UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

JÚLIA DARTORA CRAIDE

Switch (De)Composer++ Evolution and Practical Evaluation of Switch (De)Composer

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Engineering

Advisor: Prof. Dr. Weverton Cordeiro

Porto Alegre
April 2023

"Nothing in life is to be feared, it is only to be understood.
Now is the time to understand more, so that we may fear less."

— MARIE CURIE

ACKNOWLEDGMENTS

First and foremost, I need to recognize my advisor Weverton Cordeiro, your guidance and experience were fundamental to the success of this research, and you've been very special to me all these years. I would also need to thank Paula D. Bol for her initial proposal of Switch (De)Composer, without it this project would not be possible. And the help and incentive from UFRGS network group and the research team in laboratory room 208, especially Matheus Saquetti's insights related to the practical evaluation.

I would like to thank this university, its faculty, and its staff, that provide me with this unique learning experience. I'm especially grateful for all the great teachers, from basic education to now, whom I had the honor to learn from. I also met awesome classmates along the way who were very important and form true friendships, to them my gratitude. A big thanks as well to my colleagues and leaders from my internship at "SAP Labs São Leopoldo", who help me start my career in the best way possible.

I need to acknowledge the student groups I was part of, which had an immense impact on my academic and professional journey. Firstly, the "Laboratório de Programação Competitiva (LPC)" study group that help me train for programming competitions like ICPC, especially professor Rodrigo Machado who made me feel welcome right from my first semester. I also need to thank all members of "Empresa Júnior IDE", especially the ones that were on projects and/or the directory board with me, it was a pleasure working with you, I learn so much there, and hope I could be of help as well.

I'm extremely grateful for the support offered by my family and friends, who assisted me immensely. My deepest appreciation to my parents, Alex and Jesmarí, for always being there for me in the good, the bad, and the great times. My most sincere gratitude to my little sister Aléxia, who has been a blessing, sometimes in disguise, throughout my entire life. You three encouraged me so much to push forward in my studies and give my best in every situation, I cannot express in words all my gratitude.

Last, but not least, I need to offer the biggest thank you to Leonardo, who has been my partner all those years in and out of university. You've been with me every step of the way and offered unique feedback from the very first class we had together until the final revisions of this project. I'm so proud of all your accomplishments and your support means the world to me as well. I could not have asked for a better person by my side.

I apologize for those I did not have the chance to properly name here. And dedicate this as well to all the loved ones that are not here anymore and will be forever missed.

ABSTRACT

Switch (De)Composer is a solution proposed to create modular switch code leveraging the One Big Switch(OBS) abstraction according to a network topology. It enables network developers to deploy the code that promotes reusability, maintainability, and efficient resource usage, across the programmable forwarding plane. In this undergraduate thesis, we proposed Switch (De)Composer++ (CRAIDE, 2023) a continuation of the project aiming to enhance the solution and follow up with a practical evaluation on top of FPGAs, such as NetFPGA-SUME. The results obtained indicate significant improvements in latency and occupation when comparing switches generated from Switch (De)Composer to a trivial OBS model deployment.

Keywords: One Big Switch. Switch. Software Defined Network. P4. μP4. FPGA. NetFPGA SUME. P4VBox.

Switch (De)Composer++: Evolução e Avaliação Prática do Switch (De)Composer

RESUMO

Switch (De)Composer é uma solução proposta para criar código switches modulares se aproveitando da abstração One Big Switch (OBS) de acordo com a topologia da rede. Dessa forma permitindo desenvolvedores de redes realizarem a implementação do código e promovendo reusabilidade, manutenibilidade e uso eficiente de recursos, através do plano de dados. Nesse trabalho de conclusão de curso foi proposto o Switch (De)Composer++ (CRAIDE, 2023), uma continuação do projeto com objetivo de aprimorar a solução e em sequência fazer uma avaliação prática utilizando FPGAs, como NetFPGA-SUME. Os resultados obtidos indicam melhorias significativas de latência e ocupação, quando comparando switches gerados pelo Switch (De)Composer com uma implementação trivial do OBS.

Palavras-chave: One Big Switch. Switch. Software Defined Network. P4. μP4. FPGA. NetFPGA SUME. P4VBox.

LIST OF ABBREVIATIONS AND ACRONYMS

ACL	Access Control Lists
ASIC	Application-Specific Integrated Circuit
API	Application Programming Interface
BLE	Basic Logic Element
BMv2	Behavioral Model version 2
CLB	Configurable Logic Block
CLI	Command-Line Interface
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DRAM	Dynamic Random Acess Memory
DSP	Digital Signal Processing
FPGA	Field-Programmable Gate Arrays
ILP	Integer Linear Programming
IP	Internet Protocol address
IPI	Input P4 Interface
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
JSON	JavaScript Object Notation
LCS	Longest Common Subsequence
LoC	Lines of Code
LUT	Lookup Table
MAC	Media Access Control
μP4	Micro P4
MILP	Mix-Integer Linear Programs

NAT	Network Address Translation
OBS	One Big Switch
OPI	Output P4 Interface
OPL	Output Port Lookup
PCIe	Peripheral Component Interconnect Express
P4	Programming Protocol-Independent Packet Processors
PISA	Protocol-Independent Switch Architecture
RAM	Random-Access Memory
Regex	Regular Expressions
SDN	Software Defined Network
SRAM	Static Random Acess Memory
SSS	Simple Sume Switch
TDG	Table Dependency Graphs
TCP	Transmission Control Protocol
UFRGS	Federal University of Rio Grande do Sul

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

In recent years, the networking community has seen many advancements toward programming entire networks (GAO et al., 2020) or defining flow policies (ARASHLOO et al., 2016) using only a single file. These advancements are based on the One Big Switch (OBS) abstraction (KANG et al., 2013) that incentives network developers to be mindful of the behavior needed for forwarding planes globally, this already occurs on Software Defined Network (SDN) control and application planes. That way the programming necessary for the global network is facilitated, since from an individual file is possible to generate ASIC constraints that satisfy individual and/or network-wide constraints.

