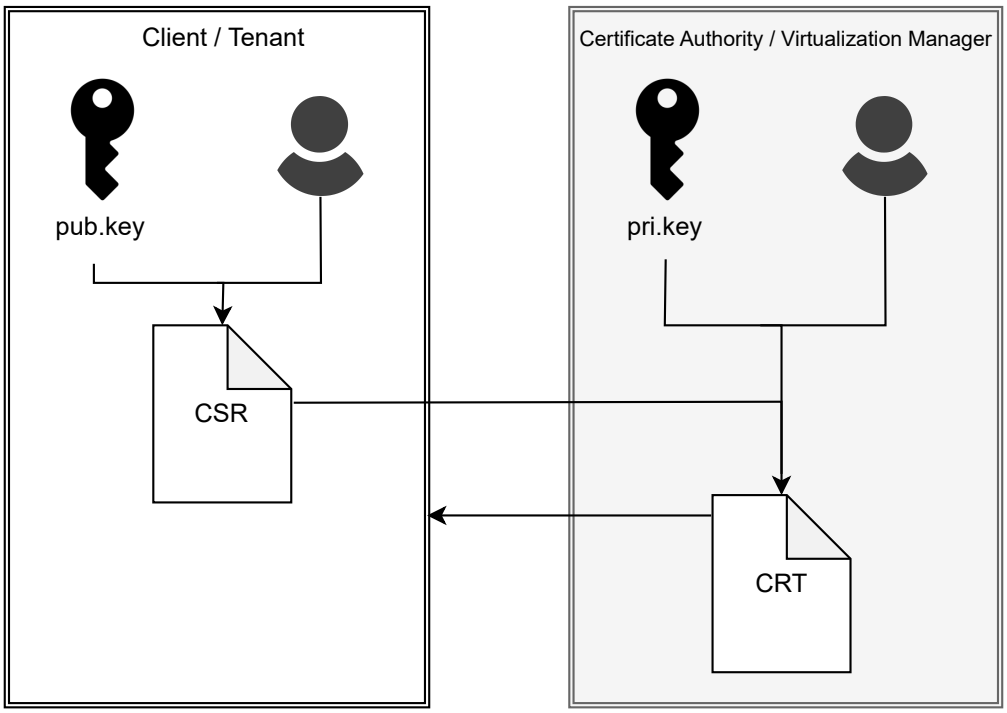
Source: The Author

The figure also depicts that multiple controllers can be responsible for managing a single device, as is the case of vSwitch 3's tenant, which instantiates two controllers. This is supported by the P4Runtime standard as a way to implement backup controllers in case of primary failures. The runtime module must also provide support for this functionality since SDN controllers are expected to behave according to this logic.

3.3.1 Controller Authentication

One of the most basic functions the runtime module must perform is to identify which tenant a connecting controller belongs to, so it can appropriately modify requests to virtual switches and verify permissions. This authentication is provided by the security module and the tenant database. As previously mentioned, P4Runtime utilizes gRPC as its underlying client/server model implementation, an open RPC framework. This framework enables secure communication via Transport Layer Security (TLS). Therefore, it is possible to utilize a TLS certificate scheme to identify the tenants of each controller.

Figure 3.11: Brief overview of the TLS certificate signing process.



Source: The Author

A brief overview of the TLS certificate signing process is depicted in Figure 3.11. A certificate authority (controlled by the virtualization solution administrator) is responsible for signing certificates for each client (tenant). The tenants create a Certificate Signing Request (CSR), which contains their public key and other information that identifies them, and sign it with their private key. Those keys are generated by the tenants in advance. This CSR is sent to the virtualization manager so that it can sign it with its private key, creating the tenant's certificate (CRT, in the figure). The controllers always provide the certificates to the server running in the P4Runtime Module to be authenticated, allowing the identification of the clients and the security of the communication channel. The P4Runtime server of the runtime module also has a signed certificate of its own, which the controllers use to verify its authenticity.

An important observation regarding the security of communications is the encryption of the communication channels. The described TLS scheme provides a secure channel between the multiple SDN controllers and the P4Runtime server of the runtime module (upper part of Figure 3.10). The channel between the P4Runtime client of the runtime module and the target's server is not protected by this scheme (lower part of Figure 3.10), as it would depend on the target vendors to implement a secure server. The system administrator is responsible for implementing external security mechanisms in those cases

in which TLS cannot be employed. As an example, it would suffice to run the P4Runtime module entirely inside the target switch, which some devices allow for, so that the channel would connect processes in the same host.

After authentication is established, the internal data structures of the runtime module provide the other authentication functionalities of the P4Runtime specification (The P4.org API Working Group, 2019). This includes waiting for *MasterArbitrationUpdate* messages informing the *electionIDs* of the controllers to establish primary/backup roles, creating the *StreamChannel* structures that will be used for packet I/O and timeout notifications, and preparing for Write/Read requests.

3.3.2 Runtime RPC modifications

To put it briefly, the main goal of the runtime module is to modify runtime requests between the tenant's controllers and the host target. Recall that the code composition module modifies the structures of the input programs (such as tables) on the merged program (Section 3.2.4). Requests for the original structures will naturally differ from those for the modified structures.

P4Runtime utilizes a special file, called *p4info*, that describes all the program structures that can be accessed through the CDPI, including the IDs and names of those structures. Each SDN controller will work with their own *p4info* files (describing structures of their virtual devices) when accessing the P4Runtime server. The P4Runtime module also has the *p4info* file for the merged program. One of the artifacts produced by the code composition module is a runtime mapping of the elements of the *p4info* files of the virtual devices to the *p4info* file of the merged program (Section 3.2.6).

With that information, it is possible to rewrite the *WriteRequest* and *ReadRequest* messages from the **Write** and **Read** RPCs, as long as the entities of the requests are registers, counters or non-merged tables. However, the merged structures that benefit from lightweight virtualization need a special adjustment to identify the programs. The code composition adds fields to identify the program to which the entries belong. At runtime, those fields are added to the requests based on the tenant of the SDN controller (and removed from the responses from the target's server).

Other adjustments not discussed in extensive detail are also made to accommodate other features. If a controller tries to read all the entries from a table (using the wildcard reads available with P4Runtime), only the entries belonging to that controller are returned;

if a controller tries to insert more entries in a table than their tenant declared as the table's size in the source program, it shall fail (even though merged table may have free entries not being used by the other tenants); etc.

The **GetForwardingPipelineConfig** RPC is implemented by returning the *p4info* files of the virtual switch associated with the controller's tenant along with its binary configuration files if needed. Typically, the controllers require only the *p4info* files. Still, for the sake of completion, it is possible to compile the individual programs to assemble the response for this RPC in full.

The **SetForwardingPipelineConfig** RPC is currently disabled for use by the tenants. As previously discussed, this RPC allows to change the whole forwarding pipeline of the target device. In other words, it enables the controller to change the program deployed on the device. The virtualization solution does not allow a single program to be changed like this. This derives from the main downside of the composition-based virtualization methods (Section 2.8.3). The virtualization architecture provides a virtual switch runtime deployment strategy that can be employed when hardware resources are available (discussed in Section 3.4), in which case the RPC is enabled.

Lastly, the **StreamChannel** RPC requires more involved adjustments, as it is a different RPC from the others. It opens a bidirectional stream that is kept open until the controller disconnects. All controllers will have a single stream of this kind open during their live time. The P4Runtime client implemented by the runtime module, which connects to the host target's server, also opens a single stream channel for its connection. The necessary messages from all controllers use this single channel in a controlled manner. A variety of different messages can be exchanged through the stream channel. The runtime module performs the following adjustments:

- **MasterArbitrationUpdate:** This message is not altered, as it does not need to be forwarded to the host's server. The runtime module's server keeps track of the electionIDs to determine the roles of each controller. This logic is isolated for all of the tenants (each will have a list of controllers with their electionIDs). When the module starts, a message of this kind is sent to the host's server by the P4Runtime client of the runtime module. This client is the only controller from the point of view of the host's server.

- **Packet I/O:** The PacketIn (packets forwarded from the programmable device to the controller) and PacketOut (the other direction, from the controller to the device) messages require additional information provided by the code composition to work

correctly. The P4 language allows the programmers to define metadata that will be sent along the packets between the device and the controllers. Those metadata are implemented as headers and emitted by the programs just like regular headers. However, the P4Runtime protocol separates them from the rest of the packet. As such, the runtime module must identify to which program the packets belong and modify the metadata appropriately. Figure 3.12 depicts how this is achieved alongside an example with different PacketIn headers. The controller headers on the input programs become regular headers in the merged program. The merged program has a controller header with metadata to identify the virtual program. The appropriate metadata is extracted (PacketIn) or added (PacketOut) to the packets in the runtime module.

- **IdleTimeoutNotification:** This message is sent by the server to warn about table entries that did not match in a period defined by the controller when they are inserted. This is mainly used to remove unused entries from tables, allowing for smaller tables, which saves up resources. This message can simply be forwarded to the table's program's controller for non-merged tables. For the merged tables, the module checks the program that the entry belongs to and forwards the message.

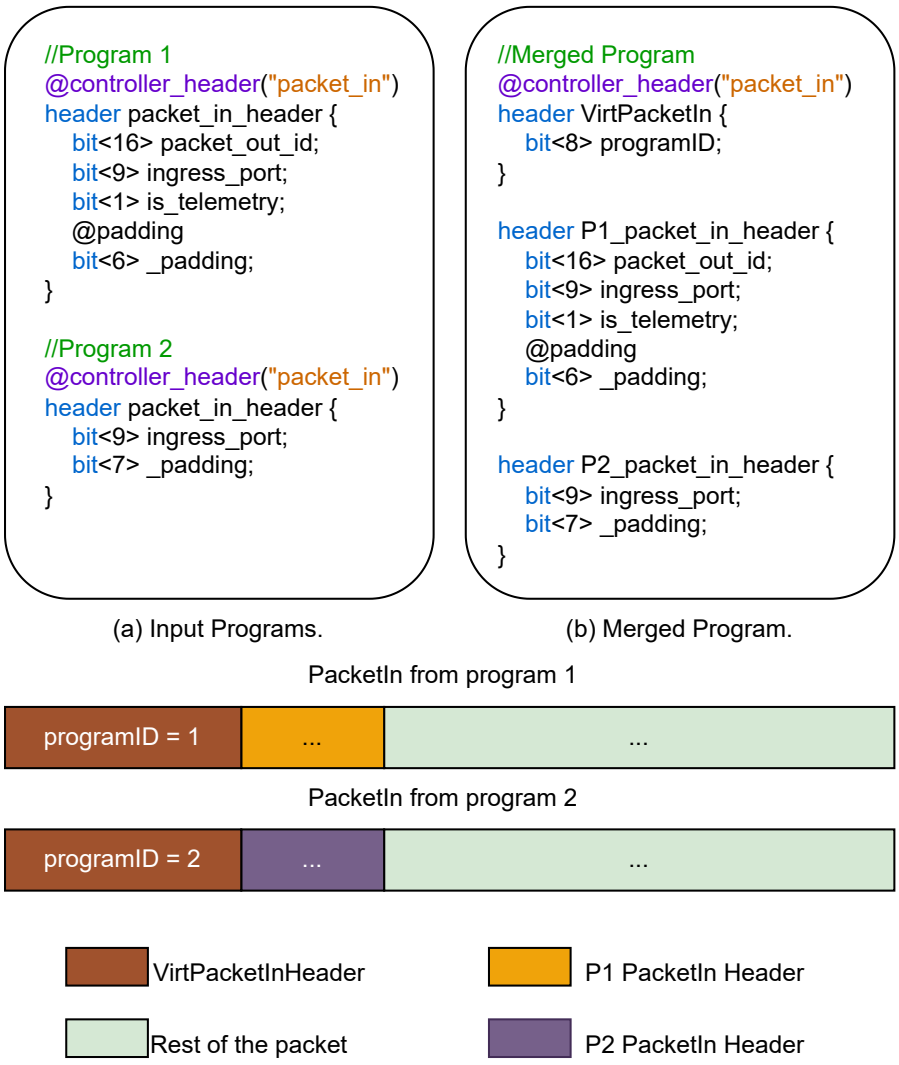
3.4 Runtime Deployment of Virtual Switches

As mentioned in Section 2.8, one of the main open challenges in composition-based programmable switch virtualization is the possibility to remove existing virtual switches or to deploy new virtual switches in runtime. This feature can also be called virtual switch "hot-swap".

To recall, this limitation is intrinsic to the code composition and deployment strategy, as those updates involve generating a new composed program based on the updated virtual switches and deploying this new program in the target. This process incurs a configuration downtime in the target device during which it is unable to process incoming traffic. Additionally, all of the internal state of the device (entries in tables and values in register memory) is lost on such updates.

Since this affects all of the virtual switches deployed in the target, it will affect all of the tenants in the system, regardless if the virtual switch being removed / deployed belongs to them, which is undesirable depending on the system's context. Ideally, the

Figure 3.12: Example of modifications on the Packet I/O headers for runtime control.



(a) Input Programs. (b) Merged Program.

(c) PacketIn(s) from the host device.

Source: The Author

other tenants in the system should not be affected by such updates or even need to be aware that those updates are occurring. To achieve this, updates should not disrupt the other tenant's traffic (or at least not as much to be perceivable by their hosts and applications).

In order to enable virtual switch hot-swapping in some deployment scenarios (when the system administrator has spare hardware resources), a strategy to update the data plane is described in this section. The goal is to update the data plane from the point of view of the requesting tenant (the tenant that owns the switch being added / removed) without disrupting or causing perceptible changes from the point of view of the other tenants in the system.

The switch in which the composed code is originally deployed will be called S_1 in

this section. The proposed strategy requires an additional programmable switch to which the traffic will be steered after the process is completed, which will be called S_2 in this section. In short terms, the main idea of the proposed hot-swap solution is to deploy the newly composed program (with the updated virtual switches) in S_2 and to steer new traffic to it. The costs of this additional switch can be amortized by the number of other operational switches in the system, as it can be shared among them. An administrator that provides virtual switches as a service, for instance, can have a large number of hardware switches available to amortize this cost. Additionally, another forwarding device that is not necessarily programmable (as it does not need to be reconfigured) is required to steer the traffic. This can be seen as a top layer switch in a tree topology. This switch will be called S_s in this section.

The proposed strategy involves the migration of part of the data plane state from S_1 to S_2. As previously described, each data plane state element (tables, registers...) of the virtual switches have an equivalent element in the composed code. Different elements of virtual switches may me mapped to the same element in the composed code. Those elements can be divided in two groups: (i) Traffic-Modifiable State elements (TMS), such as registers and (ii) Non-Traffic-Modifiable State (NTMS), such as table rules. NTMS elements may be migrated with the help of the control-plane modules (Data Plane Update Module and Runtime Module), since they can block the write access to those elements momentarily while the state is being copied. The TMS elements are trickier since the traffic itself can modify the values, and at higher speeds than the control plane modules. In the proposed strategy, this migration is achieved by utilizing the traffic itself to carry data from S_1 to S_2 with the help of S_s. In summary, with the help of S_s, the main idea to perform the swap operation is to (i) deploy the new program in S_2, (ii) copy the NTMS and TMS elements of the not-affected virtual switches (with the control plane or with data plane traffic) and (iii) redirect the flow to S_2.

3.4.1 Migration of TMS Elements

In order to migrate the TMS elements from S_1 to S_2, the composed and deployed program is augmented during the code composition stage to be able to perform specific operations during the migration process. Figure 3.13 depicts the packet flow during migration when those operations occur. S_1 is able to add a *Migration Header* to outgoing packets containing information about the state modified by that packet and its new values.

A clone of those packets is sent to S_2 when arriving in S_s, and the header is removed from the original packet before forwarding it to the next hop. S_2 is able to identify and extract the *Migration Headers* and update its internal state accordingly. This way, the information about each packet that modifies TMS elements in S_1 will be sent to S_2.

When the system administrator enables hot-swap by employing this strategy, S_1's processing relating to dropping and forwarding packets is slightly modified. Every packet arriving in a given input port P in S_1 will be sent to the same port P as output, regardless of the forwarding decision of the virtual switch that processed that packet. The *Virtualization Header*, also depicted in Figure 3.13, will contain the original decision (forwarding port, drop, multicasting...) that the virtual switch made during packet processing. S_s will be responsible to applying that decision, forwarding the packet accordingly to its equivalent port. This way, packets that would be dropped by S_1 after modifying TMS elements during this period are forwarded to S_s and dropped there instead, so that the clones can still be made to inform S_2 of the changes.

The hot-swap process must ensure that each TMS element in S_1 eventually synchronizes with its equivalent in S_2. If an element is copied at least once from S_1 to S_2, it is considered synchronized, since if the migration were to be stopped, the elements would have the same value[4]. Therefore, the Data Plane Update Module employs an algorithm (Algorithm 1) to verify whether all TMS elements of S_1 were modified at least once since the start of the process, which leads to them being copied to S_2.

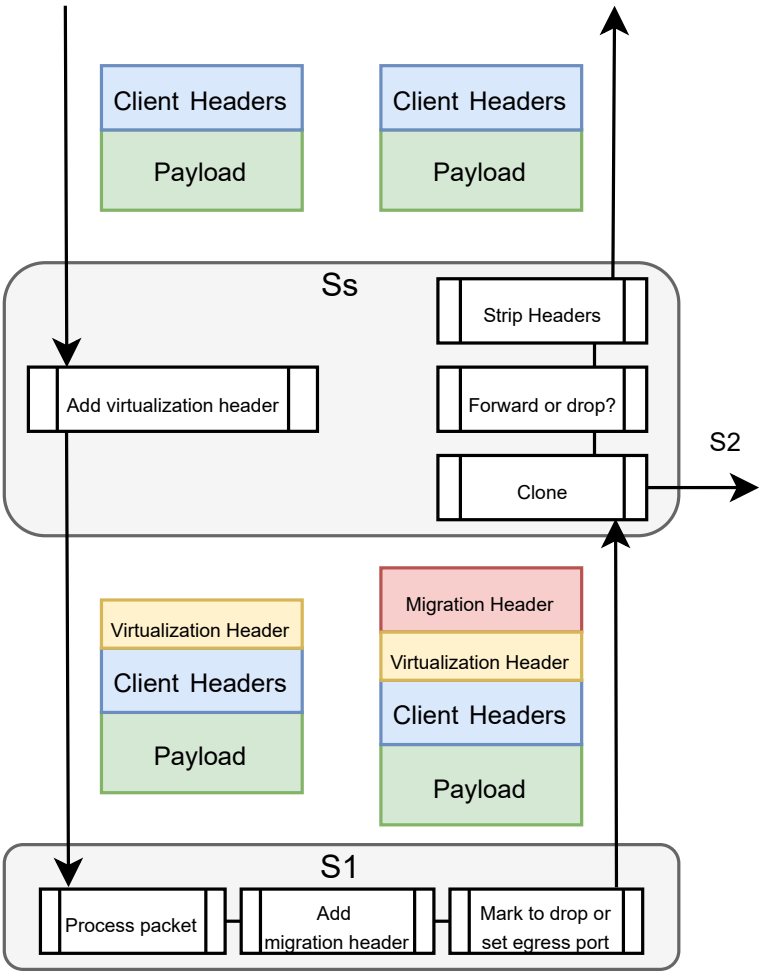
The algorithm uses two boolean arrays, **M[N]** and **S[N]**, where **N** is the maximum number of elements that may need to be synchronized. Array **M** is implemented in S_s' hardware. Array **S** is implemented in the Data Plane Update Module. Since it is expected that S_s is non-programmable, **N** must be at least as large as the maximum number of TMS elements in S_1, which varies by target device. The Data Plane Update Module maintains a one-to-one mapping of each TMS element in S_1 to an index of **M** and **S**.

Array **M** is used to track which registers were copied through data plane traffic. When S_s forwards a packet with the *Migration Header* to S_2, for each register information contained in the header and their mapped index i, it sets **M[i] = 1**. Array **S** is used to track which registers the Data Plane Update Module is sure are synchronized.

Assuming, without loss of generality, that for a given migration process, the elements to be migrated are mapped to the first **n** indexes of the arrays, the first step of the algorithm is to set every value up to **n** of the control arrays to 0 and the rest to 1 (Algo-

[4]With the exception of packets in S_1's pipeline. See Section 3.4.2

Figure 3.13: Migration of TMS Elements.



Source: The Author.

rithm 1, lines 1 - 8). At this point, the Data Plane Update Module configures S_1 so that it appends the *Migration Headers* to packets when necessary, as shown in Figure 3.13.

The algorithm then repeats a loop until every element has been synchronized (i.e., all positions of **S** are 1). It waits for a specific amount of time (t_1) to give traffic a chance to copy the remaining registers (line 10). It then checks, for every index i that has not been synchronized yet ($S[i] == 0$, line 12), if it has not been copied through the data plane ($M[i] == 0$, line 13). If it has been copied, it simply updates $S[i] = 1$ (line 21). Otherwise, the value is copied through the control plane by (i) reading the value from S_1 and (ii) writing the value to S_2 (line 14). Then, $M[i]$ is checked again to see if no packets containing information about a data plane update of that register were sent to S_2. If $M[i] == 0$, then it can be sure that the copy was performed correctly and the element was synchronized, so it sets $S[i] = 1$ (line 16). Otherwise, if $M[i] == 1$, a race condition

occurred, since during the control-plane-based copy, the data plane traffic also copied that element. Since between the control plane (i) reading the value from S_1 and (ii) writing the value to S_2, the value could have been changed in S_1, gone through S_s, and written to S_2, it is possible that the control plane wrote an invalid value to S_2. In that case, the control plane resets $M[i] = 0$ and tries to synchronize it again at the next iteration.

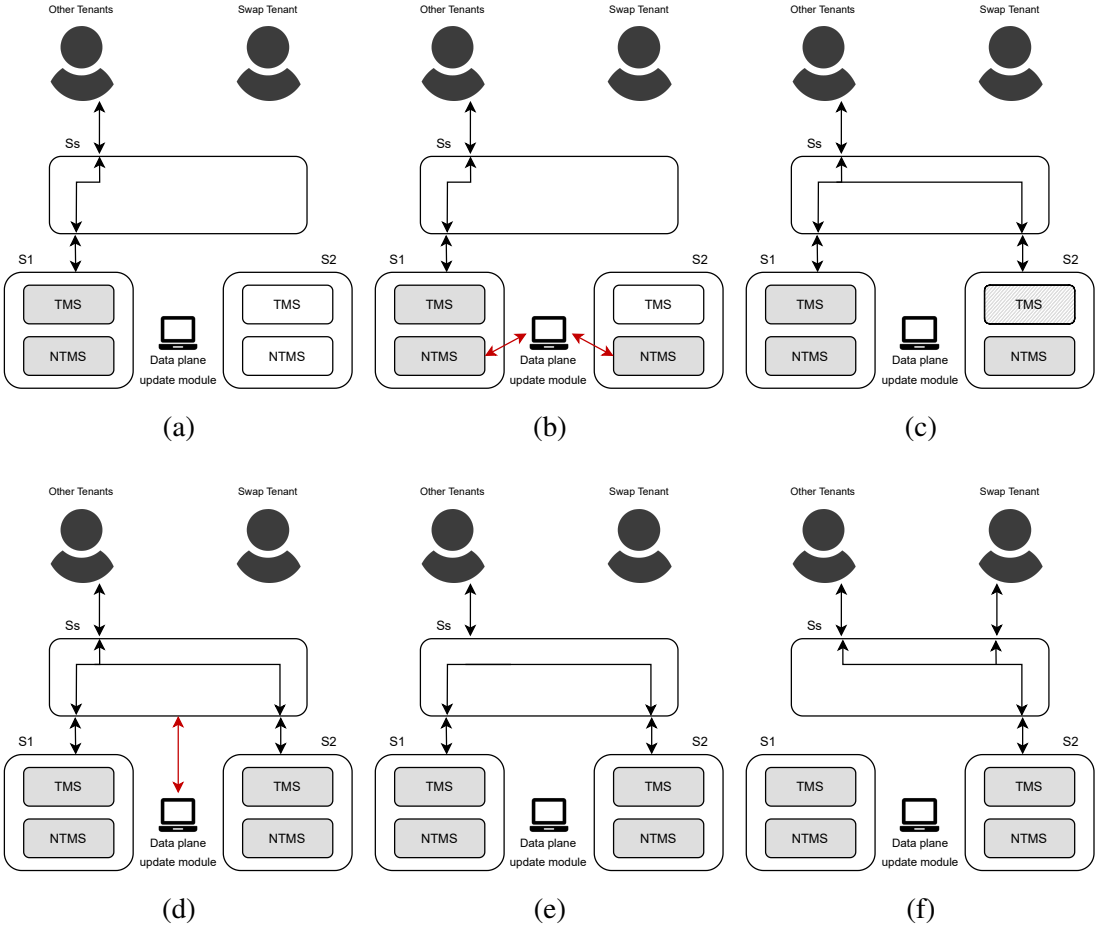
Algorithm 1 Control-plane assisted migration of TMS elements

```
1:  for i ← 1 to n do
2:      M[i] ← 0
3:      S[i] ← 0
4:  end for
5:  for i ← n + 1 to N do
6:      M[i] ← 1
7:      S[i] ← 1
8:  end for
9:  while ∃i such that S[i] == 0 do
10:     Wait for sync...
11:     for i ← 1 to n do
12:         if S[i] == 0 then
13:             if M[i] == 0 then
14:                 Copy register i from S1 to S2
15:                 if M[i] == 0 then
16:                     S[i] = 1
17:                 else
18:                     M[i] = 0
19:                 end if
20:             else
21:                 S[i] = 1
22:             end if
23:         end if
24:     end for
25: end while
```

It is possible that the check for $M[i] == 0$ at line 15 fails an indefinite number of times, but each failure is extremely unlikely. For it to fail, a packet must have modified the specific element during the operation to copy that element (line 14). This operation takes time t_2 in the order of a couple of milliseconds. As mentioned, between each attempt, the algorithm waits for an amount of time t_1 (line 10), which can be in the order of a couple of seconds. During t_1, no changes were made to that specific element. Normally, those elements are read-heavy data plane components, such as bloom filter registers. Therefore, t_1 and t_2 can be defined such that $t_1 \gg t_2$, and the probability of a race condition can be arbitrarily small.

3.4.2 Step-by-step Hot-swap Process

Figure 3.14: Step-by-step hot-swap process.



A step-by-step view of the whole process is depicted in Figure 3.14. It is composed of the following steps:

1. **Deploy the updated code in S_2:** The newly composed program is deployed in S_2. Each NTMS and TMS element of the non-changing virtual switches of S_1 will have a corresponding element in S_2.

2. **Copy the NTMS elements:** The Data Plane Update Module and Runtime module copy all of the NTMS elements of interest while delaying requests from the virtual switches controllers.

3. **Activate migration mode in S_1:** Migration mode is activated in S_1 so that packets that change the TMS elements are augmented with the *Migration Header* before being forwarded to S_s.

4. **Synchronize the TMS elements:** As described in Section 3.4.1.

5. **Flush the packets in S_1:** After the states of S_1 and S_2 have been synchronized, S_s may briefly hold back incoming traffic from the hosts until the packets flowing between S_s and S_1 are done processing. Or, alternatively, it may drop a few packets still in S_1's pipeline.

6. **Reestablish the flow to S_2:** After the migration is concluded, the traffic from the tenants is redirected to S_2. From the point of view of the tenant's hosts, no change occurred in the network.

Step 5 requires special attention so that the data plane state of the migrated virtual switches remains consistent. If the traffic is just steered to S_2 (step 6), for a given virtual switch, new packets may arrive at S_s and be forwarded to S_2 before packets still at S_1 are forwarded to S_2, which may modify the data plane state. After flow is reestablished to S_2, it cannot be allowed for *Migration Headers* to modify S_2' state, as it may lead to inconsistencies. For instance, a virtual switch that implements a simple monotonically increasing packet counter could have the last copied value for that counter be X when the flow is reestablished, but a packet is still going through S_1, which will leave with a *Migration Header* containing $X+1$. Before that packet is sent to S_2, S_s sends an arbitrary number of other packets to S_2 from normal traffic, each increasing the counter in S_2. By the time the delayed *Migration Header* reaches S_2, the $X + 1$ value would be outdated, and the counter would decrease.

Therefore, the virtualization architecture provides two options for that step. One option is for the packets still flowing through S_1 to be dropped at S_s. The number of affected packets depends on the capacity of S_1's output buffers, its number of pipelines, and its stage count. The other option is to hold new traffic in S_s via recirculation or input buffers for a sufficient time until no more packets are flowing through S_2.

The options provided may not be viable if the system administrator wishes to provide tenants with higher standards of guarantees regarding packet loss. However, in many situations where resource utilization is not at maximum, simply steering the flow will not result in consistency problems. Recall that in S_1 every packet is forwarded to its input port, regardless of the forwarding decision of the virtual switch, which is only considered in S_s. Therefore, output buffers in S_1 can only increase if the additional space required by the *Migration Headers* is greater than the per-port unutilized bandwidth for a given average packet size. The relationship between those elements can be formalized.

During the migration process, if PPS denotes the packets per second at a given port, MHS the size of the ***Migration Header***, TP_{tenant} the bandwidth allocated for the tenant requesting the migration[5] (which is inactive), TP_{port} the throughput per port of the target device and APS the average packet size during migration, then:

$$PPS \times MHS \leq TP_{tenant}$$

$$\frac{TP_{port} - TP_{tenant}}{APS} \times MHS \leq TP_{tenant}$$

$$MHS \leq \frac{APS \times TP_{tenant}}{TP_{port} - TP_{tenant}}$$

Assuming that the switch is operating at maximum throughput, $MHS \times PPS$, must not exceed the fraction of the bandwidth allocated to the swapping tenant. MHS depends on the number of elements, their bitwidths, and their size in number of indexes. For instance, a register in the TNA architecture with 32 bits and 2^{16} indexes would incur 48 bits for the size of MHS. If n such registers could potentially be altered by a single packet, then MHS would be equal to $48 \times n$. Realistically, APS cannot be too low for hardware switches with guaranteed throughput. For instance, in Tofino switches, it has been observed that packets shorter than 256 bytes caused drops in maximum throughput (SCAZZARIELLO et al., 2023; ZHANG, 2022). So, for instance, in a scenario with 4 tenants with equal shares of bandwidth and APS of 256 bytes, MHS, in bits, would be:

$$MHS \leq (256 \times 8) \times \frac{1}{4} \approx 512$$

[5]Currently, the virtualization architecture does not provide a way to enforce per-tenant bandwidth. Therefore, this analysis assumes an external mechanism implemented by the system administrator.

3.5 Evaluation

To validate the proposed architecture, an evaluation with the core modules described in the earlier sections was conducted. The implemented modules are (i) Code Composition, (ii) P4Runtime, (iii) Security, and (iv) Data Plane Update. Two experiments were performed to verify the additional costs in management time (in terms of latency) of the proposed solution and the isolation and security of the virtual switches. The repository containing the current implementation's source code also contains scripts that allow for easy reproducibility of the results. Additionally, an experiment evaluating runtime deployment via the hot-swap strategy was conducted using Intel Tofino's simulator.

3.5.1 Composition and Latency Experiment

For the first experiment, two P4 programs that are extensions of the basic forwarding program for the P4 tutorial were designed, which contain a table to forward packets based on the destination IP address. The base code of the programs was also adapted to the Tofino Native Architecture (TNA), as the experiments were performed in an Intel® Tofino™ switch. This hardware switch can process packets with an aggregated throughput of 6.4 terabits per second.

The first program was extended to be able to receive packets from the controller, which are processed in a different manner from regular traffic by populating metadata fields containing information about the switch's egress queue occupation. The controller was designed to operate as a simplified in-band telemetry application (KIM et al., 2015), which periodically sends packets to the switch to get information about the egress queues. The experiment aims to measure the delay between the telemetry request (packet-out) and the response (packet-in).

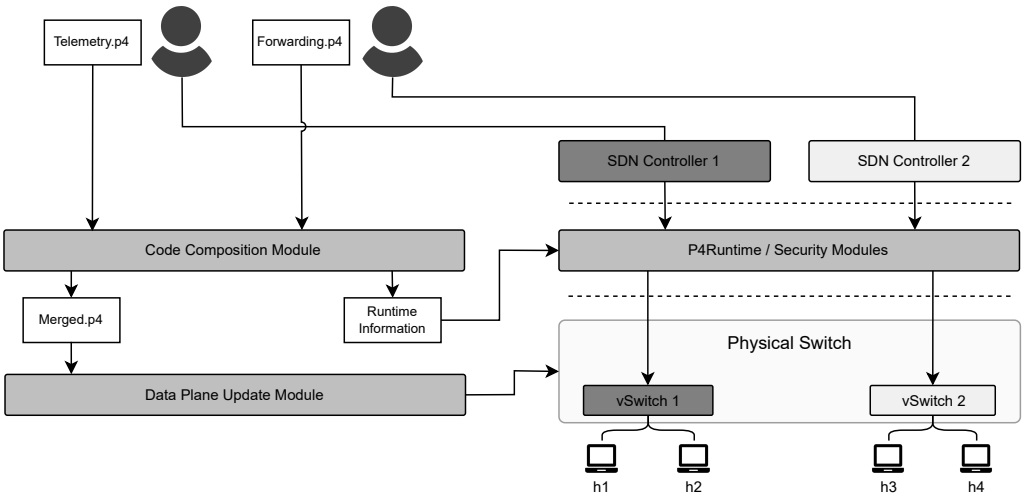
The second program was extended to send any packets that resulted in a table miss to the controller for further inspection, along with metadata that informs the packet's ingress port. This way, the controller can insert rules in the switch's table as needed, similar to a simplified reactive forwarding application. The rules inserted by the controller also contain an *idle timeout* field set to 5 seconds, which will cause the switch to generate a message to warn the controller of inactive entries (those that did not have matches in the specified period) so that the controller can take action such as removing them to save table space. The delay that is measured is the Round Trip Time (RTT) of the first packet

of a ping between two connecting hosts.

The programs were designed to evaluate the solution's ability to merge programs that pose some challenges, such as (i) different headers for packet-in and packet-out runtime messages, (ii) a table that can be shared to reduce TCAM hardware usage but such sharing must be transparent to the tenants, (iii) different parser and control flow logic, and (iv) runtime table entry idle timeout messages that must be translated to the controller's expected formats.

The test was conducted with (i) a controller application for tenant 1 that periodically (every 3 seconds) sends packet-outs to the switch (in this case, the P4Runtime module in the middle) and registers the delay to receive a packet-in in response and (ii) a controller application for tenant 2 that receives packet-ins from table-misses, inserts the appropriate rules and send the packet back as a packet-out and also removes any table-rules that generates idle timeout notifications. Both virtual switches and controller applications are running at the same time.

Figure 3.15: Topology for the composition and latency experiment.



Source: The Author

The complete scheme of the experiment is depicted in Figure 3.15. It shows the whole process of instantiating the virtual devices. It starts with the tenants submitting their programs to the code composition module. This module, in turn, generates the merged program and the mapping information needed by the runtime module. The data plane update module compiles the merged program, generating the binary configuration files that are deployed in the target host device. After the merged program's deployment and the initialization of the P4Runtime and Security modules, the tenants can have their SDN controllers connect to the server instantiated in the runtime module, just as they would

connect to the servers instantiated in a physical device.

Table 3.1: Delays measured on the latency experiment.

	Virtualization (ms)	**No Virtualization (ms)**
C1 (Telemety)	4.089 ± 0.16	2.197 ± 0.07
C2 (Forwarding)	18.82 ± 0.34	10.97 ± 0.34

Table 3.1 shows the delays for the telemetry packets for tenant 1 and the RTT of the first packet of a ping between two hosts of tenant 2. For comparison, the table also shows the same delays when only the code of one of the switches is running (i.e., no virtualization solution is employed). The results are an average of 30 repetitions. The controller applications used with and without virtualization are the same because the architecture provides transparent virtualization. The overhead of the experiments shows that the virtualization scheme has suitable performance for the chosen applications since the telemetry application only perceives a 2ms average additional delay every 3 seconds, and the forwarding application only incurs an 8ms average additional delay every time a new flow needs to be established.

The achieved bandwidth was limited by the hosts' CPUs capacity to generate data, which oscillated between 15Gbps and 30Gbps (per port). The specific Tofino switch used in this experiment guarantees 100Gbps throughput on each of its physical ports (in both directions) if the compilation process is successful. This makes it not relevant for this experiment to measure it thoroughly.