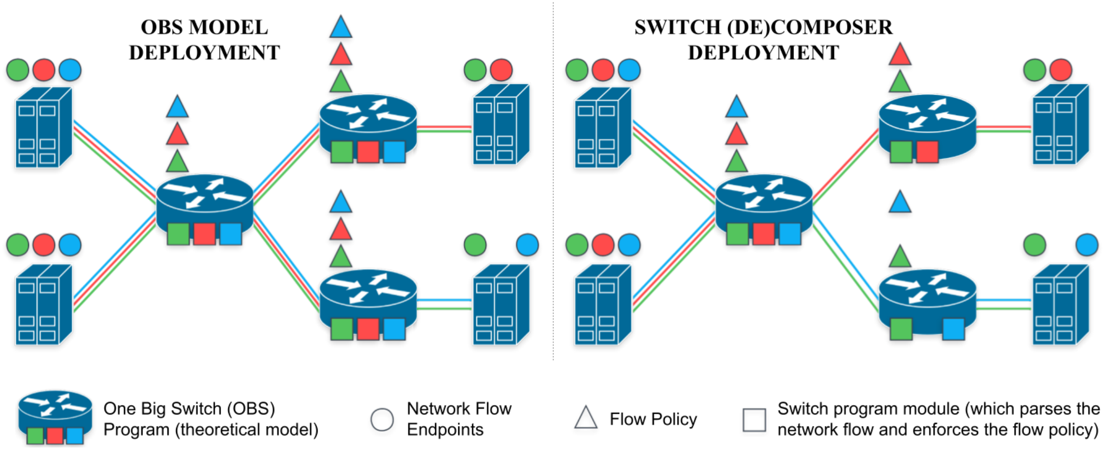
On the other hand, this abstraction can make developers less aware of run time constraints, like policies, when deploying and remain tied to the OBS model. Since all behavior are expressed on a single program, changes to specific protocols and services implementations may be difficultated, because they would depend o shared resources like headers and parsers. This can also make the program tightly coupled and tied to some scenarios, making it harder to reuse in different cases.

The proposed Switch (De)Composer (BOL et al., 2021a) is a solution that allows programmers to leverage the OBS abstraction, and simultaneously produce modular switch code, that facilitates reusability in different domains, functionality extensions, and independent maintainability. In order to produce modular custom switch code, Switch (De)Composer receives three parameters as inputs: the switch constraints, the target forwarding plane topology, and the big switch code. The switch constraints are the flow policies that must be enforced on a given flow. The forwarding plane topology, on the other hand, describes the architecture of switches and connections that decides the arriving package's destination. The big switch code, in this context, could be an OBS abstraction written in a modular language like μP4 (SONI et al., 2020b).

The solution can also generate switch programs that are more efficient in resource usage. As an example, Figure 1.1, has a network with three switches and four access points. Each colored line represents a flow, the triangles a switch constraint, and the circles a flow endpoint. Comparing Switch (De)Composer deployment with a strawman OBS, in this hypothetical case, the total number of modules is reduced from nine to seven.

While Switch (De)Composer presented very promising theoretical results for resource savings in large-scale network backbones and data centers, no practical evaluation was performed considering the particularities of real-world hardware implementations.

Figure 1.1 – OBS deployment vs. Switch (De)Composer deployment


OBS MODEL DEPLOYMENT
SWITCH (DE)COMPOSER DEPLOYMENT
One Big Switch (OBS) Program (theoretical model)
Network Flow Endpoints
Flow Policy
Switch program module (which parses the network flow and enforces the flow policy)

Source: The Author, adapted from (BOL et al., 2021a)

In particular, no assessment of potential resource savings and impact on network flows remained as a prospective direction to be analyzed in future investigations. In this context, the goal of this work is to provide a practical evaluation of Switch (De)Composer focusing on real-world hardware, also closing the gap on the refinement of OBS models considering the policies implemented in the network for each programmable forwarding device. We also aim to go beyond the previous development, adding more features and making important modifications, therefore achieving a more complete solution we call Switch (De)Composer++ (CRAIDE, 2023).

The remainder of this document is organized as follows. Chapter 2 elaborates in-depth on background concepts, whereas Chapter 3 reviews related work, both important topics used as the basis for the rest of the work. Chapter 4 provides an overview of Switch (De)Composer and the preliminary evaluation previously made about the projects proof-of-concept version (BOL et al., 2021a). Chapter 5 discusses the software improvements intended to be developed for this undergraduate manuscript and how those changes were actually implemented. Chapter 6 describes the steps we took in order to make the practical evaluation of Switch (De)Composer output switches on FPGA boards, followed by an examination of the premises and results of this evaluation. Finally, Chapter 7 has the outlook of what was developed in this undergraduate project, the conclusions taken, and mentions the possibilities for future work.