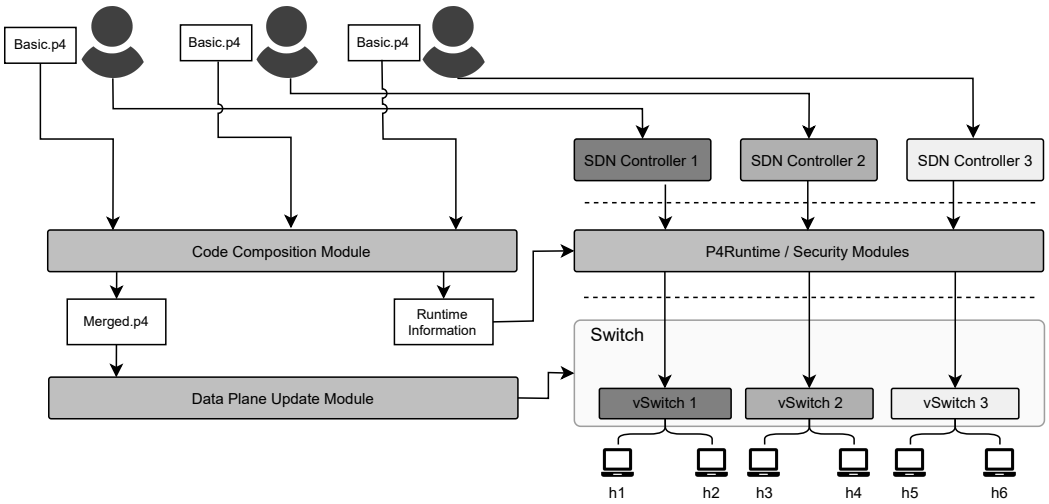
In summary, the experiment ascertained that the solution is able to compose programs with different pipeline control logic and packet metadata specifications and manage them in parallel in a completely transparent way to the tenant's controllers. The overhead delay, acceptable for the applications, is a small price for the advantage of running multiple virtual switches in parallel with real hardware line rate processing and throughput.

3.5.2 Isolation Test

In this test, to evaluate the implemented solution's isolation and security functionality, we design a scenario of composition involving three virtual switches from different tenants. We use the same program ("basic forwarding") for all tenants to demonstrate that the solution is able to correctly identify the permissions of each request and translate them accordingly. The main functionality of the virtual switches is to forward packets based on destination IP. Initially, each virtual switch's forwarding table has rules for forwarding

packets to connected hosts.

Figure 3.16: Topology for the isolation experiment.



Source: The Author

The scheme of this experiment is similar to the one depicted in Figure 2.14, which was used in Chapter 2 to illustrate the requirements of a complete multi-tenant virtualization solution. Figure 3.16 depicts the expanded scheme with the additional modules specific to this thesis's architecture. It shows the complete deployment process, similar to the previous experiment.

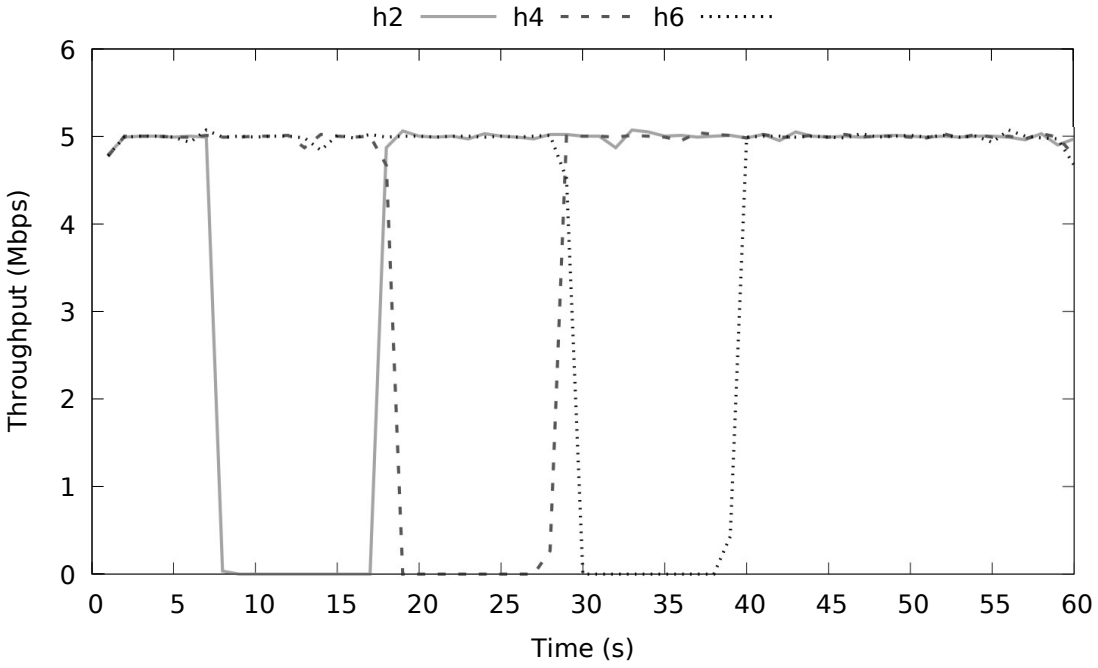
The controllers of each virtual switch are applications developed to insert exactly the same set of rules in the forwarding table of their virtual switches; the only apparent difference to the P4Runtime module is the credentials used to identify the *tenants*. Applications initially send requests to remove all rules from the routing table, wait 10 seconds, and send requests to reinsert the removed rules. The RPC requests sent by each application are exactly the same, but requests from a specific controller are expected only to affect hosts connected to ports on their virtual switch.

As this experiment is focused on evaluating the isolation and security of the solution and not its performance, the software switch bmv2 was used as the host. As previously discussed, the virtualization architecture allows for the implementation of the modules to be target-independent, which is also ascertained by this experiment.

The test was performed by starting each controller 10 seconds apart. Before the first controller starts, *iperf* flows are established between the devices of each virtual switch. Figure 3.17 shows the flows observed by *h2*, *h4*, and *h6*, which are the receivers of the *iperf* flow of each virtual switch. We can observe that the rules inserted by each

controller do not affect the virtual switch flow of others. This is due to composition operations, which add the virtual program ID as a key in each table.

Figure 3.17: Flows in the receptors on the isolation experiment.



Source: The Author

3.5.3 Runtime Deployment

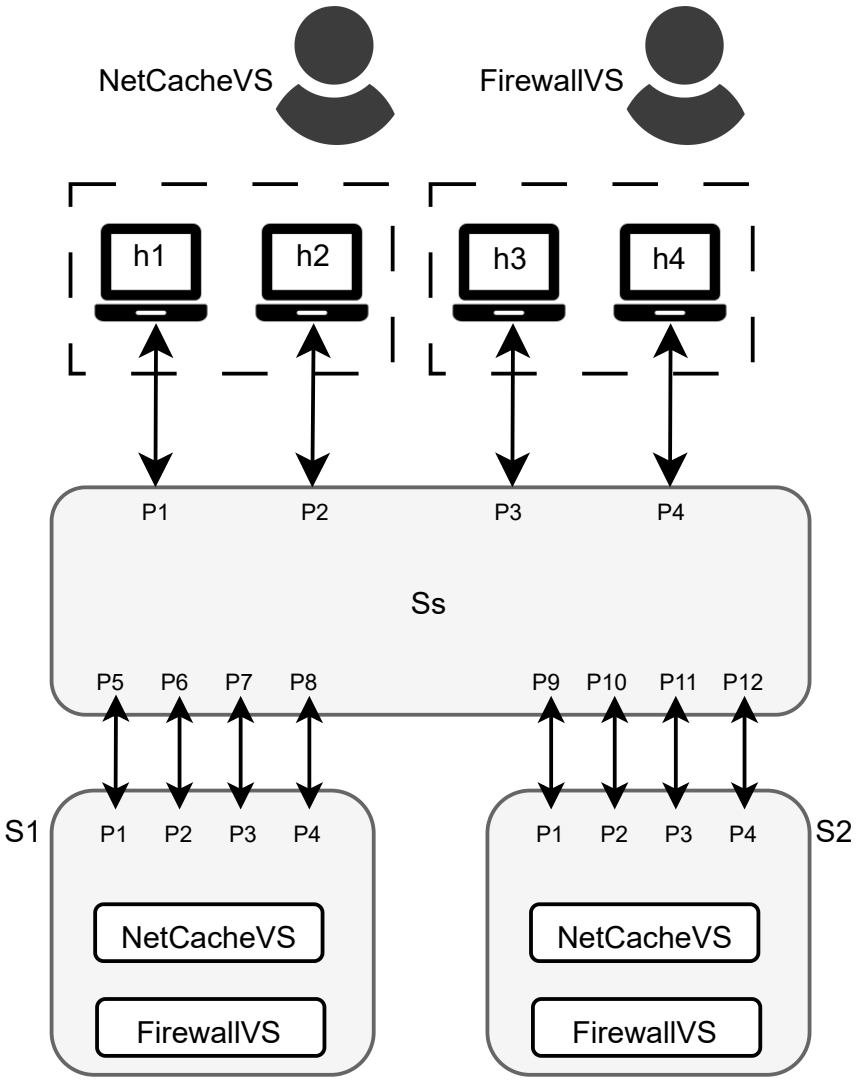
To evaluate the proposed runtime deployment strategy, an experiment was conducted to verify the data-plane-based state-migration algorithm for virtual switches (Algorithm 1). The experiments were conducted targeting Intel Tofino switches with the TNA PDP architecture.

This is a preliminary evaluation performed on the register-accurate Intel Tofino model provided in the P4Studio development kit, which can precisely simulate the behavior of hardware switches. The main disadvantage is the restricted performance of the simulator in terms of throughput and latency, in particular when running multiple instances as the strategy requires. At the time of writing, experiments in a testbed with multiple programmable switches are being planned as part of a future research paper.

For this experiment, two PDP applications featuring virtual switches were implemented. The first is a version of NetCache (JIN et al., 2017), that implements a key-value store scheme in which a server respond to client requests for key values. A programmable switch in the network path between the clients and the server implements a caching func-

tionality, which involves detecting frequent keys and storing their values in local memory to improve query times. The second is a stateful firewall that blocks incoming traffic from WAN ports without prior connection establishment.

Figure 3.18: Topology of the runtime deployment experiment.



Source: The Author

Figure 3.18 depicts the topology for the runtime-deployment experiment. The logical view provided for each tenant is that their host devices are attached to certain ports of the target switch (P1, P2 and P3, P4). In reality, their hosts are connected to the first 4 ports of S_s. S_s is configured to forward traffic to either S_1 or S_2, however is the current active programmable switch. S_s is also responsible for adding or removing additional headers containing virtualization metadata to each packet and assisting in the migration process, as previously explained in Section 3.4.

In the experiment, the Netcache client runs in $h1$ and the server runs in $h2$. Simi-

larly, the firewall virtual switch process flow from an iperf connection between $h3$ (client) and $h4$ (server).

In the reference implementation of the virtualization architecture, the Management Interface module is responsible for triggering a hotswap process. Before the process starts, the module collects the updated programs for each tenant that require changes in their virtual devices and generates the new composed program. The Management Interface module then provides the Data Plane Update module with (i) the new composed program and (ii) the list of virtual switches that are unchanged, and thus must have their states preserved through the migration algorithm and (iii) a mapping of the stateful elements of the unchanged virtual switches in S_1 to their counterparts in S_2. Assuming that the current active host device is S_1, the updated program is then deployed in S_2. For this experiment in particular, both of the programs are marked for migration, so all of the elements in S_1 must be migrated to S_2.

Table 3.2: Results for the runtime deployment experiment.

	Without Hotswap				With Hotswap			
	100Kbps	**200Kbps**	**300Kbps**	**400Kbps**	**100Kbps**	**200Kbps**	**300Kbps**	**400Kbps**
Netcache Retry Count	0	2	12	9	3	10	59	83
Netcache Total Time (s)	240	287	310	333	301	345	364	389
Time to Deploy (s)					172	168	140	106

Table 3.2 contains the results for the experiments with multiple configurations. The netcache client is configured to send 2048 random requests with a 1-second retry interval if no requests. The iperf connection of the firewall clients is configured to run for 300 seconds with varying bandwidths. The chosen bandwidths are relatively low (in the order of Kbps) due to the high processing requirements of the multiple Tofino simulators. The hotswap process starts 20 seconds after the flows from the clients are established.

The results show a monotonic increase in the time required for the Netcache client to complete all requests across both versions of the experiments (with and without flow steering). It was verified that the increase in traffic processed by the Tofino switch (packets per second) was responsible for additional latency incurred by Netcache requests. This demonstrates the simulator's inadequacy for experiments, even with meager throughput.

The experiment shows that the algorithm can steer traffic to the standby switch. However, current experimental results are insufficient to ascertain the impact of the proposed hot-swap process in real hardware targets. Further evaluation is necessary and is planned to be conducted in the near future.

4 TOPOLOGY EMULATION INTEGRATION

This Chapter presents an application for the architecture described in Chapter 3, which is the main contribution of this thesis. It begins by describing the concept of network topology simulation and emulation, and its applications. Then, it discusses the current approaches and their limitations with a focus on SDN / PDP networks, motivating the advantages of integration with the multi-tenant virtualization architecture. Finally, it describes how this integration can be achieved, discusses the benefits compared to traditional scenarios, and provides experimental results of reproducible experiments and a proof-of-concept implementation.

4.1 Network Topology Simulation and Emulation

As with many fields of research, computer network innovations must be prototyped in safe and controlled environments before being deployed in production. Besides that, some applications, such as educational courses, benefit from an agile "sandbox" environment with fast deployment. For those and other use cases, network **simulators** and **emulators** can provide this agility.

Network **simulators** are computer programs that compute the interaction between hosts in a network using a network model for elements such as hosts, forwarding devices, links, and even applications. Normally, such models operate in a pre-determined level of abstraction. For example, one might want to verify the implementation of a novel transport layer protocol in a simulated environment, without the need for an accurate implementation of lower layers.

Since those programs are based on models, they cannot be perfectly suited for extensive testing of certain elements of network protocols that might depend on specific hardware. Nevertheless, simulators provide a controlled, reproducible, easy-to-log and debug environment that suits educational and research purposes.

A classical open-source discrete event network simulator for computer networks is NS (network simulator), which is a robust software that has evolved since 1989 and still continues to receive community support (RILEY; HENDERSON, 2010). This simulator contemplates all traditional computer network layers above the physical layer of electrical signals, making it possible to simulate and log a broad range of operations, from as low-level as ethernet switching to as high level as DNS name resolution.

As previously mentioned, NS is open-source, and its C++ implementation can be extended to accommodate more specific use cases. Due to that flexibility, many extensions have been developed for NS, such as the support for parallel network acceleration (SZYMANSKI et al., 2002), visible light communication (ALDALBAHI et al., 2017), and flow monitoring tools (CARNEIRO; FORTUNA; RICARDO, 2009), among others. Different network simulators also exist with a similar purpose, such as OMNeT++ (OMNET++, 2023), but NS' flexibility makes it stand out as a preferred environment for prototyping. In a greater level of detail, Campanile et al. (2020) presents a systematic literature review of network simulators and their applications with a focus on NS.

Lately, even SDN concepts have been implemented as extensions to NS, such as the separation of the control plane logic from the data plane forwarding. There are extensions that allow for the "deployment" of OpenFlow switches in an NS simulated network topology (UNICAMP, 2023). Beyond that, more recently, even PDPs started to be contemplated by NS. Multiple efforts aim to simulate forwarding devices that can be programmed with a subset of the P4 language specification (BAI et al., 2018), although these solutions still lack maturity for large-scale experiments.

Still, there are disadvantages of those simulators, specifically the difficulties associated with prototyping, which include the limited model granularity and hidden implementation details of the end-hosts applications that cannot be fully grasped. Moreover, the underlying platform for describing simulations may be incompatible with the "real-world" ones in which the innovative protocols and algorithms will be deployed. Using NS as an example, researchers need to describe simulations using C++ code, but the production deployment is targeted for P4 or HDL-enabled devices. This entails the inevitable difference between the artifacts that are required for the implementation of the prototype in the simulator, such as program source codes, and the actual artifacts deployed in production. Migrating to real-world implementation requires some level of reshaping of those artifacts, which is not a trivial process and may be error-prone. The alternative to mitigate those limitations is network **emulators**.

On the other hand, **emulators** aim to mimic the behavior of a certain system on a more precise level, especially the hardware details. In the field of computer networks, the most prominent emulator is Mininet (LANTZ; HELLER; MCKEOWN, 2010). It is an open-source software that creates a virtual network of hosts and forwarding devices inside the host operating system. Even more complex scenarios, including wireless networks, can also be emulated with Mininet.

The main advantage of using such a tool is the possibility to run "real" applications (that is, the exact same code) on emulated end hosts and network servers that communicate through virtual interfaces. Mininet, for instance, allows for virtual hosts and forwarding devices to be instantiated, and commands can be executed on those hosts just as they would on a "real" system. This leads to a more realistic evaluation scenario, addressing the aforementioned issues related to simulators.

However, emulators usually require a significantly higher processing power than simulators, due to real-time processing. Consequently, time and performance-dependent algorithms, such as TCP's congestion window management, cannot be thoroughly evaluated in those scenarios, as the throughput will be limited by the host hardware. As a result, large-scale tests with hundreds or thousands of nodes must be delegated to the discrete-event simulators, which can take and arbitrarily large amount of time to conclude a test case. Moreover, while testing with software that runs on real hosts is closer to the production environment, it prevents fine logging and debugging during the tests, which may be required in certain scenarios.

Previously in Chapter 2, in the section about PDP virtualization (Section 2.8), it was shown that many of the existing solutions were tested on network emulators. Such an environment may be suited for a functional evaluation of the solutions, but it is entirely inappropriate for performance evaluation. It is nearly impossible to isolate the performance impact caused by the overhead in the host's hardware imposed by the emulation solution from the impact of the virtualization solution itself. Consequently, there is no clear way to generalize the results to real forwarding devices' hardware.

4.2 Recent Advances in SDN and PDP emulation

Network emulators such as Mininet, in contrast with simulators such as NS, are heavily focused on the SDN paradigm. This can be attributed to the different age in which these software were developed. As it was mentioned, network simulators have existed since the 1980s, a time were network flexibility was not a concerning issue. Emulators are more recent, as they needed more processing power than what was available at that time, and when they became viable in the 2010s with the release of Mininet, SDN was the leading paradigm at the core of computer network innovations. There are a number of surveys, such as the one from Oliveira et al. (2014), that compare the different advances in SDN enabled by emulators.

When learning about PDPs, the first hands-on contact with programmable devices one has is likely the tutorials of the P4 language, which are deployed in the software switch bmv2 running on Mininet. Most PDP solutions are developed and evaluated in such testing systems. Still, as previously mentioned, hardware testing is also required for high-performance evaluation.

There are some recent efforts to enable topology emulation similar to Mininet's for prototyping purposes that take advantage of real hardware devices' high performance (RODRIGUEZ et al., 2023) or even mixed scenarios of hardware-based / software-based devices (BUZURA et al., 2023). However, there are a series of limitations on the current approaches that prevent the reproducibility of the experiments, mainly the heavy target-dependency. Also, no solution enables multiple topologies from different tenants of virtual switches running on real hardware, which would benefit organizations with multiple researchers and limited resources.

This current scenario motivates the design of a solution for multi-tenant topology emulation that leverages PDP virtualization. The solution can be designed to escalate with the available hardware (the number of physical forwarding devices and their capabilities).

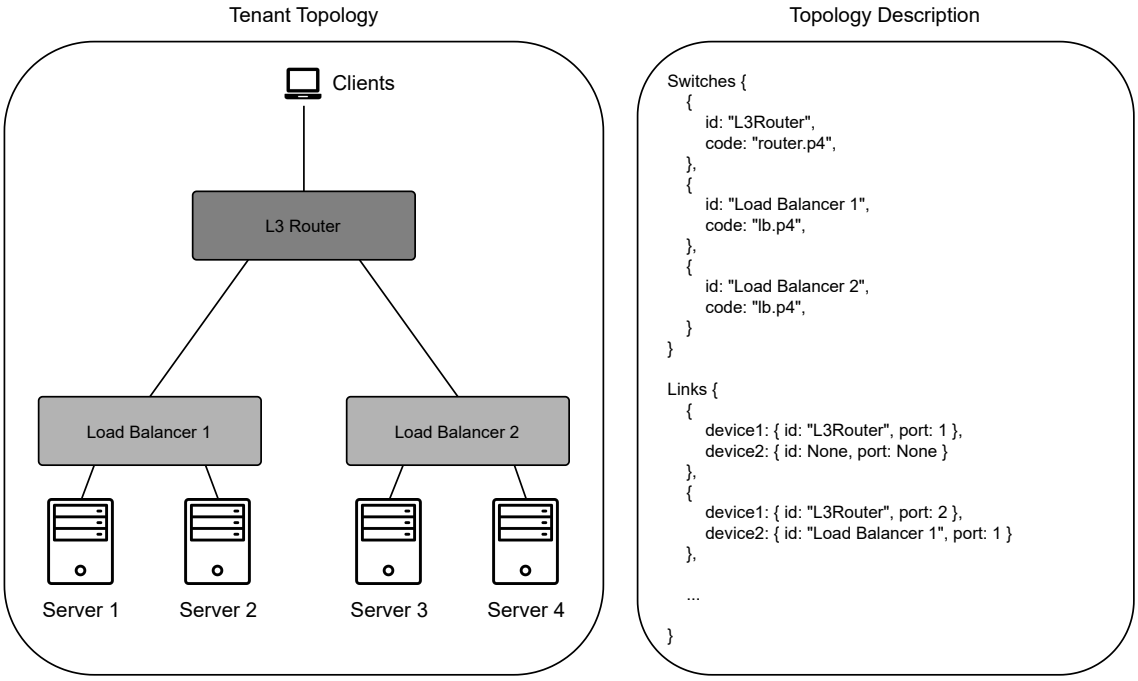
4.3 Multi-Tenant Hardware-Based Topology Emulation

Using the virtualization architecture from Chapter 3, which is designed for target-independent multi-tenant virtual switch instantiation and management, the current limitations of hardware-based topology emulation can be overcome. This can be achieved due to the modular approach of the presented architecture, allowing for extensions to handle multiple virtual topologies.

Figure 4.1 shows an example of a virtual topology defined by a user (or tenant) of the system. On the left, it depicts the elements of the network that the tenant wants to instantiate, including the virtual switches (rectangles) and the hosts (clients and servers). It also depicts the links between the interfaces of those elements. On the right, a textual description of this topology is shown, which serves as input to the system.

The input configuration files contain information about the virtual switches, including their P4 codes, and the links between them in a style similar to JSON. The servers and end-hosts of the system are denoted by entries with ID "None" in those files.

Figure 4.1: Example of a tenant-defined topology and its text description.



Source: The Author

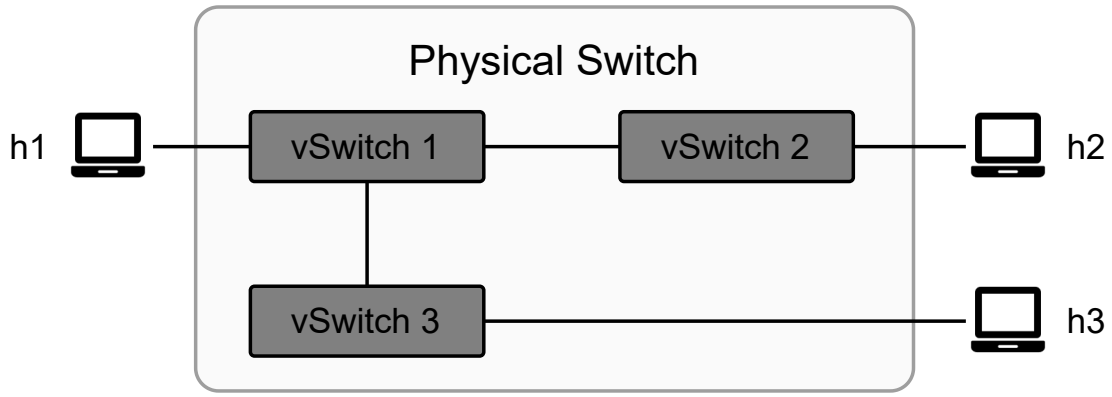
4.4 Experimental Results

To demonstrate the advantages of the proposed solution, two preliminary experiments were conducted. Their goals are (i) showing the performance advantages of topology prototyping leveraging real hardware enabled by the proposed architecture in comparison to software-based emulation and (ii) verifying the hardware resource usage when accommodating topologies from multiple tenants simultaneously.

4.4.1 Performance Test

The first experiment replicates a scenario described in a previous work of this thesis' author (BUENO et al., 2022) that, at the time, utilized bmv2 switches in conjunction with Mininet and a modification of P4VBox (Saquetti et al., 2020). The goal is to compare the control plane latency in inserting table rules with new state-of-the-art programmable switches. An Intel Tofino switch was employed for this experiment with the multi-tenant topology solution. Figure 4.2 depicts the topology defined in that experiment. In the original experiment, all of the virtual switches implemented simple L2 forwarding. The experiment consisted of a flow from h1 to h2, passing through vSwitch 1 and vSwitch 2,

Figure 4.2: Topology of the latency comparison experiment.



Source: The Author

which is at some point redirected to h3, passing through vSwitch 1 and vSwitch 3.

Initially, all of the switches start without any forwarding rules on their tables. The rules are inserted by a controller (omitted in Figure 4.2) when required. The measured delays are the measured time the RPCs from the controller take to return. The flow redirection to h3 is also a rule inserted by the controller.

Table 4.1: Comparison of the different targets used in the virtual topology experiment.

	bmv2	**P4VBox**	**MTTS**
Latency	18ms	7ms	5ms
Throughput	50 Mbps	10 Gbps	17.9 Gbps*

Table 4.1 shows a comparison of the results obtained in the previous work with the results obtained using the proposed multi-tenant topology solution (MTTS in the table). The results are an average of 30 repetitions, the same used in the original work.

The latency observed is competitive with the P4VBox solution. In the original work, the authors used a Python controller and hypothesized that a more efficient controller written in C/C++ could reduce their latency overhead, which is implemented in MTTS. This could explain the 2ms average improvement over P4VBox.

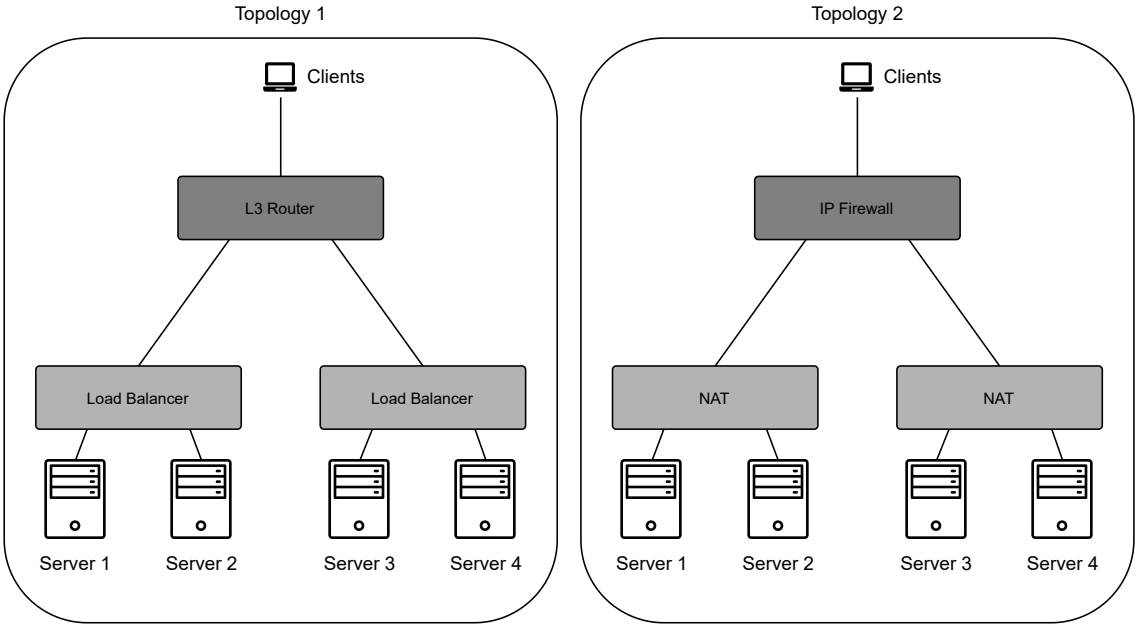
The hardware throughput, however, is where the advantages of MTTS become more apparent. In the experiment, MTTS was able to achieve 17.9 Gbps throughput, but it was limited by the host capacity of generating traffic. As previously mentioned, hardware switches typically guarantee no throughput loss, therefore 100 Gbps could be achieved. One possible future work from this thesis is to repeat this experiment with SmartNICs capable of generating high-throughput traffic.

The results with *bvm2*, both in terms of latency and throughput, once again illustrate that software-based switches are not appropriate for performance evaluation. Their main advantage is the functionality evaluation of the innovative protocols. However, if those protocols call for specific timing and performance guarantees, hardware-based prototyping is necessary.

4.4.2 Resource Usage Test

The second experiment was designed to ascertain the hardware resource usage of the emulation solution and the virtualization architecture in general. As previously mentioned, the lightweight virtualization scheme employed for the architecture implies implicitly sharing the underlying hardware between the guests. Still, the virtualization solution itself implies hardware overhead, as control structures need to be implemented to guarantee isolation and to compensate for target-specific requirements.

Figure 4.3: Topology of the resource usage experiment.



Source: The Author

Figure 4.3 shows the setup for this experiment, which involves two isolated topologies from different tenants emulated inside a single hardware switch. In the experiment, the links in the topology are physical cables connected between ports of the switch belonging to each virtual device. Of the available 32 ports of the host target employed, 18 were used in total.

Table 4.2: Resource usage of the individual programs and of the merged program.

RAM Blocks	ipv4_t	firewall	src_t	dst_t	nhop	lb_table	Overhead	Total
L3 Router	5							5
Load Balancer					4x2	4x2		16
Firewall	4	4						8
NAT	4x2		4x2	4x2				24
Merged	14	4	8	6	4	5	5	46

The resource usage in terms of RAM allocated for each table in the hardware for each of the individual programs (when compiled separately) is shown in Table 4.2. The usage of resources for tables in the merged program that were merged from multiple tables from individual programs is less than the sum of the usages of the individual programs. For example, the merged program contains an *ipv4_t* table that was merged from 4 input programs which occupies 14 blocks of RAM, but the sum of the usage for the individual programs is 17. This can be explained by multiple factors, such as table hardware optimizations which result in sub-linear space growth as the number of entries increases (which is a target-specific feature) and that the actions associated with each table are only instantiated once in the merged program (even without hardware optimizations).

The additional overhead of the virtualization solution comes from the tables and parser stages needed to isolate the virtual devices, such as the table that verifies that the assigned output port of the packets belongs to their virtual device. The results show that, depending on the degree of compatibility of the tables in the programs, it is possible to compensate for this additional overhead. For this test case, the total number of blocks used (46) is 21% less than a composition without merging tables (58).

More complex program codes, programs with a low degree of structure compatibility, or a larger number of virtual devices will inevitably exhaust the available hardware resources. A more thorough resource exhaustion test and analysis on the fair sharing of resources among the tenants is left for future work.

4.5 Planned Features

One possible extension to the presented solution is the integration of multiple hardware targets to expand the topology size limit. Testbeds with many hardware targets could utilize them in unison, given a mapping of the virtual devices of the topologies to the physical devices. Each host target would be responsible for implementing a subset of the virtual devices defined in the topologies.

Another goal is the integration of the hardware emulation testbed with other projects, such as Skynet (BUENO et al., 2022), which aims to provide data plane integration with artificial intelligence and machine learning algorithms. Currently, only software-based programmable devices (bmv2) are employed for the evaluation of the works of the project due to the large number of devices required. By utilizing the presented topology emulation testbed it would be possible to ascertain the capacity to perform the complex calculations involved in the AI and ML algorithms at real hardware line-rate.

5 SHARING VIRTUAL SWITCH RESOURCES

The proposed PDP virtualization architecture achieves one of its goals, multi-tenant support, through logical isolation of each virtual device belonging to different tenants. To recall, this isolation is provided mainly by the Code Composition, Runtime and Security modules. Before deployment, the Code Composition module augments the merged programs to include additional checks to provide isolation (such as dropping packets from tenants that are scheduled to a port belonging to a different tenant if port partition is available). After deployment, the runtime / security modules impose barriers and safeguards through message translation to restrict access to the composed elements by the controllers.

However, there are scenarios in which **sharing resources and state** may be interesting to the virtual switches (LAMB et al., 2024). For instance, if multiple virtual switches implement a Denial of Service (DOS) mitigation scheme through a firewall table, they will both declare a similar table in their programs. As mentioned, the lightweight virtualization scheme employed may be able to merge those tables in the generated code while keeping them logically isolated by adding a key field to the entries. Still, in the DOS application, it may be that multiple entries exist for a given key. Assuming the keys to this table in the original programs are the IPv4 (32 bits) or IPv6 (48 bits) addresses of potentially malicious hosts, each key can potentially be cloned in the merged table once for every program ID. If the tenants agree to share the entire table, the table's size can be significantly reduced, as multiple entries for a single key would be removed.

Another use case in which explicitly sharing the host's resources would be beneficial relates to the concept of *modular dataplane programming* introduced in Section 2.5. As discussed, one of the problems of the P4 language is its inherently monolytical design (e.g. programs defined by single translation unit, abuse of preprocessor directives, etc.). While some previous works addressed this problem by defining novel DSLs which could then be translated to P4 code (SONI et al., 2020; GAO et al., 2020), this thesis provides a novel approach to the problem leveraging PDP virtualization. A complex program defining the behavior of a switch could be divided into multiple programs, to be implemented as virtual switches, and their shared interfaces could leverage the virtualization architecture feature of explicit resource sharing. In contrast to the previous works on modular dataplane programming, this approach does not require the definition of a novel PDP DSL. As such, a lesser burden is imposed to adapt the existing code.

The mechanisms described in this Chapter require some level of trust between the tenants of the system and, to some extent, breaking the logical view that their devices are the only ones executing in the target. If a single entity[1] owns multiple virtual devices and is acting as multiple tenants, there is no need to establish trust. A mechanism to establish trust among the tenants is out of the scope of this thesis, and it is not required by tenants who wish to opt out of sharing resources altogether, as the default behavior is not to share anything among the virtual switches.

5.1 Augmenting the PDP programs to implement resource sharing

In order for tenants to indicate which resources they want to share and what permissions the other tenants can have in them, the virtualization architecture must be extended. This could be achieved by changes each tenant makes to the program code of each virtual switch, or by additional information provided by the tenants, along with the virtual switch's code before composition, to be appended to the *tenant database* of the virtualization architecture.

The first option was chosen due to the official support built into P4. The P4 programming language specification defines *annotations* as a mechanism for small language extensions. This mechanism is often used by vendors to define specific parameters that only exist in their targets.

Having tenants augment (or modify) their source code by adding annotations does not conflict with the design goal of not requiring modifications to the original programs. As the use case is explicitly about sharing resources among virtual switches, it does not apply to existing code that certainly did not consider virtualization. Moreover, this design decision will not impose any additional burden on the tenants than the alternative would (requiring them to specify shared element information through another means to populate the tenant database).

Figure 5.1 contains an example of a header declared in a P4 program developed for Intel Tofino switches (TNA). This specific target can optimize the resulting binary code during compilation by relaxing the requirements of header fields that it knows are not required by the tenant, allowing arbitrary writes to those fields during runtime. The

[1] As it is assumed by the terminology used throughout this thesis, the mapping of "tenants" to virtual switches is 1 to 1. The word "entity" was chosen for cases where multiple virtual switches have the same "owner". As such, the mapping of "entity" to "tenants" may be one to many.

programmer indicates such fields with the special "padding" annotation. The example in question is the same used in Section 3.2.2, augmented with the annotations. Note that, for this specific use case, the target's compiler cannot make the assumption that unused fields in the program are safe to be rewritten, as other devices in the network may require the information on those fields (such as the source IP address in this example). Therefore, this annotation must be manually placed by the programmer.