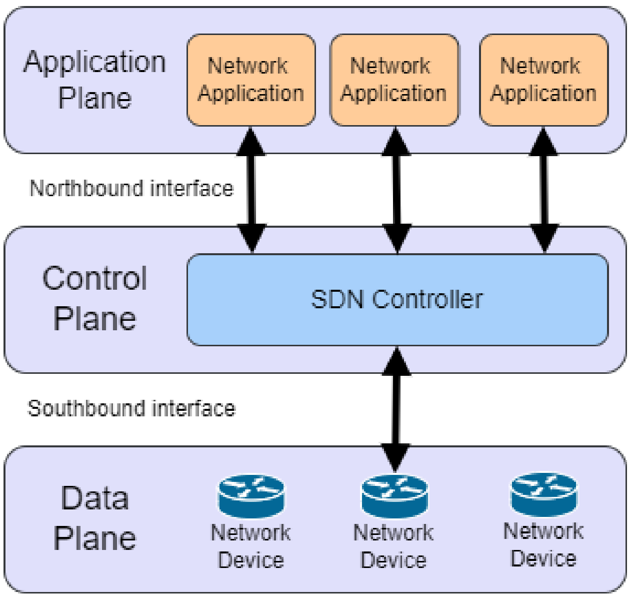
2 BACKGROUND

In this section, we discuss more in-depth about the relevant background for this undergraduate thesis. We conceptualize the main terms and technologies involved in the development of the Switch (De)Composer solution.

2.1 Software Defined Network (SDN) and OpenFlow

Software Defined Network (SDN) is a concept that gain a lot of traction in the last decades, due to the many benefits, like flexibility, controllability, maintainability, scalability, and reliability, it brings to network architectures compared to the classical model. The main idea behind SDN is to divide the network into planes as shown in Figure 2.1, that way control and forwarding functionalities into two planes, the control plane, and the data plane. That way, decoupling the physical devices from the routing and forwarding logic. Furthermore, in this architecture, the control plane is centralized in a single controller that oversees the network and defines the network policies that will be enforced (BENZEKKI; FERGOUGUI; ELALAOUI, 2016).

Figure 2.1 – Software Defined Network planes



Source: The Author, adapted from (GOBATTO et al., 2021)

Although SDN inspired many improvements related to network management on the control plane side, there are still concerns raised about the data plane (GOBATTO et al., 2021). These concerns are mainly related to how to extend the protocol sets on physical devices. One challenge that persists is that many network devices have functionalities predefined by manufacturers, which impede the support of newer protocols.

One of the first real implementations of Software Defined Networks was made by OpenFlow (BENZEKKI; FERGOUGUI; ELALAOUI, 2016), which is an open protocol that enables traffic management and communication between the control panel and the network devices. The initial OpenFlow(MCKEOWN et al., 2008) objective was to enable researchers to control and test their experimental flows on production environments, in a way they will be isolated from production traffic on the same network. The OpenFlow switch also attempted to fulfill four goals, to comply with high-performance and low-cost implementations, to support a large range of research, to assure traffic flows isolation, and to be consistent with vendors' need for closed platforms.

Currently, OpenFlow is one of the most commonly deployed solutions based on SDN (LARA; KOLASANI; RAMAMURTHY, 2014). The OpenFlow architecture has three main components, the OpenFlow-compliant switches are only responsible to forward the packets according to their flow tables, the controller provides the forwarding rules to the data plane by manipulating the flow tables, and a secure channel is the interface in which the control plane applications and data plane elements can communicate via the OpenFlow protocol. The flow tables consist of a list of entries that has match fields, a counter, and instructions, when a packet arrives in the switch its header fields are compared against the entries, and they equal to an entry match field they are processed following the instructions actions, while the counter is used to keep entries statistics.

In such a manner, OpenFlow provides an abstraction (NGUYEN et al., 2016) that hides the complexity of the network devices thus facilitating and giving freedom to the network management operators. This abstraction allows any high-level policy to be translated into flow entries that can be distributed to the switches in a network topology. However, a current open challenge is how to translate these high-level policies into lower lever forwarding rules, while also satisfying resource constraints.

2.2 One Big Switch (OBS)

One Big Switch is a popular abstraction amongst the networking community, that considers the whole network as one (big) switch (KANG et al., 2013). That way, it conveniently hides the internal package forwarding topology details so the network developer can focus on developing the global forwarding plane. This way all the necessary code can be written on a single file, that can be used to generate specific switches in a topology.

One of the main challenges in implementing this abstraction is how to optimize the mapping of the high-level policies into low-level ones to be placed in each switch on the topology (KANG et al., 2013). This is also known as the big switch problem. The trivial solution to this problem is to implement all possible policies in all switches, but this results in a poor resource usage efficiency.

2.3 P4 and μP4

Programming Protocol-Independent Packet Processors (P4) is a high-level language for programming the forwarding behavior of protocol-independent packets on network devices (BOSSHART et al., 2014). The three main goals of the language are: (*i*) reconfigurability in the field, (*ii*) protocol independence, and (*iii*) target independence (GOBATTO et al., 2021). Reconfigurability in the field means that the controller should be able to change how a given field is parsed and processed once deployed. Protocol independence means that the controller can configure the parser that extracts the necessary header field and the match action table to process the field, this way the device is not tightly coupled with a specific protocol and/or packet format. Finally, target independence means the P4 code, which describes the processing function, should be written independently of the underlying hardware, the P4 compiler is the one responsible to turn the P4 program into a target-dependent program.