Figure 5.1: Example of annotations in header fields.

```
header ipv4_t {
    @padding
    bit<14>   _unused1;
    bit<2>    ecn;
    @padding
    bit<48>   _unused2;
    bit<8>    ttl;
    @padding
    bit<8>    _unused3;
    bit<16>   hdrChecksum;
    @padding
    bit<32>  _unused4;
    bit<32>  dstAddr;
}
```

Source: The Author

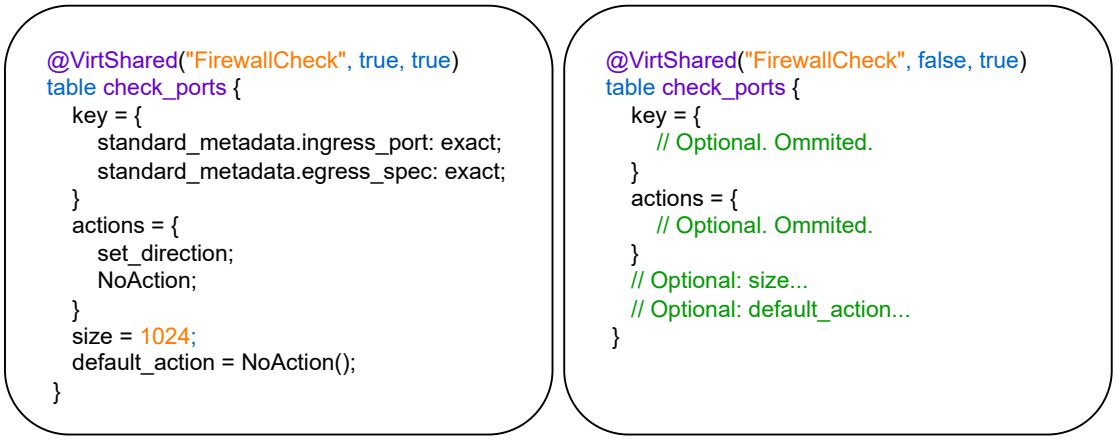
Some annotations may be supported by multiple targets from different vendors. This is the case with annotations from the P4Runtime CDPI, as it is an open standard supported by many targets. Any target that implements P4Runtime will recognize those annotations. For instance, the "*controller_header*" annotation specifies headers that should be converted to and from P4Runtime metadata. When sent to the control plane (PacketIn), those headers are stripped from the packet, and their data is used to populate fields in the protobuf[2] messages. When sent from the control plane to the data plane (PacketOut), the metadata fields of the messages are serialized into the packet, just like a "normal" header. This was depicted earlier in Section 3.3 (Figure 3.12). Note that it is possible to add arguments to the annotations, such as "packet_in" and "packet_out" in the example.

The virtualization architecture introduces the "*VirtShared*" annotation to tables and registers of P4 programs. It is used by tenants to specify which elements are shared with other tenants. The annotation contains the following arguments:

[2]Protocol Buffers (protobuf) is the language-independent message format of gRPC messages (GOOGLE, 2026).

- **Name:** The name of the shared element. It is used to identify the elements among multiple programs. Tenants who wish to share an element must agree on a name beforehand and use it in their programs.
- **Owner:** A boolean value indicating if that program's tenant is the owner of the element. Each element has exactly one owner who may specify access control policies for that element. An error is raised during composition if multiple programs declare themselves as the owner of an element.
- **Writable (by non-owners):** A boolean value indicating if non-owners should be able to write to this element. If the value is *false*, only the owner may write, and the other tenants can only read. This argument is only considered in the context of the owner's program to determine if a table is writable. The non-owner programs should populate this argument with the same value. If there is a mismatch, an error will be raised during the composition.

Figure 5.2: Example of the *VirtShared* annotation.



(a) Owner program. (b) Other program.

Source: The Author

Figure 5.2 shows an example of the use of the proposed annotation. The owner program code (a) declares the table with all its components (keys, actions, size...) and annotates it to indicate that it wants to share it with other tenants. The other tenants (b) that also want to access the table also declare the same table, annotated with the same *Name* argument and appropriate flags. As the table owner is the authoritative source for its components, other tenants may optionally omit them from their source code. If the components are present, the code composition module verifies that they are equal and raises an error if a discrepancy is found.

5.2 Application in Modular Programming

This Section describes a use case for the proposed mechanism for resource sharing involving modular data plane programming. It describes a work on a data plane inference engine developed in conjunction with the virtualization architecture of this thesis (LAMB et al., 2025). Without a virtualization scheme, the inference engine requires heavy modifications to the PDP code, which is not ideal for broad adoption. However, with virtualization, we can leverage modular data plane programming to decouple the inference engine from the program code, providing almost seamless integration for programmers.

5.2.1 Graph Neural Networks

In recent years, the adoption of Artificial Intelligence (AI) and Machine Learning (ML) in many areas of computer networks has grown substantially, including the area of computer networks. In this context, AI and ML could be fundamental building blocks of self-driving networks (FEAMSTER; REXFORD, 2017; LUIZELLI et al., 2021), supporting solutions for tasks such as congestion control, traffic engineering, or anomaly detection (WINSTEIN; BALAKRISHNAN, 2013; BERNáRDEZ et al., 2021). Combining AI and ML with SDN and PDPs to run novel solutions directly on the data plane could achieve unprecedented fine decision granularity and low response times.

GNNs are a model of ANN that handles graph-structured data (SCARSELLI et al., 2009). This model overcomes the shortcomings of converting graphs to tabular data to be consumed by traditional ANN models. GNNs can generalize the trained models to different graph topologies, attracting the attention of researchers working on many different areas[3]. Many research problems in computer networks have properties or structures that make them suitable for modeling as graphs. Network topology, for instance, is often represented as a graph. Therefore, GNNs are a powerful tool for ML studies in the computer networks area with a variety of applications such as intrusion detection (PUJOL-PERICH et al., 2021b) and performance estimation (RUSEK et al., 2020).

One of the most used architectures for GNNs is the Message Passing Neural Network (MPNN) (GILMER et al., 2017). In this architecture, the nodes and edges of the graph have "hidden state" vectors that encode features of those elements and are iteratively shared with their neighbors.

[3] A comprehensive survey on GNNs is provided by (WU et al., 2021).

- **Message:** A GNN iteration starts with each node producing a message to its neighbors. A *message* operation is modeled by a function that may take the hidden state of either the sender/receiver node or the edge connecting them as input.

- **Aggregation:** After receiving all the messages from its neighbors, each node produces a resulting vector with a function that aggregates the received messages. Typically, the aggregation function involves an average of the received messages.

- **Update:** Each node updates its hidden state with a function that takes as inputs its current hidden state and the produced aggregation vector.

After a defined number of iterations of these three operations, a readout operation extracts the useful information for each node or edge or for the entire graph. The application then uses this information as it would use the result of a traditional calculation. Typically, the functions for update and readout operations are implemented with trainable ANN models, such as Multi-Layer Perceptrons (MLPs). The parameters of the trainable components of the GNN are adjusted via gradient descent on a loss function defined over the difference between GNN predictions and target values from the dataset.

5.2.2 Integration of GNNs in PDPs

PDP devices, in turn, can derive/extract inputs and metrics such as queue occupation, flow rate, and round-trip delay, and exchange this information with neighboring nodes of the GNN. Those nodes can be mapped to the next-hop network devices when the GNN topology is similar to/equal to the network's topology. Additionally, with PDPs, it is possible to leverage existing hardware without requiring network operators to acquire new GNN-enabled forwarding devices, as they are reconfigurable. By adopting a modular data plane programming approach, the time necessary to adapt existing programs can be largely reduced. In this sense, PDP virtualization may be a deciding factor in enabling this workflow.

Existing research on GNNs applied to networking has either focused on modeling network behavior (FERRIOL-GALMéS et al., 2022) or determining network operations & management actions in the control plane (RUSEK et al., 2020). However, ultra-low-latency packet processing in switches and ever-increasing link speeds make it challenging for AI/ML applications running on the control plane to keep up with network management

decisions at the scale of hundreds of nanoseconds. Such response times may be required for some fields of application, such as security. As such, a novel data plane-based GNN classification mechanism is proposed in this thesis. To the best of my knowledge, this work is the first to present a preliminary discussion of GNNs running in a distributed manner directly in PDPs.

As part of the investigation, it was found that the current PDP hardware is capable of running limited pre-trained GNN models for inline classification. Therefore, the main idea is to first perform an offline training step to obtain the models and, afterwards, translate them into data plane code (P4) that can be compiled and deployed at the target. Applications that require online training are out of the scope of this work.

The work was evaluated with a use case for controlling congestion in data center networks based on the Data Center Transmission Control Protocol (DCTCP). DCTCP was originally proposed to improve the throughput and delay of TCP traffic in a data center. DCTCP is a modification of the classic TCP that leverages Explicit Congestion Notification (ECN). Two implementations of DCTCP in P4 were implemented for comparison, one with a classical monolithic approach and ECN markings based on queue occupation, and the other with a modular design where a GNN decides the markings.

5.2.3 Evaluation with Simulations

For the initial feasibility evaluation, an algorithm was developed to be deployed on network core devices to implement the MPNN GNN architecture. To evaluate this algorithm, Network Simulator 3 (NS3) (RILEY; HENDERSON, 2010), a popular simulator for computer network experiments[4], was employed. The simulated experiments allow for fine-tuning the algorithm in a more flexible environment before proceeding to the actual PDP implementation.

5.2.3.1 Case-Study: DCTCP ECN markings

As a case study, it was evaluated whether GNNs can replicate the widely deployed DCTCP protocol (ALIZADEH et al., 2010). Essentially, it is a modification of the classic TCP that leverages ECN markings on packets so that the transmitting hosts can react faster to queue buildup in intermediate switches, as opposed to the classical TCP approach

[4]Network simulators were previously described in Section 4.1

of detecting congestion by packet loss. More precisely, switches set the *Congestion Experienced* codepoint in packet headers when queue occupancy exceeds a threshold. This threshold (and, more generally, the packet marking algorithm) is opaque to TCP implementations running on end hosts and is usually set to a small value to ensure low latency.

We can model DCTCP's packet marking as a GNN classification task. Given a snapshot of the network topology at a given time, we model each queue in a forwarding device as a GNN node, with its hidden state initialized using queue-occupation information. For the edges, there are multiple approaches to determining a node's neighborhood. Here, we create an edge from a GNN node representing a queue Q_1 to another GNN node representing queue Q_2 if packets leaving Q_1 may be queued at Q_2. In other words, we connect nodes Q_1 and Q_2 if Q_1's output port is connected to the switch owning Q_2. Other possible neighborhoods may involve connections only between more directly related nodes, such as queues that packets visit in sequence, which leads to fewer edges in the GNN graph and may reduce communication overhead. This formulation illustrates one possible advantage of GNNs over classical approaches: Allowing each queue to leverage information about the queue occupation of its neighbors in its marking decision, as opposed to DCTCP, where the decision depends only on local queue occupancy.

5.2.3.2 Dataset Generation and Model Training

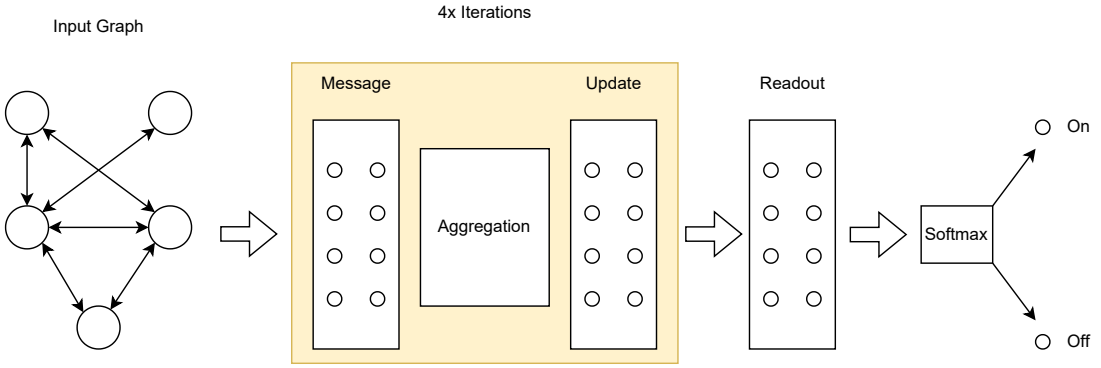
Before implementing the GNN classification framework on the simulator, a dataset was collected to train a GNN to perform ECN markings for DCTCP based on local and neighboring queue occupations. This was performed with a simulation of a data center using a fat tree topology with eight top-of-rack switches, each with four end hosts connected to them. Links between end hosts and the switch have 1.0Gbps speed and 10 μs latency. Links between switches have 1.5Gbps speed and 10 μs latency.

For each queue in the network, its occupation was recorded every 100 microseconds for 30 seconds. Each of the 32 hosts sends data sporadically to another host through a DCTCP connection throughout the experiment. This scenario was used for all experiments to compare performance. This experiment generated a temporal dataset with 300,000 samples, each representing a snapshot of the network at a given time, which was reduced to approximately 38,000 graphs by considering only those in which at least one queue marked a packet with ECN.

Building a labeled dataset for training the GNN to mimic DCTPC's ECN markings is rather straightforward since the decision is purely based on a threshold on the queue

occupations. To further explore the extra possibilities GNNs bring to the table, a new feature was introduced in the dataset that contains information about the rate at which each queue is filling up. This is computed as the difference between the occupation at the current snapshot and the previous one. The hypothesis is that knowing the neighbors' queue occupation trends will allow for a better-informed final decision. The model was also trained to try to predict ECNs some time steps ahead of the observed in the dataset. This was done by training the GNN to predict the ECN markings at snapshot t using queue occupation information at a previous snapshot. This procedure potentially allows the GNN to react earlier to changes in the network traffic, which may further improve TCP performance.

Figure 5.3: GNN Experiment Model.



Source: The Author

After acquiring the dataset, the MPNN GNN was trained with the TensorFlow-based GNN training kit iGNNition (PUJOL-PERICH et al., 2021a). iGNNition was designed for fast prototyping with GNNs in network scenarios, including examples of well-known model descriptions, such as RouteNet (RUSEK et al., 2020). The GNN model (shown in Fig. 5.3) is a simple MPNN with four iterations of *Message* (MLP), *Aggregation* (Average), and *Update* (MLP) steps followed by a *Readout* (MLP) for each node. After the Readout, a *Softmax* operation is applied to the results to produce a binary value. This binary value determines whether incoming packets should be marked with ECN.

5.2.3.3 Implementation

NS3's Random Early Detection (RED) queues were modified to implement the GNN model by replacing RED with the GNN's output to set ECN marks and turning off packet dropping. Since each queue is mapped directly to a GNN node in this test case,

the RED queues were also modified to include hidden state information and other GNN metadata. Note that, in an actual data plane implementation, each forwarding device needs a data structure to keep track of the hidden states of each of its GNN nodes (one node per queue).

Every forwarding device sends information about the occupation from each of its queues to its neighbors at each time interval via a small packet. The exact size of each packet varies with each case, depending on the dimensions of the hidden state vectors and the number of nodes per switch. The packet contains an Ethernet header and a special GNN header with that information. The programmable devices can identify this header by a particular value in the Ethernet type field of the Ethernet header.

The time interval is another configurable parameter that needs to be tuned so that the switches can update the GNN classification more quickly and react to network traffic. However, we must also consider that more frequent messages incur higher communication overhead. In the presented experiments, each GNN message packet is 40 bytes long, and multiple time interval values were evaluated.

Since NS3 allows programmers to define any processing operations for the network devices to perform using C++ code, it is possible to implement the classification algorithms of the GNN directly in the simulator. This implementation mainly includes MLP forwarding, consisting of several loops for MLP layers, multiplications, divisions, and activation functions. These operations are not trivially implemented in actual programmable data planes and will be discussed further in Section 5.2.4.

For the classification, the GNN configuration is loaded from iGNNition's trained output weights at the start of the simulation, as opposed to the data plane implementation, in which the configuration is described in the code due to platform restrictions. The source code of the simulated experiments, with additional scripts to reproduce the results in the simulator, is publicly available.[5]

5.2.3.4 Results

Table 5.1: Results for the simulated experiments

	No ECN	**DCTCP**	**GNN (10ms)**	**GNN (1ms)**	**GNN (100us)**
Average Throughput (MBit/s)	170.65	313.03	152.42	220.24	312.73
Average GNN Accuracy			0.9087	0.9399	0.9645
Average GNN Precision			0.1259	0.5158	0.9573
Average Message Exchange BW Overhead			6.82×10^{-5}	6.82×10^{-4}	6.82×10^{-3}

[5]GitHub: <https://github.com/Ivanatorion/NOMS2025-GNN-Simulation>

Table 5.1 contains the main results of the simulations with the fat tree topology with multiple hosts attached to each switch. The GNN message exchange period varies between 10 ms, 1 ms, and 100 us. The table shows the average throughput for all test cases, the GNN accuracy and precision[6] for the experiments that uses GNNs, and the average bandwidth overhead of the message exchanges. Bandwidth overhead is given by the fraction of the 1.5Gbps link bandwidth used for GNN messages, according to Eq. (5.1), where NQ is the number of queues, NS is the number of switches, MSGSize is the size of messages, and MSGFreq is the frequency of messages. As the equation implies, the higher the link bandwidths, the less the communication overhead. Results show that with sufficiently high message frequency, one can achieve precision of more than 95% with relatively low overhead.

$$\frac{(\mathrm{NQ}/\mathrm{NS}) \times \mathrm{MSGSize} \times \mathrm{MSGFreq}}{\mathrm{LinkBandwidth}} \tag{5.1}$$

Table 5.2: Results for the experiment considering random topologies

	No ECN	**DCTCP**	**GNN (10ms)**	**GNN (1ms)**	**GNN (100us)**
Average Throughput (Compared to DCTCP)	93.54%	100%	99.63%	99.63%	99.63%

Table 5.2 shows the results for an additional test case with multiple (100) randomly generated network graphs. This brief test case was performed to show that the results can be generalized to different graph topologies. The table shows the average percentage of total throughput achieved by the traditional DCTCP algorithm for each ECN marking method. The results show that, in terms of throughput, the GNN performs almost as well as the traditional DCTPC algorithm, and both perform better than traditional TCP connections (without ECN).

5.2.4 P4 Framework

After validating the proposal in the simulator, the next goal was to prove the feasibility of the algorithm with actual data plane forwarding devices. In such an environment, other considerations not present in the simulated environment must be made, including network delays, packet losses, and, most importantly, the restrictions to the algorithm imposed by the PDP language (P4).

[6]For this case, **precision** is a better metric than **accuracy** since the vast majority ($\approx 92\%$) of the instances should be labeled as negatives due to the imbalanced nature of ECN markings.

To address these modifications, a framework for automatically generating P4 code that implements the algorithm, given a pre-trained GNN description, was developed. The framework source code and scripts for reproducing the experiments described in this section are also publicly available.[7]

The framework facilitates the user's job, as they only need to define the NNs that implement the message, aggregation, update, and readout functions. Still, as of now, the user must perform the training prior to the deployment in the programmable data plane, providing the weights of the NN's along with their architectural description.

After the generation of the P4 code, the user needs to insert the sources and sinks of data in the code. In the ECN example, those would be the queue occupancy and a boolean variable to decide the marking of packets, respectively. This way, we can replace traditional decision-making algorithms with GNN classification that runs entirely and distributedly in the PDP. The user may alternatively implement the entire generated code as a virtual switch and share the sources and sinks with another virtual switch that implements the main application logic.