One of the main challenges of P4 programs is related to portability and reusability, since the P4 code tends to be monolithic and dependent on underlying hardware architecture. The μP4 framework (SONI et al., 2020b) was created to address those challenges, it provides logical architecture abstractions needed to compose data planes independently from the hardware-level structures. It also supports a powerful kind of program composition, enabling developers to write modular code and use functions from other programs and/or reusable libraries. Finally, a compiler that transforms the μP4 programs into P4 programs was provided, so the code can be deployed on the same environments.

2.4 Mininet

Mininet is an open-source tool that allows its users to emulate large virtual networks, with support for OpenFlow devices and SDN controllers (KAUR; SINGH; GHUMMAN, 2014). It is an alternative to the physical testbed, which can be very expensive and hard to configure, and to other virtualization and simulators, which can require code modification, and may struggle with scalability, and speed. Compared to the mentioned alternatives, Mininet has a lower cost, is more convenient to use, and enables rapid prototyping, realistic accuracy, and scalability.

In other to provide the emulation of scalable networks on a single machine Mininet uses lightweight virtualizations, such as process-based virtualization and network namespaces (CONTRIBUTORS, 2022). These techniques permit the quick creation, customization, and sharing of prototypes, but are also the reason why Mininet depends on the Linux kernel. The Mininet code was mostly written in Python, that is why an extensive Python API is provided to create and test network experiments.

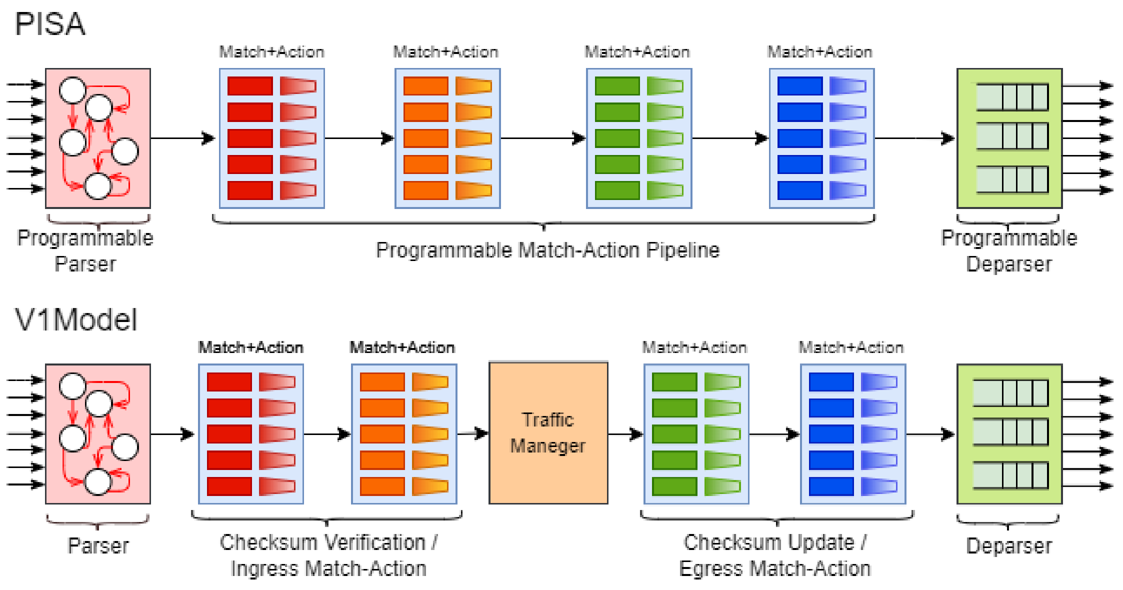
Applicability is a big advantage of Mininet (OLIVEIRA et al., 2014) since its resources can be replicated in real environments and other test infrastructures without extensive changes. It also exceeds expectations in prototyping and sharing since it provides fast startup times, low overhead to test designs, the possibility to test on modest hardware and ease to share projects, tests, and configuration by allowing multiple developers to work independently on the same topology. However, the tool has its limitation especially regarding network fidelity on larger scales, since all nodes run on a shared computer, therefore being bound by its CPU and bandwidth.

2.5 Intel Tofino and V1Model

The Intel Tofino (BYTE, 2017) is a network processing platform designed to handle the growing demands of data traffic and complex network management tasks in modern data centers. Based on Application-Specific Integrated Circuit (ASIC) chip technology, the Tofino offers a high throughput and ultra-low latency, making it ideal for organizations that need to handle large volumes of real-time data. Additionally, the Tofino is highly programmable and customizable, using P4 allows developers to program the data plane, creating network solutions to meet the specific needs of their customers. It also has metadata export features that using P4, allow the programmer to send metadata to orchestration platforms for better control.

The Barefoot Tofino switches are compatible with P4 programs that follow the V1Model architecture (P4LANGUAGE, 2022). This architecture is included with the P4C compiler and was created to support version $P4_{16}$ of the P4 language. It is also fully compatible with the architecture used on the previous P4 version $P4_{14}$ and implemented on top of the BMv2 framework and Simple Switch target implementation.

Figure 2.2 – PISA Architecture vs V1Model Architecture



Source: The Author, adapted from (P4.ORG, 2017)

V1Model (P4.ORG, 2017) also follows the principles from Protocol-Independent Switch Architecture (PISA), since it has a P4 programmable input parser, followed by match-actions, and at the end a P4 programmable deparser as shown in Figure 2.2. The difference to the basic PISA architecture is that it has defined blocks on the math-action, those being a block to checksum verification, an ingress pipeline, it has a traffic manager, an egress pipeline, and finally a block to update checksums on the packets. The traffic manager is can schedule and replicate packets between I/O ports, it is also not programmable in the P4 language.

2.6 Field-Programmable Gate Arrays (FPGA)

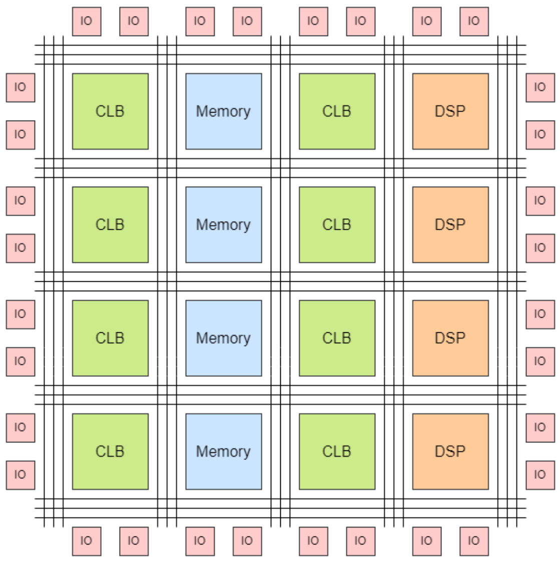
Field-Programmable Gate Arrays (FPGA) are prefabricated programmable devices that can be changed into nearly any integrated circuit needed (KUON; TESSIER;

ROSE, 2008). This reconfigurability and flexibility capacities are one of the main differentials of FPGAs allowing them to change the hardware at runtime and be used as fast prototyping. Compared to a microprocessor, FPGAs can change the hardware itself changing data and control flow (YANG et al., 2014), which leads to more efficiency in performance and power consumption.

Compared to their ASICs counterparts, FPGAs are cheaper to buy (low volume cost), fast to set up, and can be electrically reconfigured in case of a mistake or a necessary change, while ASICs are fixed-function, have a high cost and can take a long time to be fabricated (KUON; TESSIER; ROSE, 2008). The trade for these advantages is that FPGAs have a significantly higher cost in area, delay, and power consumption: an FPGA requires around 20 to 35 times more area than an ASIC made with standard cells, and it has a speed performance of approximately 3 to 4 times slower and consumes approximately 10 times more energy (KUON; ROSE, 2007). These losses in performance, area and power consumption are mainly caused by the routing mesh used by the FPGA to connect the data traffic between its internal logic blocks.

The fundamental logic blocks that compose FPGAs are Configurable Logic Blocks (CLBs), Digital Signal Processing (DSP) slices, and RAM memory blocks, as shown in Figure 2.3 these blocks are internally organized in a matrix and connected by a mash. The CLBs are the main FPGA block used to implement, sequential, combinatorial, and logic functions. DSPs are focused on efficient processing of digital signal functions, multiplication being one of the most important examples (GOBATTO et al., 2021).

Figure 2.3 – FPGA internal organization



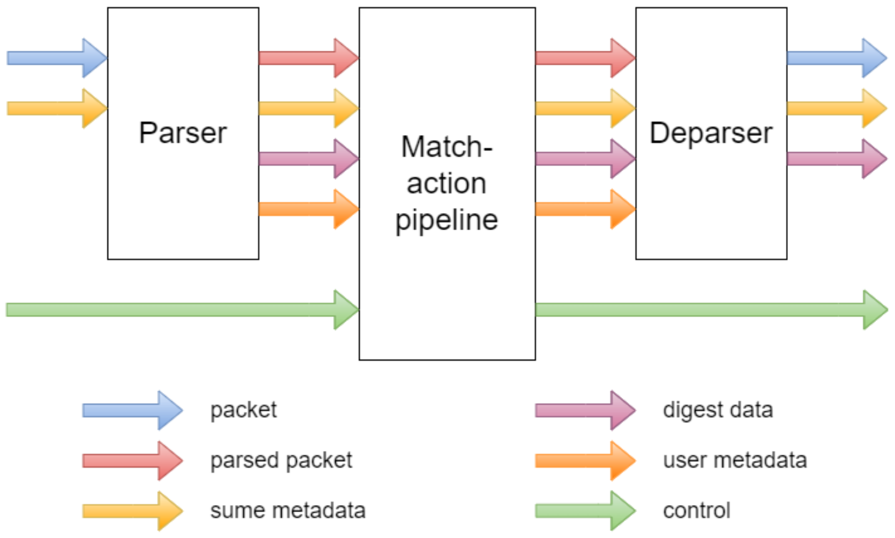
Source: The Author, adapted from (GOBATTO et al., 2021)

2.7 NetFPGA and NetFPGA SUME

NetFPGA(IBANEZ et al., 2019) is an open-source project that provided software and hardware to simplify high-speed networking research. The project is focused on academic and education communities, so beyond software and hardware, the NetFPGA team also supports learning resources such as tutorials, workshops, forums, and summer camp events. The community around NetFPGA is strong leading to many contributions from members on the related projects and the education resources mentioned.