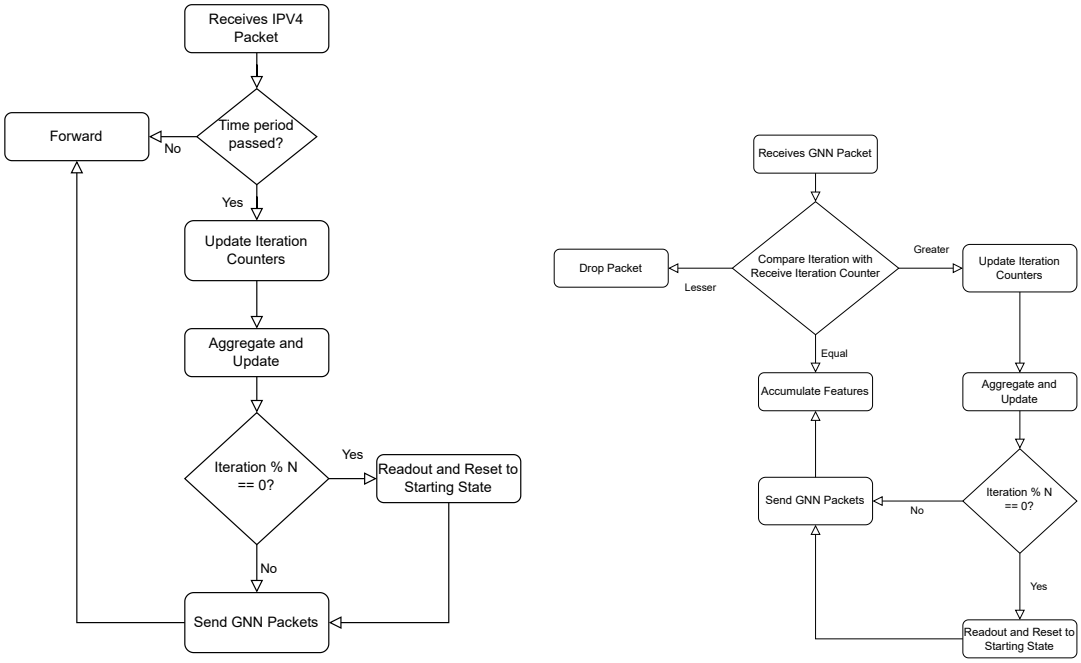
As mentioned throughout this thesis, various targets support the P4 language, but different architectures impose different constraints on the language. As such, the code generated by the framework is target-dependent. For example, the set of possible operations on registers in the TNA differs from the set of operations of the v1model architecture. Currently, the framework supports code generation for the bmv2 target. Support for Tofino switches is possible with tweaks to the framework's implementation, but is out of the scope of this thesis. For the hardware evaluation, the Tofino programs were manually created. The code for both architectures differs in several ways, mainly because of how they implement registers and limit the number of memory accesses per processed packet.

Implementing generic algorithms, such as MLP processing, in a PDP is more complex than in a general-purpose programming language. Those were the main challenges encountered and the insights for the solutions:

- **Lack of floating point support:** Fixed-point arithmetic is employed for the multiplication operations. Divisions when calculating averages of N neighbors with the multiplication by $1/N$. The number of bits employed for each part of the numbers (integer and fraction can impact the results of the classification. During testing, 8 bits for the integer part and 24 for the fraction were a good compromise, given the maximum 32-bit multiplication supported by the Tofino hardware.

[7] GitHub: <https://github.com/ComputerNetworks-UFRGS/NOMS2025-GNN-Dataplane>

Figure 5.4: Flowchart for the data plane GNN algorithm.



Source: The Author.

- **Creating packets:** Traffic packets are cloned[8] to perform GNN message exchange. After cloning the packets, the L3 protocol (IPv4, in the tests) is stripped, and the GNN protocol header is inserted in its place, containing the necessary information for each message. It is not necessary for each switch in the network to receive ordinary traffic for packet creation, as they can perform the same operation when receiving GNN packets from their neighbors. This way, a single packet received by any switch in the network is enough to trigger an iteration.

- **Loops:** Each loop is unrolled as necessary for the multiple layers of MLP processing. Since the GNN is trained offline, it is possible to hardcode the weights directly in the data plane program and sequentially apply each multiplication, aggregation, and activation function in the MLP.

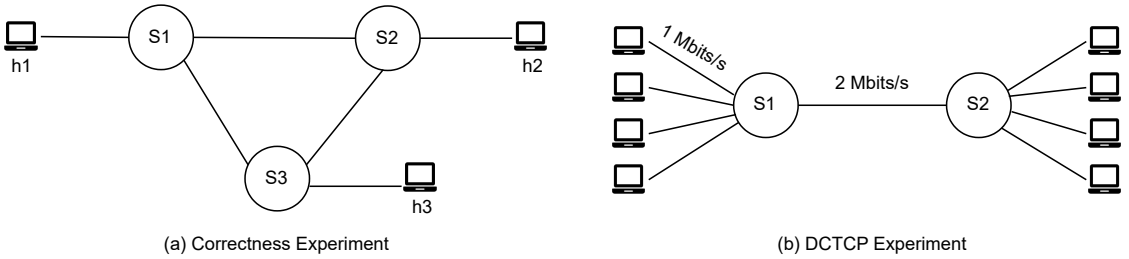
The modified algorithm that is implemented by the P4 framework is illustrated in Fig. 5.4. The iterations are triggered when a switch in the network receives a packet associated with network traffic. In the figure, the network processes IPv4 packets. The left side of the flow-chart depicts this initial step, which will update internal variables and send GNN packets with the feature vectors when the period passes. The right side of the flow-chart depicts the process that occurs when a switch receives such GNN packets, which

[8]Other target-dependent solutions may be available (such as Tofino's packet generator).

depends on the iteration counter for the current GNN iteration. The counter is important so that data from different iterations are not mixed, which may alter the classification result. Suppose the received packet has a counter less than the current iteration counter of the node. In this case, it means that the packet belongs to a previous iteration and has experienced a delay due to events in the network, so it should be discarded. Packets with an equal iteration counter are accumulated in the feature vectors. Packets with a greater counter indicate that a new iteration has begun, so the previous iteration is finalized (by applying the aggregate and update functions) before starting the next one with the received feature vector. When the number of iterations reaches N (defined by the user), the internal feature vector is reset, and the readout variables are updated for use by the application.

The correctness of the algorithm was evaluated with software switches running on mininet. Additionally, it was evaluated that a hardware-based Tofino switch can perform the same processing as the software switches.

Figure 5.5: Topology of the experiments with the GNN P4 framework.



Source: The Author.

Fig. 5.5 depicts the topologies for the correctness experiment (a) and the topology for the DCTCP experiment (b). For the correctness experiment (a), the network comprises three L3 switches in a triangle topology, processing IP packets between hosts connected to each switch. The network is kept busy with iperf flows between each pair of hosts during the experiment, which alternates in throughput to generate periods of congestion. The trained GNN classification model using the iGNNition framework was employed. It was verified that the programmable switches were able to correctly perform the operations of all NNs defined by the model (message, update, and readout) and mark packets accordingly. For the DCTCP experiment (b), the topology includes two L3 switches with multiple hosts in their subnets. The switches are connected by a link, the traffic bottleneck between hosts in different subnets. The experiment consists of 4 simultaneous iperf connections between pairs of hosts in each subnet, passing through the bottleneck link. The total duration of the experiment is 1 minute, and flows start with a delay of 10 seconds

each. The topology is similar to the one used by DCTPC's authors to evaluate the protocol (ALIZADEH et al., 2010). Table 5.3 contains the results regarding the aggregated throughput of the iperf connections and the fairness of the traffic distribution based on the Jain index. The results are an average of 30 repetitions. The GNN results are similar to the traditional DCTCP algorithm and similar to the results achieved in the simulated environment. Both DCTPC and GNN algorithms achieve higher throughput and fairness than Cubic TCP, which does not rely on ECN.

Table 5.3: Total throughput for the TCP congestion control schemes

	Total Throughput (Mbits/s)	**Jain Fairness**
Cubic	1.929	0.83
DCTCP	2.161	0.94
GNN	2.144	0.95

5.2.5 Advantages of Modular Programming

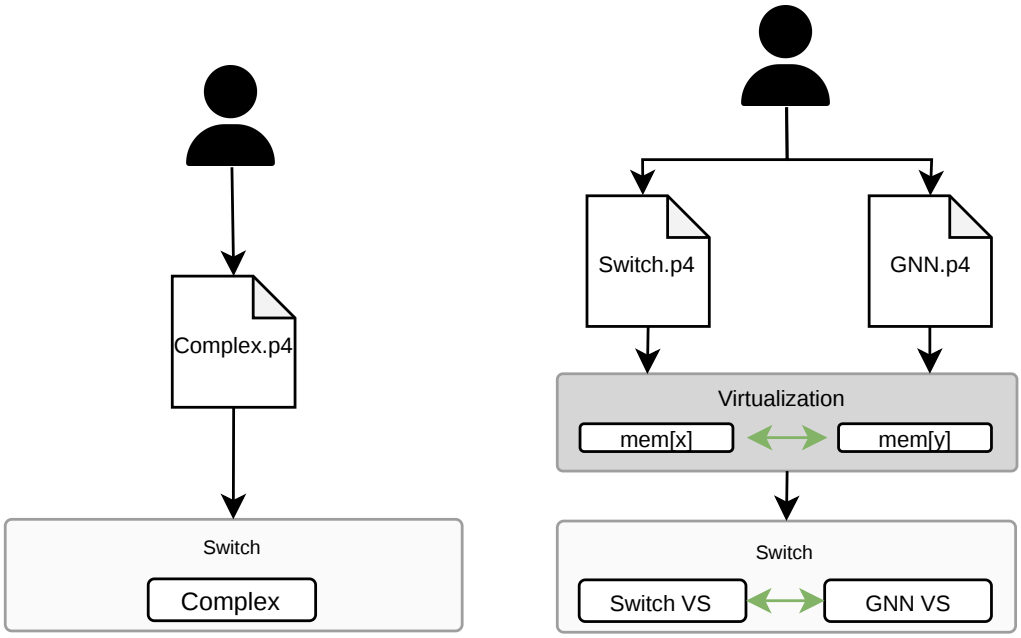
As mentioned in the previous section, after generating the code with the GNN framework, the programmer must adapt the code to insert the sources (e.g. queue occupation) and sinks (e.g. flag for marking ECN) of data. Due to the monolithic nature of P4 programs, this may lead to a tightly coupled implementation, which may prove difficult to maintain.

Modular data plane programming, enabled by the virtualization solution, improves on this aspect by allowing the decoupling of the main application logic and the GNN inference engine. Figure 5.6 compares the traditional and modular approaches. A single complex P4 code is deployed in the switch in the traditional approach (a). On the other hand, explicitly sharing internal memory on the host (which can be table entries or registers) creates an interface between virtual switches at runtime, enabling the separation of the program into multiple self-contained code files (b).

5.2.6 Conclusion

This work takes the first steps toward leveraging PDP resources to assist in distributed ML tasks. GNNs are a popular ML model, and their applications in the area of computer networks grow by the day. By performing GNN classification tasks directly in PDPs, we free up control plane resources in terms of processing time and eliminate

Figure 5.6: Flowchart for the data plane GNN algorithm.



(a) Traditional. (b) Modular.

Source: The Author.

the communication overhead between those planes, particularly as network speeds grow beyond 100G per link. The proposed PDP virtualization architecture, in particular the resource sharing features, facilitates the integration of the GNN classification engine to multiple application areas.

One possible continuation of this work involves the possibility of ML training directly in PDPs, which is much more challenging than classification. One promising approach involves using unsupervised ML techniques, which have been successful in some applications running on SmartNICs (CANNAROZZO et al., 2024).

5.3 Final Considerations

This Chapter presented mechanisms of the proposed PDP virtualization architecture that enable explicit resource sharing among tenants. This is an optional feature achieved by an extension to the already established P4 language, which supports annotations as a mean to extend the language specification.

This feature can be employed in two main contexts, (i) a single *entity* could control

multiple *tenants* of the system to deploy multiple virtual switches with shared resources or (ii) multiple *entities* could coordinate to share virtual switch resources. Mechanisms for coordination of different *entities* are not in the scope of this thesis.

It is also explained how modular data plane programming can be achieved with the help of this feature. A single *entity* can split their complex data plane logic into multiple virtual switches partially implementing the defined device behavior. The common interfaces of the modules can be implemented as shared resources with the help of the virtualization architecture.

The Chapter goes into detail of an example of modular data plane programming involving GNN classification running entirely in the PDP. The modular design allows for the separation of the GNN classification engine and the main application. This way, the virtualization architecture allows for the classification engine to be employed by multiple applications. Without modular data plane programming, the classification engine and the application would have to be tightly coupled into a single translation unit, hindering re-usability and adoption.

6 CONCLUSION AND FUTURE WORK

This chapter concludes this thesis with a summary of contributions. It also discusses possible directions for future work, including additional features could be incorporated to the virtualization architecture and PDP virtualization challenges in general.

6.1 Contributions

Programmable data plane virtualization technologies are still in the academic development phase, as no complete solution is currently available for broad utilization. Existing works focus on the concepts and strategies necessary for its implementation without worrying about practical concerns such as logical isolation among virtual devices, restricting the flexibility of the source programs, burdening the tenants with code rewriting, and integration with available control plane software.

The issue is aggravated when discussing multi-tenant solutions, as it requires even more involved mechanisms to guarantee logical isolation of the virtual devices. The first multi-tenant solutions heavily limited the freedom of the operators by limiting the devices to a subset of functions (CHUNG et al., 2021) or utilizing target-dependent hardware support, such as multiple pipelines that isolate traffic (STOYANOV; ZILBERMAN, 2020).

To better understand the limitations of current solutions, this thesis provided a background on the covered topics (Chapter 2). The background includes contextualization on the main topics related to this thesis, including SDNs, PDPs, In-network Computing, the P4 language, PDP virtualization, among others. It also includes a summary of some of the recent works on multi-tenant PDP virtualization.

In this thesis, a complete multi-tenant and target-independent PDP virtualization architecture was presented in order to address the identified limitations. The architecture design allows for the advancement of virtualization towards industry adoption by virtue of its design goals:

- **Support an arbitrary number of tenants:** The architecture can scale to accommodate as many tenants as the underlying hardware supports. The architecture provides a native slicing scheme based on the target ports, which does not require external control mechanisms. Additionally, the architecture supports tag-based slicing with the help of external mechanisms.

- **Unrestrained P4 language features:** The architecture does not restrict the P4 language to a subset of features more than the restrictions implied by the host target's hardware itself. Therefore, tenants may deploy their existing PDP programs as virtual switches just as they would deploy in a real target. This eliminates the need for adapting source code to comply with virtualization solution requirements.

- **Runtime deployment of virtual switches:** This thesis presents a novel approach to runtime deployment of virtual switches in composition-based virtualization. The goal is to provide each tenant with a logical view that (i) their program was changed, if they requested an update or (ii) nothing has changed, if they did not requested an update. This is achieved with the help of additional hardware resources which can be amortized by the number of targets in an operator's infrastructure.

- **Seamless control plane integration:** The runtime module, overlooked in earlier works, allows for the same control plane software that manages "real" switches to manage the virtual ones using standard CDPIs such as P4Runtime or even proprietary ones through extensions. This also eliminates the need for the tenants to adapt control plane software to comply with the virtualization architecture requirements.

- **Modular design that allows for use-case tailoring**: The modules of the architecture can be independently customized to each test case. This allows for an efficient implementation of solutions such as the multi-tenant topology emulation scheme presented in Chapter 4.

- **Public reference implementation:** A virtualization solution must offer a readly available implementation to foster reproducibility of scientific results and encourage adoption. The reference implementation developed along with this thesis is available online[1].

The main contributions of this thesis are: (i) the presentation of a ready-to-deploy PDP virtualization architecture with a publicly available reference implementation (Chapter 3), (ii) a novel runtime deployment strategy for virtual switches in a composition-based PDP virtualization architecture (Section 3.4), (iii) two concrete use-cases for the presented architecture, on the topics of PDP prototyping environtment (Chapter 4) and in-network GNN inference leveraging modular data plane programming (Chapter 5).

[1] <https://github.com/Ivanatorion/VirtualizationArchThesis>

6.2 Future Work

Some of the possible directions for further research include:

- **Table merging resource usage evaluation:** When discussing the strategy used for selecting the possible tables to be merged (Section 3.2.4), it was noted that an optimal solution requires solving an MWIS instance. Previous work in the area proposed the use of heuristics to solve those instances (ZHENG; BENSON; HU, 2018), but there is a lack of analysis for the need for such heuristics for the virtual PDP use case. A more thorough investigation of complex production P4 code merging is planned to verify the feasibility of finding optimal solutions. This study will also determine the appropriate weight that each merging receives by ascertaining the impact of the merging in terms of hardware resource usage.

- **Source code protection:** One feature that may be interesting to some tenants is the protection of the virtual switches' source code. The tenants might not want to disclose their codes to the system administrator of a solution that implements the architecture. P4VBox (Saquetti et al., 2020), previously discussed in Section 2.8.4, brought the discussion of protecting the intellectual property of the tenants of a virtualization solution. Essentially, P4VBox can achieve this because of its composition method (FPGA partial configuration), which only requires the binary outputs of the synthesis process. As of right now, the code composition module needs full access to the P4 files of each virtual device that is to be implemented. This additional functionality could be implemented in the Security Module. One possible measure to mitigate this problem is to use code obfuscation (COLLBERG; THOMBORSON; LOW, 1997), which is a feature implemented by some P4 SDEs, such as Intel's. Other recent studies also show that code obfuscation can improve the data plane resilience to network attacks even on non-multi-tenant or non-virtual scenarios (BLACK; SCOTT-HAYWARD, 2023).

- **Consider multiple available hosts:** The virtualization architecture presented assumes a single host for the multiple virtual switches. In Chapter 4, it was mentioned that when considering multiple hosts available, it may be possible to optimize virtual switch placement. This optimization could be applied independently of the use case by formalizing it in the architecture's definition. Future work may explore placement strategies of virtual switches given multiple available hosts.

- **Evaluation with other targets:** Since target independency is one of the main goals of the virtualization architecture, evaluation with other PDP targets, such as SmartNICs, should be considered. During the development of this thesis, there has been informal experimentation of code composition for the DOCA Pipeline Language of NVIDIA's Bluefield3 devices (NVIDIA, 2025). A former evaluation could lead to a usable implementation of the architecture that supports those devices.

REFERENCES

ABBATE, J. E. **From ARPANET to Internet: A history of ARPA-sponsored computer networks, 1966-1988**. [S.l.]: University of Pennsylvania, 1994.

ALDALBAHI, A. et al. Visible light communication module: An open source extension to the ns3 network simulator with real system validation. **IEEE Access**, v. 5, p. 22144–22158, 2017.

ALIZADEH, M. et al. Data center tcp (dctcp). **SIGCOMM Comput. Commun. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 40, n. 4, p. 63–74, aug 2010. ISSN 0146-4833. Available from Internet: <https://doi.org/10.1145/1851275.1851192>.

ANDERSON, T. et al. Overcoming the internet impasse through virtualization. **Computer**, v. 38, n. 4, p. 34–41, 2005.

ANEROUSIS, N. et al. The origin and evolution of open programmable networks and sdn. **IEEE Communications Surveys & Tutorials**, v. 23, n. 3, p. 1956–1971, 2021.

BAI, J. et al. Ns4: Enabling programmable data plane simulation. In: **Proceedings of the Symposium on SDN Research**. New York, NY, USA: Association for Computing Machinery, 2018. (SOSR '18). ISBN 9781450356640. Available from Internet: <https://doi.org/10.1145/3185467.3185470>.

BERNáRDEZ, G. et al. Is Machine Learning Ready for Traffic Engineering Optimization? In: . [S.l.: s.n.], 2021.

BIANCHI, G. et al. Openstate: Programming platform-independent stateful openflow applications inside the switch. **ACM SIGCOMM Computer Communication Review**, v. 44, p. 44–51, 04 2014.

BLACK, C.; SCOTT-HAYWARD, S. Defeating data plane attacks with program obfuscation. **IEEE Transactions on Dependable and Secure Computing**, p. 1–13, 2023.

BOMZE, I. M. et al. The maximum clique problem. In: ____. **Handbook of Combinatorial Optimization: Supplement Volume A**. Boston, MA: Springer US, 1999. p. 1–74. Available from Internet: <https://doi.org/10.1007/978-1-4757-3023-4_1>.

BOSSHART, P. et al. P4: Programming protocol-independent packet processors. **SIGCOMM Comput. Commun. Rev.**, ACM, New York, NY, USA, v. 44, n. 3, p. 87–95, jul. 2014. ISSN 0146-4833.

BOWMAN-AMUAH, M. K. **Load balancer in environment services patterns**. [S.l.]: Google Patents, 2003. US Patent 6,578,068.

BROADCOM. **NPL Language**. 2023. Retrieved November 23, 2023 from <https://nplang.org/>.

BUENO, G. et al. Managing virtual programmable switches: Principles, requirements, and design directions. **IEEE Communications Magazine**, v. 60, n. 2, p. 53–59, 2022.

BUZURA, S. et al. A hybrid software and hardware sdn simulation testbed. **Sensors**, v. 23, n. 1, 2023. ISSN 1424-8220. Available from Internet: <https://www.mdpi.com/1424-8220/23/1/490>.

CAMPANILE, L. et al. Computer network simulation with ns-3: A systematic literature review. **Electronics**, v. 9, n. 2, 2020. ISSN 2079-9292. Available from Internet: <https://www.mdpi.com/2079-9292/9/2/272>.

CANNAROZZO, L. et al. Spinner: Enabling in-network flow clustering entirely in a programmable data plane. In: **NOMS 2024-2024 IEEE Network Operations and Management Symposium**. [S.l.: s.n.], 2024. p. 1–9.

CARNEIRO, G.; FORTUNA, P.; RICARDO, M. Flowmonitor: a network monitoring framework for the network simulator 3 (ns-3). In: **Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools**. [S.l.: s.n.], 2009. p. 1–10.

CHUNG, B. et al. P4mt: Designing and evaluating multi-tenant services for p4 switches. In: **2021 22nd Asia-Pacific Network Operations and Management Symposium (APNOMS)**. [S.l.: s.n.], 2021. p. 267–272.

COLLBERG, C.; THOMBORSON, C.; LOW, D. A taxonomy of obfuscating transformations. **http://www.cs.auckland.ac.nz/staff-cgi-bin/mjd/csTRcgi.pl?serial**, 01 1997.

DACIER, M. C. et al. Security challenges and opportunities of software-defined networking. **IEEE Security & Privacy**, IEEE, v. 15, n. 2, p. 96–100, 2017.

DANG, H. T. P4dns: In-network dns. **ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)**, IEEE, 2019.

EGEVANG, K. B.; FRANCIS, P. **The IP Network Address Translator (NAT)**. RFC Editor, 1994. RFC 1631. (Request for Comments, 1631). Available from Internet: <https://www.rfc-editor.org/info/rfc1631>.

ERNST, M.; BADROS, G.; NOTKIN, D. An empirical analysis of c preprocessor use. **IEEE Transactions on Software Engineering**, v. 28, n. 12, p. 1146–1170, 2002.

FEAMSTER, N.; REXFORD, J. Why (and how) networks should run themselves. **CoRR**, abs/1710.11583, 2017. Available from Internet: <http://arxiv.org/abs/1710.11583>.

FERRIOL-GALMéS, M. et al. Routenet-erlang: A graph neural network for network performance evaluation. In: **IEEE INFOCOM 2022**. [S.l.: s.n.], 2022. p. 2018–2027.

GAO, J. et al. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In: **Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication**. [S.l.: s.n.], 2020. p. 435–450.

GILMER, J. et al. **Neural Message Passing for Quantum Chemistry**. arXiv, 2017. Available from Internet: <https://arxiv.org/abs/1704.01212>.

GNU Project. **Bison**. [S.l.], 2023. Available from Internet: <https://www.gnu.org/software/bison/>.

GNU Project. **The C Preprocessor**. [S.l.], 2023. Available from Internet: <https://gcc.gnu.org/onlinedocs/cpp/>.

GNU Project. **Flex**. [S.l.], 2023. Available from Internet: <https://www.gnu.org/software/flex/>.

GOOGLE. **gRPC**. 2015. Retrieved November 30, 2023 from <https://grpc.io/>.

GOOGLE. **Protocol Buffers Specification**. 2026. Retrieved January 22, 2026 from <https://protobuf.dev/>.

HANCOCK, D.; MERWE, J. V. D. Hyper4: Using p4 to virtualize the programmable data plane. In: ACM. **CoNEXT'16**. [S.l.], 2016. p. 35–49.

HANG, Z. et al. Programming protocol-independent packet processors high-level programming (p4hlp): Towards unified high-level programming for a commodity programmable switch. 2019.

IBANEZ, S. et al. The p4-netfpga workflow for line-rate packet processing. In: **ACM/SIGDA Int'l Symposium on Field-Programmable Gate Arrays**. New York, NY, USA: ACM, 2019. (FPGA '19), p. 1–9. ISBN 978-1-4503-6137-8. Available from Internet: <http://doi.acm.org/10.1145/3289602.3293924>.

Intel. **Tofino Native Architecture**. [S.l.], 2021. Available from Internet: <https://github.com/barefootnetworks/Open-Tofino>.

JIN, X. et al. Netcache: Balancing key-value stores with fast in-network caching. In: **Proceedings of the 26th Symposium on Operating Systems Principles**. [S.l.: s.n.], 2017. p. 121–136.

KARP, R. M. Reducibility among combinatorial problems. In: ____. **Complexity of Computer Computations**. Boston, MA: Springer US, 1972. p. 85–103. ISBN 978-1-4684-2001-2. Available from Internet: <https://doi.org/10.1007/978-1-4684-2001-2_9>.

KHASHAB, S.; RASHELBACH, A.; SILBERSTEIN, M. Multitenant In-Network acceleration with SwitchVM. In: **21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)**. Santa Clara, CA: USENIX Association, 2024. p. 691–708. ISBN 978-1-939133-39-7. Available from Internet: <https://www.usenix.org/conference/nsdi24/presentation/khashab>.

KIANPISHEH, S.; TALEB, T. A survey on in-network computing: Programmable data plane and technology specific applications. **IEEE Communications Surveys & Tutorials**, v. 25, n. 1, p. 701–761, 2023.

KIM, C. et al. In-band network telemetry via programmable dataplanes. In: **ACM SIGCOMM**. [S.l.: s.n.], 2015. v. 15, p. 1–2.

KUROSE, J.; ROSS, K. **Computer networks: A top down approach featuring the internet**. [S.l.]: Pearson Addison Wesley, 2010.

LAMB, I. P. et al. Multi-tenant programmable switch virtualization leveraging explicit resource sharing. In: **2024 20th International Conference on Network and Service Management (CNSM)**. [S.l.: s.n.], 2024. p. 1–9.

LAMB, I. P. et al. Distributed graph neural networks in programmable data planes. In: **NOMS 2025-2025 IEEE Network Operations and Management Symposium**. [S.l.: s.n.], 2025. p. 01–09.

LAMB, I. P. et al. Protecting virtual programmable switches from cross-app poisoning (cap) attacks. In: **NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium**. [S.l.: s.n.], 2022. p. 1–9.

LANTZ, B.; HELLER, B.; MCKEOWN, N. A network in a laptop: Rapid prototyping for software-defined networks. In: **Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks**. New York, NY, USA: Association for Computing Machinery, 2010. (Hotnets-IX). ISBN 9781450304092. Available from Internet: <https://doi.org/10.1145/1868447.1868466>.

LI, Y.; CHEN, M. Software-defined network function virtualization: A survey. **IEEE Access**, v. 3, p. 2542–2553, 2015.

LIAO, H.-J. et al. Intrusion detection system: A comprehensive review. **Journal of Network and Computer Applications**, Elsevier, v. 36, n. 1, p. 16–24, 2013.

LUIZELLI, M. C. et al. In-network neural networks: Challenges and opportunities for innovation. **IEEE Network**, v. 35, n. 6, p. 68–74, 2021.

LUO, S.; YU, H.; VANBEVER, L. Swing state: Consistent updates for stateful and programmable data planes. In: **Proceedings of the Symposium on SDN Research**. New York, NY, USA: Association for Computing Machinery, 2017. (SOSR '17), p. 115–121. ISBN 9781450349475. Available from Internet: <https://doi.org/10.1145/3050220.3050233>.

MCKEOWN, N. et al. Openflow: Enabling innovation in campus networks. **ACM SIGCOMM Computer Communication Review**, v. 38, p. 67–79, 2008.

MOY, J. **OSPF Version 2**. RFC Editor, 1998. RFC 2328. (Request for Comments, 2328). Available from Internet: <https://www.rfc-editor.org/info/rfc2328>.

NVIDIA. **P4 Language Support in DPL**. 2025. Retrieved January 27, 2026 from <https://docs.nvidia.com/doca/sdk/p4-language-support-in-dpl>.

OKTIAN, Y. E. et al. Distributed sdn controller system: A survey on design choice. **computer networks**, Elsevier, v. 121, p. 100–111, 2017.

OLIVEIRA, R. L. S. et al. Using mininet for emulation and prototyping software-defined networks. In: **2014 IEEE Colombian Conference on Communications and Computing (COLCOM)**. [S.l.: s.n.], 2014. p. 1–6.

OMNET++. **OMNeT++**. 2023. Retrieved November 28, 2023 from <https://omnetpp.org/>.

ONOS. **Fabric Switch**. 2021. Retrieved January 14, 2026 from <https://github.com/opennetworkinglab/onos/blob/master/pipelines/fabric/impl/src/main/resources/fabric.p4>.

ONOS. **Open Network Operating System. Apps and Use Cases**. 2021. Retrieved January 14, 2026 from <https://docs.onosproject.org/>.

OPEN-NETWORK-FOUNDATION. **SDN Architecture Overview**. 2013. Retrieved September 3, 2021 from <https://opennetworking.org/wp-content/uploads/2013/02/SDN-architecture-overview-1.0.pdf>.

OPEN-NETWORK-FOUNDATION. **OpenFlow Switch Specification**. 2015. Retrieved September 2, 2021 from <https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf>.

ORDABAYEVA, G. K. et al. A systematic review of transition from ipv4 to ipv6. In: **Proceedings of the 6th International Conference on Engineering & MIS 2020**. New York, NY, USA: Association for Computing Machinery, 2020. (ICEMIS'20). ISBN 9781450377362. Available from Internet: <https://doi.org/10.1145/3410352.3410735>.

P4LANG-CONSORTIUM et al. **P4-bmv2. Website**. 2019. Available from Internet: <https://github.com/p4lang/behavioral-model>.

PARIZOTTO, R. et al. Prime: Programming in-network modular extensions. In: **NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium**. [S.l.: s.n.], 2020. p. 1–9.

PUJOL-PERICH, D. et al. Ignnition: Bridging the gap between graph neural networks and networking systems. **IEEE Network**, IEEE, v. 35, n. 6, p. 171–177, 2021.

PUJOL-PERICH, D. et al. **Unveiling the potential of Graph Neural Networks for robust Intrusion Detection**. arXiv, 2021. Available from Internet: <https://arxiv.org/abs/2107.14756>.

QUINN, P.; NADEAU, T. **Problem Statement for Service Function Chaining**. RFC Editor, 2015. RFC 7498. (Request for Comments, 7498). Available from Internet: <https://www.rfc-editor.org/info/rfc7498>.

RILEY, G. F.; HENDERSON, T. R. The ns-3 network simulator. In: ____. **Modeling and Tools for Network Simulation**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 15–34. ISBN 978-3-642-12331-3. Available from Internet: <https://doi.org/10.1007/978-3-642-12331-3_2>.

RODRIGUEZ, F. et al. Network emulation with p7: A p4 programmable patch panel on tofino-based hardware. In: **Anais Estendidos do XLI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos**. Porto Alegre, RS, Brasil: SBC, 2023. p. 40–47. ISSN 2177-9384. Available from Internet: <https://sol.sbc.org.br/index.php/sbrc_estendido/article/view/24631>.

RUSEK, K. et al. RouteNet: Leveraging graph neural networks for network modeling and optimization in SDN. **IEEE Journal on Selected Areas in Communications**, Institute of Electrical and Electronics Engineers (IEEE), v. 38, n. 10, p. 2260–2270, oct 2020. Available from Internet: <https://doi.org/10.1109\%2Fjsac.2020.3000405>.

Saquetti, M. et al. P4vbox: Enabling p4-based switch virtualization. **IEEE Communications Letters**, v. 24, n. 1, p. 146–149, Jan 2020. ISSN 2373-7891.

SCARSELLI, F. et al. The graph neural network model. **IEEE Transactions on Neural Networks**, v. 20, n. 1, p. 61–80, 2009.

SCAZZARIELLO, M. et al. A High-Speed stateful packet processing approach for tbps programmable switches. In: **20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)**. Boston, MA: USENIX Association, 2023. p. 1237–1255. ISBN 978-1-939133-33-5. Available from Internet: <https://www.usenix.org/conference/nsdi23/presentation/scazzariello>.

SIVARAMAN, A. et al. **Packet Transactions: High-level Programming for Line-Rate Switches**. 2016.

SONG, H. 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**. New York, NY, USA: Association for Computing Machinery, 2013. (HotSDN '13), p. 127–132. ISBN 9781450321785. Available from Internet: <https://doi.org/10.1145/2491185.2491190>.

SONI, H. et al. Composing dataplane programs with microp4. In: **Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication**. New York, NY, USA: Association for Computing Machinery, 2020. (SIGCOMM '20), p. 329–343. ISBN 9781450379557. Available from Internet: <https://doi.org/10.1145/3387514.3405872>.

STOYANOV, R.; ZILBERMAN, N. Mtpsa: Multi-tenant programmable switches. In: **Proceedings of the 3rd P4 Workshop in Europe**. New York, NY, USA: Association for Computing Machinery, 2020. (EuroP4'20), p. 43–48. ISBN 9781450381819. Available from Internet: <https://doi.org/10.1145/3426744.3431329>.

SZYMANSKI, B. et al. Genesis: A system for large-scale parallel network simulation. In: . [S.l.: s.n.], 2002. p. 89–96.

TAN, W. et al. From the service-oriented architecture to the web api economy. **IEEE Internet Computing**, v. 20, n. 4, p. 64–68, 2016.

TANAEMBAUM, A. S.; BOS, H. **Modern Operating Systems**. 4th. ed. [S.l.]: Prentice Hall, 2014.

TENNENHOUSE, D. et al. A survey of active network research. **IEEE Communications Magazine**, v. 35, n. 1, p. 80–86, 1997.

The P4.org API Working Group. **P4 16 Specification**. [S.l.], 2017. Available from Internet: <https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html>.

The P4.org API Working Group. **P4Runtime Specification**. [S.l.], 2019. Available from Internet: <https://p4.org/p4runtime/spec/v1.0.0/P4Runtime-Spec.pdf>.

TOKUSASHI, Y.; MATSUTANI, H.; ZILBERMAN, N. Lake: The power of in-network computing. **International Conference on ReConFigurable Computing and FPGAs (ReConFig)**, IEEE, 2018.

UJCICH, B. E. et al. Cross-app poisoning in software-defined networking. In: **2018 ACM SIGSAC Conference on Computer and Communications Security**. New York, NY, USA: ACM, 2018. (CCS '18), p. 648–663. ISBN 9781450356930.

UNICAMP, C. N. L. at. **OFSwitch13**. 2023. Retrieved November 29, 2023 from <http://www.lrc.ic.unicamp.br/ofswitch13/>.

WHITE, J. E. A high-level framework for network-based resource sharing. In: **Proceedings of the June 7-10, 1976, National Computer Conference and Exposition**. New York, NY, USA: Association for Computing Machinery, 1976. (AFIPS '76), p. 561–570. ISBN 9781450379175. Available from Internet: <https://doi.org/10.1145/1499799.1499878>.

WINSTEIN, K.; BALAKRISHNAN, H. TCP Ex Machina: Computer-generated Congestion Control. In: . [S.l.: s.n.], 2013.

WOODRUFF, J. P4xos: Consensus as a network service. **IEEE/ACM Transactions on Networking**, IEEE, v. 28, 2020.

WU, Z. et al. A comprehensive survey on graph neural networks. **IEEE Transactions on Neural Networks and Learning Systems**, v. 32, n. 1, p. 4–24, 2021.

ZHANG, C. et al. Hypervdp: High-performance virtualization of the programmable data plane. **IEEE Journal on Selected Areas in Communications**, v. 37, n. 3, p. 556–569, March 2019. ISSN 0733-8716.

ZHANG, J. **Tencent Implements Gateway Functionality in P4 Switching Appliance**. 2022. Retrieved January 28, 2026 from <https://p4.org/p4-programmable-switch-appliance-in-tencent-data-center-network/>.

ZHANG, X. et al. Scion: Scalability, control, and isolation on next-generation networks. In: **2011 IEEE Symposium on Security and Privacy**. [S.l.: s.n.], 2011. p. 212–227.