NetFPGA SUME (ZILBERMAN et al., 2014) is a NetFPGA platform created to enable research and development focus on the high requirement of the data center networks, such as large bandwidth and throughput. It is an FPGA-based PCIe board that has support for an I/O that exceeds 100Gb/s and is provided by 3013.1GHz transceivers, it also has other practical interfaces, SRAM, and extensible DRAM memory. The NetFPGA SUME is the third and latest FPGA board on the NetFPGA hardware family, its main objectives were to have a low-cost, commodity and flexibility to be used on a wide range of research applications since it was designed for the academic community.

Figure 2.4 – SimpleSumeSwitch Architecture



Source: The Author, adapted from (IBANEZ; ZILBERMAN, 2018)

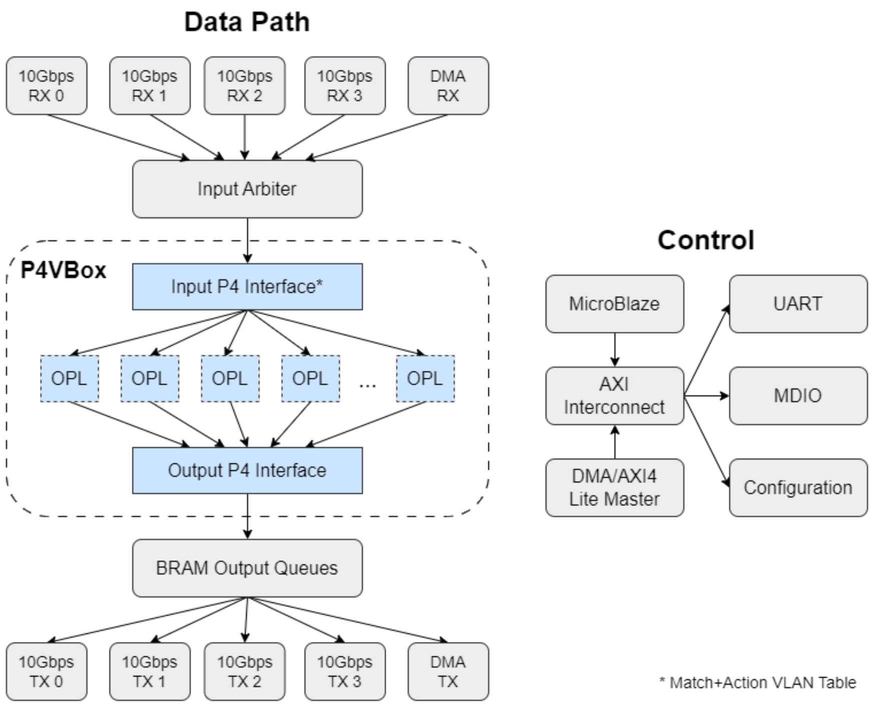
In implementations of switches using the NetFPGA platform, we follow the Simple Sume Switch (SSS) model[1]. The SSS model consists of and P4 architecture with 3 switch logic steps. The first step is a packet parser, followed by a single match action pipeline and then a packet deparser as shown in the Figure 2.4.

[1]https://github.com/NetFPGA/P4-NetFPGA-public/wiki/SimpleSumeSwitch-Architecture-(v1.2.1-and-Earlier) (IBANEZ; ZILBERMAN, 2018)

2.8 P4VBox

P4VBox (SAQUETTI et al., 2020) in an architecture that enables the virtualization of independent P4-based switches on the data plane and implements a methodology for hot-swapping the virtual switches deployed. This architecture aims to satisfy 5 simultaneous virtualization challenges: (*i*) virtual switch instance decoupling from the virtualization environment, (*ii*) network flow isolation, (*iii*) hardware resource isolation, (*iv*) virtual networking within the hypervisor, and (*v*) feasible performance and memory footprint.

Figure 2.5 – P4VBox design



Source: The Author, adapted from (SAQUETTI et al., 2020)

The architecture increments the canonical NetFPGA reference design by replacing the single Output Port Lookup (OPL) for a structure that introduces Input P4 Interface (IPI), an Output P4 Interface (OPI), and multiple OPL instances. Figure 2.5 shows the P4VBox design, where it replaces the single OPL from the canonical design. P4VBox was tested on a NetFPGA SUME with 3 types of virtual switches: layer-2 switches, routers, and firewalls, obtaining very positive results for bandwidth and latency.

3 RELATED WORK

In this section, we present investigations related to our by also proposing solutions to enhance switch programmability for the forwarding plane. Among these investigations, some are also related to the One Big Switch abstraction, some to the Software Defined Network concept, and some to the usage of P4 language as a basis for their optimizations. As so understanding these different approaches may lead to meaningful insights for analyzing the Switch (De)Composer results. We provide in Chapter 4 a more in-depth comparison of the related work that form the conceptual basis of this work.

3.1 Switch (De)Composer

Switch (De)Composer was initially a solution proposed on a 2021 SIGCOMM poster (BOL et al., 2021a) its main objective was to be able to generate a modular switch code given policies constraints and the topology of the desired network. The idea behind it was to leverage the One Big Switch abstraction, it did this by using OBS model μP4 program previously annotated in the places where it would invoke and instantiate other modules, then the solution would include the required modules from the OBS program onto a template program that has also been annotated, therefore generating switch program with only the necessary modules for all switches in the topology. That way it would allow the network programmer to generate and deploy switches on the programmable forwarding plane that are easier to maintain due to its reusable modules, and that fosters efficient hardware resource usage since it only included the required modules.

The solution was developed in Python 2 and was composed of a program that did the process of constructing the μP4 previously described, another Python program to run tests on Mininet, the μP4 templates and modules, and a few examples of the solution utilization. The initial constraints supported were network policies, mainly, ethernet, IPv4, and IPv6, since they would be used as the basis to later support further constraints. The project also previewed the utilization of a constraint interpreter to directly read the policy's constraints to determine which modules should be included in each switch, but this was not developed before the work on the project cease.

The solution was also not able to be tested on the target FPGA hardware, so the poster evaluation was based on the number of switch tables μP4 and JSON on two topology scenarios, obtaining results that indicate a potential to enhance resource usage. This undergrad project is a continuation of the development of the Switch (De)Composer solution, where one of the final objectives is to make a practical evaluation on top of FPGAs. Therefore the results of this Switch (De)Composer proof-of-concept preliminary evaluation will be discussed in depth in Chapter 4, and mentions of the original proof-of-concept will be present in the remaining text.

3.2 SNAP

SNAP is a stateful programming model used to develop SDN programs on top of the one big switch abstraction (ARASHLOO et al., 2016). The main challenge it solves is providing advanced managing support for state persistence on the data plane, enabling programmers to implement a large range of stateful applications. SNAP was evaluated and validated for the expressiveness of its language and the in relation to the compiler process on a wide range of sample programs.

In order to provide the stateful model SNAP provides global persistent arrays to which the OBS programs can read and write from. The SNAP compiler is the component responsible to take care of the read/write dependencies on the data arrays. It also uses an internal representation based on binary decision diagrams to translate OBS programs, and finally, these diagrams are transformed into Mix-Integer Linear Programs (MILP) which optimizes state placement and the traffic routes on top of the topology.

The SNAP language provides stateful operations alongside packet processing, it is based on the NetCore/NetKAT family of languages, in which a program consists of a set of predicates and policies. In the SNAP context, a predicate receives the packet and must decide based on state reads either to drop or pass the packet, but it cannot alter the state. Policies, on the contrary, still need to process the packet, but they are able to modify packets and the state, therefore every predicate is a policy that does not modify.

3.3 Flightplan

Flightplan (SULTANA et al., 2021) is a proposed target-agnostic toolchain that can disaggregate a single P4 program into cooperative programs to run on heterogeneous data planes. It also maps the subprograms generated to run on distributed systems on different kinds of hardware, leveraging their strengths and exploiting features. That way it keeps the same code behavior while optimizing bandwidth, energy consumption, device heterogeneity, and latency, as it has shown in its evaluation.

The code segmentation on Flightplan was not automated at the time of the original publication, but it consisted of adding annotations to delimit boundaries for the segmentations. Those segments of code are then automatically transformed into abstract programs that consist of Directed Acyclic Graphs(DAGs). Then these programs, in conjunction with resource semantics rules, network topology description, and optimization objectives, are sent to the Flightplan planner which will lazily and exhaustively generate plans that satisfy the given plan input constraints.

Then the generated plan consists of 3 components the allocation model, the annotated program, and a control profile. The first one is used to explain how the allocation found will be modeled when the program executes across data planes. The second is the original program, with annotations about how the segments will be mapped to the data plane, which will later be used to split the programs and compiler to the specific target device. Finally, the control profile is used to configure Flightplane runtime, by distributing the programs, starting the execution, and querying its states.

3.4 SPEED

SPEED is a system for the deployment multiple of data plane programs (CHEN et al., 2020) on a network with programmable switches. It does it by merging different data plane programs written in $P4_{14}$ or $P4_{16}$ to reduce redundancy in the code deployed. The SPEED objective was to achieve high performance since many applications demand strict performance requirements and resource efficiency in order to not exceed the network programmable switches' internal capacity.

The three biggest challenges in the implementation of SPEED were acceptance of a diversity of programs, dealing with the strict performance and resource constraints, and preserving the data plane program logic when splitting it into multiple switches. To fulfill its requirement, Speed reduces program redundancy by merging input data, abstracting the network to the OBS model, and deploying in stages the merged program, that way reducing the resources used. It also maps the OBS on the substrate network respecting the given constraint and aiming for optimal placement to enhance performance.

SPEED's program merging algorithm is based on Longest Common Subsequence (LCS) problem and uses generates Table Dependency Graphs (TDG), which help to deal with heterogeneous programs. It also provides metadata sharing and inter-device packet scheduling to ensure the correct behavior of the original logic, after splitting among switches. To test the solution it was constructed a testbed with a Barefoot Tofino switch and server to send data, which was able to achieve positive results regarding resource efficiency and high end-to-end performance.

3.5 P²GO

P²GO (WINTERMEYER et al., 2020) is a system that works alongside the P4 compiler and uses runtime information to do profile-guided optimizations to the programs and their placement in hardware. This approach could also reduce compilation fails since it considers actual data with unrealistic and infrequent inputs. That way the compiler can leverage real traffic information to make possible a broader spectrum of optimizations to minimize resource utilization.