ZHENG, P.; BENSON, T.; HU, C. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In: **CoNEXT '18**. New York, NY, USA: ACM, 2018. p. 98–111. ISBN 978-1-4503-6080-7.

APPENDIX A — RESUMO EXPANDIDO EM PORTUGUÊS

Este apêndice apresenta de forma resumida esta tese de doutorado, intitulada "Uma arquitetura de virtualização de planos de dados programáveis multi-inquilino e indepêndente de hospedeiro".

A.1 Introdução

Os avanços proporcionados pelo paradigma de Redes Definidas por Software (SDN do Inglês Software-defined Networking) e Planos de Dados Programáveis (PDPs do Inglês *Programmable Data Planes*) fornecem aos administradores de redes e pesquisadores uma grande flexibilidade sobre como um dispositivo do plano de dados (*switch*) interpreta e processa o tráfego a nível de pacotes. Esta flexibilidade tornou possível a implementação de diversas funcionalidades e serviços diretamente no plano de dados, aproveitando-se da vantagem de processamento de pacotes na velocidade do enlace, que atualmente atinge a ordem de Tbps. As aplicações incluem *caching*, monitoração, armazenamento de chaves, consenso, entre outras. Ao delegar parte do processamento de dados aos dispositivos de encaminhamento, também é possível reduzir a carga de trabalho dos servidores tradicionais e dispositivos na ponta da rede, que são os principais responsáveis por tais tarefas no modelo tradicional. Mais importante ainda, o atraso da comunicação do plano de dados ao plano de controle (na ordem de milissegundos) é removido.

Nesta nova era das redes, a complexidade das operações realizadas pelos dispositivos de núcleo aumentou, visto que podem implementar quase qualquer algoritmo que o operador defina. Este cenário gera a necessidade de ambientes de prototipagem, teste e depuração de projetos inovadores de redes. Recentemente, a virtualização de PDPs emergiu como uma ferramenta capaz de fornecer tais ambientes. Esta tecnologia permite a divisão de *hardware*, de forma similar a máquinas virtuais em servidores tradicionais. Desta forma, operadores e pesquisadores podem adotar testes mais exaustivos de protocolos que dependam da interação de múltiplos dispositivos utilizando apenas um *switch* físico. A proposta torna-se ainda mais interessante ao se considerar os custos de aquisição e manutenção destes equipamentos, que chegam à ordem de dezenas de milhares de dólares.

A.2 Motivações

As soluções de virtualização existentes ainda não adquiriram maturação o suficiente para a adoção do mercado e da indústria. As soluções inicias focam nas tecnologias subjacentes que permitem a divisão de *hardware* (emulação e composição), sem se importar com aspectos práticos de implementação em sistemas multi-inquilinos (do Inglês, *multi-tenant*), como o isolamento lógico dos dispositivos virtuais. Soluções mais recentes, que almejam fornecer um sistema de virtualização multi-inquilino, impõem restrições severas ao comportamento que pode ser definido pelos usuários (indo contra o princípio de planos de dados programáveis) ou requerem a adoção de um novo modelo / linguagem de programação (exigindo grandes reescritas de código).

Além disto, a indepêndencia de hospedeiro (do Inglês, *target-independency*) é uma caracteristica fundamental pouco valorizada pelas soluções atuais. Ela torna as soluções fácilmente portáveis para múltiplas infraestruturas de redes e estimula a reproducibilidade científica através do compartilhamento dos artefatos de pesquisa (códigos fonte, por exemplo). Muitas das soluções de virtualização atuais, focadas em um *hardware* hospedeiro específico, não podem (ou não querem) publicar estes artefatos devido a acordos de não divulgação restritivos com fabricantes.

A.3 Contribuições

Esta tese apresenta uma arquitetura completa de virtualização de planos de dados programáveis juntamente a uma implementação de referência. Os princípios de projeto são estabelecidos a fim de abordar as limitações atuais citadas. A implementação de referência, além de incentivar a reprodutividade da pesquisa, permite o uso de virtualização em projetos maiores, ampliando o escopo da tecnologia. A implementação de referência da arquitetura foi validada através de testes experimentais e de casos de usos concretos envolvendo emulação de topologias em *hardware* e inferência de redes neurais no PDP.

A arquitetura proposta permite (i) a composição de programas distintos que definem o comportamento de um dispositivo de encaminhamento em um único programa, (ii) o mapeamento dos elementos do programa composto para os dispositivos virtuais e (iii) um esquema de controle de acesso para os *tenants* controlarem seus dispositivos de forma segura, utilizando o mesmo software de controle utilizado sem virtualização (i.e. não é necessária reescrita de código).

Os módulos fundamentais da arquitetura proposta são os de (i) composição de código e (ii) controle em tempo de execução (do Inglês, *runtime*). O módulo de composição de código é capaz de gerar um programa único que será implementado no dispositivo hospedeiro utilizando múltiplos códigos pertencentes a diferentes *tenants*, cada um descrevendo o comportamento de um *switch* virtual. O módulo de *runtime* fornece uma camada de abstração para que os *tenants* interajam de forma transparente com os *switchs* virtuais, utilizando o mesmo software de controle que utilizariam em um *switch* "real".

Destacam-se, portanto, as seguintes contribuições:

1. Uma arquitetura de virtualização leve independente de hospedeiro capaz de fornecer suporte a múltiplos *tenants* de forma transparente.

2. Uma implementação de referência da arquitetura proposta, testada em diferentes dispositivos hospedeiros, provendo reprodutibilidade à pesquisa desenvolvida.

3. Uma estratégia de modificação dos switches virtuais em tempo de execução em uma arquitetura de virtualização baseada em composição de código.

4. Uma solução para emulação de topologias em *hardware* utilizando a arquitetura de virtualização proposta.

5. Uma solução para inferência de redes neurais diretamente no plano de dados que utiliza a arquitetura de virtualização para alcançar programabilidade modular do plano de dados.

A.4 Resultados Experimentais

Os resultados obtidos dos experimentos são descritos em maiores detalhes ao decorrer desta tese. Eles demonstram a capacidade de gerenciar de forma transparente múltiplos *switches* virtuais pertencentes a diferentes *tenants*. Os testes foram realizados em diferentes dispositivos hospedeiros, tanto de *hardware* como *software*, para demonstrar a independencia de dispositivo da arquitetura proposta.

Os principais resultados dos experimentos são:

- **Isolamento entre dispositivos virtuais:** Os módulos de *runtime* e *security* da arquitetura proposta conseguem detectar requisições inválidas aos *switchs* virtuais e impedir a interferência no funcionamento dos dispositivos.

- **Desempenho:** O overhead em termos de latência e *throughput* são aceitáveis para diversas aplicações devido à estratégia de composição utilizada.

- **Uso de hardware:** Devido ao uso de virtualização leve, em que elementos de múltiplos programas são compartilhados no *hardware* físico, é possível obter ganhos sublineares de espaço em *hardware* conforme o número de tabelas e registradores dos *switches* virtuais aumenta.

A.5 Trabalhos Futuros

No decorrer da tese são descritos problemas ainda não atacados pela arquitetura prospota. Estes problemas poderão ser abordados em trabalhos futuros. Destacam-se:

- **Casos de uso adicionais:** Além dos casos de uso envolvendo emulação de topologias e inferência de redes neurais, mais soluções podem ser apresentadas para demonstrar o impacto do uso de virtualização em redes modernas. Uma possibilidade muito discutida é a de *switches* virtuais como serviço, similar a máquinas virtuais em *data centers* tradicionais. A modularidade da arquitetura facilita as adaptações para seu uso em diferentes ambientes.

- **Proteção do código-fonte:** Uma característica que pode ser interessante para alguns usuários é a proteção do código-fonte dos switches virtuais. Os usuários podem não querer divulgar seus códigos ao administrador do sistema de uma solução que implementa a arquitetura. Alguns trabalhos relacionados discutidos nesta tese trazem à tona a discussão sobre a proteção da propriedade intelectual dos usuários de uma solução de virtualização. Algumas soluções podem conseguir isso devido ao método de composição (e.g. configuração parcial de FPGAs), que requer apenas as saídas binárias do processo de compilação ou síntese. No momento, o módulo de composição de código da arquitetura proposta precisa de acesso total aos arquivos fonte de cada dispositivo virtual a ser implementado. Essa funcionalidade adicional poderia ser implementada no Módulo de Segurança. Uma medida possível para mitigar esse problema é usar ofuscação de código, um recurso implementado por alguns SDEs P4, como os da Intel (P4 Studio). Outros estudos recentes também mostram que a ofuscação de código pode melhorar a resiliência do plano de dados a ataques de rede, mesmo em cenários não multitenant ou não virtuais.

- **Considerar múltiplos hospedeiros disponíveis:** A arquitetura de virtualização apresentada pressupõe um único dispositivo hospedeiro para os vários switches virtuais. Ao considerar vários hosts disponíveis, pode ser possível otimizar o posicionamento do switch virtual. Isto pode ser formalizanda na definição da arquitetura. Trabalhos futuros podem explorar estratégias de posicionamento de switches virtuais, considerando vários hospedeiros disponíveis.

- **Avaliação com outros dispositivos:** Como a independência do dispositivo hospedeiro é um dos principais objetivos da arquitetura de virtualização, deve-se considerar a avaliação com outros dispositivos de PDPs, como SmartNICs. Durante o desenvolvimento desta tese, houve experimentação informal da composição de código com outros dispositivos, mas sem uma avaliação formal rigorosa. Tal avaliação poderia levar a uma implementação utilizável da arquitetura para suportar esses dispositivos.

APPENDIX B — PUBLICATIONS

The presented work resulted in the publication of the following papers:

- LAMB, I. P.; et al. Arquitetura de Virtualização Leve Multi-Tenant para Switches Programáveis. In: **Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC) 42**, 2024, Niterói, RJ. DOI: 10.5753/sbrc.2024.1235

- LAMB, I. P.; et al. Multi-Tenant Programmable Switch Virtualization Leveraging Explicit Resource Sharing. In: **20th International Conference on Network and Service Management (CNSM)**, 2024, Prague, Czech Republic. DOI: 10.23919/CNSM62983.2024.10814530

- LAMB, I. P.; et al. Distributed Graph Neural Networks in Programmable Data Planes. In: **NOMS 2025-2025 IEEE Network Operations and Management Symposium**, 2025, Honolulu, USA. DOI: 10.1109/NOMS57970.2025.11073693

Additionally, the presented work resulted in the publication of a short paper presented in a conference poster session:

- LAMB, I. P.; et al. Multi-Tenant Programmable Switch Virtualization Architecture. **IEEE Network Operations and Management Symposium (NOMS 2024)**., 2024. Seoul, Republic of Korea. DOI: 10.1109/NOMS59830.2024.10575917.

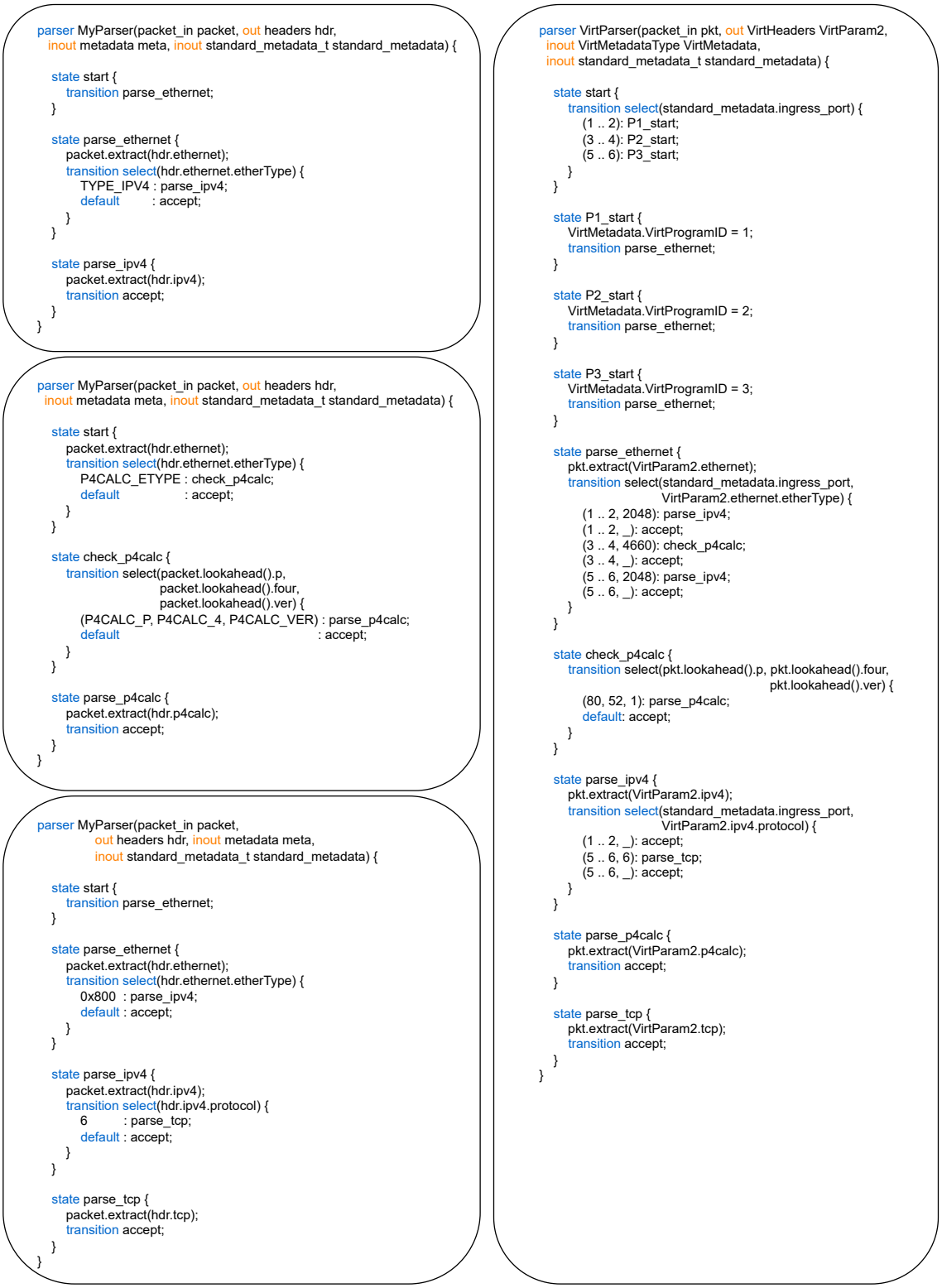
In the context of in-network computing through data plane programmability, the author of this thesis also contributed to the following papers:

- CANNAROZZO, L.; et al. Spinner: Enabling In-network Flow Clustering Entirely in a Programmable Data Plane., **IEEE Network Operations and Management Symposium (NOMS 2024)**., 2024. Seoul, Republic of Korea. DOI: 10.1109/NOMS59830.2024.10575413.

- SILVA, A. F.; et al. Posicionamento e Encadeamento em NFV: Como Lidar Quando o Plano de Dados é Programável e Multi-Tenant?, **Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC) 43**., 2025. Natal, RN. DOI: 10.5753/sbrc.2025.6254.

APPENDIX C — EXAMPLE OF THE MERGING OF PARSER BLOCKS

Figure C.1: Example of the merging of three ingress parsers.



```
parser MyParser(packet_in packet, out headers hdr,
  inout metadata meta, inout standard_metadata_t standard_metadata) {

    state start {
        transition parse_ethernet;
    }

    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_IPV4 : parse_ipv4;
            default   : accept;
        }
    }

    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition accept;
    }
}
```

```
parser MyParser(packet_in packet, out headers hdr,
  inout metadata meta, inout standard_metadata_t standard_metadata) {

    state start {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            P4CALC_ETYPE : check_p4calc;
            default      : accept;
        }
    }

    state check_p4calc {
        transition select(packet.lookahead().p,
                          packet.lookahead().four,
                          packet.lookahead().ver) {
            (P4CALC_P, P4CALC_4, P4CALC_VER) : parse_p4calc;
            default                          : accept;
        }
    }

    state parse_p4calc {
        packet.extract(hdr.p4calc);
        transition accept;
    }
}
```

```
parser MyParser(packet_in packet,
                out headers hdr, inout metadata meta,
                inout standard_metadata_t standard_metadata) {

    state start {
        transition parse_ethernet;
    }

    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            0x800  : parse_ipv4;
            default : accept;
        }
    }

    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition select(hdr.ipv4.protocol) {
            6       : parse_tcp;
            default : accept;
        }
    }

    state parse_tcp {
        packet.extract(hdr.tcp);
        transition accept;
    }
}
```

(a) Input Programs.

```
parser VirtParser(packet_in pkt, out VirtHeaders VirtParam2,
  inout VirtMetadataType VirtMetadata,
  inout standard_metadata_t standard_metadata) {

    state start {
        transition select(standard_metadata.ingress_port) {
            (1 .. 2): P1_start;
            (3 .. 4): P2_start;
            (5 .. 6): P3_start;
        }
    }

    state P1_start {
        VirtMetadata.VirtProgramID = 1;
        transition parse_ethernet;
    }

    state P2_start {
        VirtMetadata.VirtProgramID = 2;
        transition parse_ethernet;
    }

    state P3_start {
        VirtMetadata.VirtProgramID = 3;
        transition parse_ethernet;
    }

    state parse_ethernet {
        pkt.extract(VirtParam2.ethernet);
        transition select(standard_metadata.ingress_port,
                          VirtParam2.ethernet.etherType) {
            (1 .. 2, 2048): parse_ipv4;
            (1 .. 2, _): accept;
            (3 .. 4, 4660): check_p4calc;
            (3 .. 4, _): accept;
            (5 .. 6, 2048): parse_ipv4;
            (5 .. 6, _): accept;
        }
    }

    state check_p4calc {
        transition select(pkt.lookahead().p, pkt.lookahead().four,
                                             pkt.lookahead().ver) {
            (80, 52, 1): parse_p4calc;
            default: accept;
        }
    }

    state parse_ipv4 {
        pkt.extract(VirtParam2.ipv4);
        transition select(standard_metadata.ingress_port,
                          VirtParam2.ipv4.protocol) {
            (1 .. 2, _): accept;
            (5 .. 6, 6): parse_tcp;
            (5 .. 6, _): accept;
        }
    }

    state parse_p4calc {
        pkt.extract(VirtParam2.p4calc);
        transition accept;
    }

    state parse_tcp {
        pkt.extract(VirtParam2.tcp);
        transition accept;
    }
}
```

(b) Merged Program.

Source: The Author.