To more efficiently allocate resources to the P4 programs, the three main things P²GO does are: (*i*) remove not manifested dependencies, (*ii*) adjust table and register sizes to reduce the pipeline length, and (*iii*) offload rarely used parts of the program. It works alongside the compiler by iteratively modifying, compiling, and analyzing its changes compared to the original program, ensuring the same behavior in the given traffic trace. It also can present information about program usage, so the developer can examine unused pieces of code for further optimization based on a particular observation.

The idea behind the profile-guided optimizations comes from general-purpose languages which use execution profiles to facilitate memory and pipeline stages optimization. P²GO works with four phases to accomplish its optimizations iteratively, the first one is profiling the program the next three use this to make optimization, there is a phase for removing dependencies, then reducing memory, and finally offloading code to the controller. The prototype evaluation was done on a Tofino switch and showcased the benefits of profiling techniques applied to the P4 compiler.

4 OVERVIEW AND PRELIMINARY EVALUATION

In this chapter, we will present Switch (De)Composer, in terms of its design vision and evaluation reported. Section 4.1 explains the core questions that motivated the creation of the solution. In Section 4.2 we show the initial evaluation of Switch (De)Composer preliminary version and comparison with other bodies of work.

4.1 Switch (De)Composer Overview

Switch (De)Composer design (BOL et al., 2021a) addresses three main questions: (*i*) What switch constraints to support? (*ii*) How to compute switch module dependency based on the expressed constraints? And (*iii*) How to build a custom switch program from a given set of modules? As it is, the answer to the first question is that Switch (De)Composer only has network policy constraints. The reason for this is as a way to enforce further constraints, the switch must include the code to support the network policies, so those were the first addressed. As an example, a switch must include an IPv6 handling module, in order to enforce IPv6 routing policies.

About question number two, Switch (De)Composer uses module annotations, inside the OBS module to compute dependencies based on the constraints. For that was made a constraints interpreter, specific to the constraint type, to determine the necessary modules. To illustrate this process let's imagine we have "match: tcp.dport == 80; action: drop" as a policy constraint. In this scenario, the interpreter should find the switch module annotation, written by the network developer, and would include a module to handle the TCP flows. And then when parsing the TCP header, it may also require a module for parsing IPv4 or IPv6, thus making a dependency graph of necessary modules.

About the third and final question, to build the custom switch programs the original version of Switch (De)Composer solution was to use templates to base what modules should be included in a given switch. In the development of this project, we decided to stop working with templates to start working only with the OBS μP4 module itself. So now the network developer will provide a dependency graph where there must be a head OBS module that serves as the foundation to build the switch. This base module must contain annotations like *@ModuleInvokeBegin(submoduleName)*, *@ModuleInvokeEnd(submoduleName)*, *@ModuleInstantiateBegin(submoduleName)*, and *@ModuleInstantiateEnd(submoduleName)*, surrounding lines that refer to a given submodule, which the name is between the parenthesis. As an alternative Switch (De)Composer also allows annotating the switch internal modules from a repository for switch code composition.

Algorithm 1 Switch (De)Composer Main Routine (BOL et al., 2021a)

```
Input: Forwarding Plane Topology, OBS Program
Output:Set of Custom-Made Switch Programs
1: for each switch in forwarding plane topology do
2:     $RequiredModules \leftarrow \textbf{FetchAndInterpretConstraints}(switch)$
3:     $FullModulesList \leftarrow \textbf{ComputeDependencies}(RequiredModules, OBSProgram)$
4:     $CustomSwitchProgram \leftarrow \textbf{GenerateProgramFromTemplate}(FullModulesList, OBSProgram)$
5: end for
```

Lastly, Algorithm 1 is the one used to generate the custom code for each switch in the forwarding plane topology. The solution may foster higher resource usage efficiency across the network, but it is depended on the constraint placed. For switches policies, one alternative is SNAP (ARASHLOO et al., 2016), which uses Integer Linear Programming (ILP) to compute the optimal policy placement.

4.2 Preliminary Evaluation

The original proof-of-concept of Switch (De)Composer[1] was made in Python and μP4(SONI et al., 2020b), and two scenarios were evaluated (BOL et al., 2021a), using Mininet and BMv2. The first scenario is the right one illustrated in Figure 1.1. The second Stanford topology SNAP was used. The network constraint policies used in the scenarios were IPv6, IPv4, and NAT/ACL. The test methodology was to sum the number of μP4 and JSON switch tables used to deploy the OBS program and our custom Switch (De)Composer switches, to asses the solution's effectiveness.

Table 4.1 – Number of Switch Tables (Cumulative)

Scenario	**Number of Tables Required**			
	OBS		**Switch (De)Composer**	
	JSON	μP4	JSON	μP4
1	72	15	53 (▼26%)	11 (▼26%)
2	624	130	554 (▼11%)	114 (▼12%)

Source: (BOL et al., 2021a)

The outcome of each scenario can be seen in Table 4.1, the quantity of match-action tables was reduced by approximately 26% in the first scenario, and approximately 11% in the second. These results suggest a potential in Switch (De)Composer for generating resource savings in Switches implementation. This evidentiates the need for further

[1]https://github.com/pauladbol/SwitchDeComposer (BOL, 2021)

testing on a target NetFPGA-SUME (SAQUETTI et al., 2019; SAQUETTI et al., 2020).

Switch (De)Composer shares the OBS philosophy with related work such as Flightplan (SULTANA et al., 2021) and SPEED (CHEN et al., 2020). Our solution may lead to more independent and reusable programs when compared to Flightplan since the last one splits P4 programs into modules tightly coupled to network domains and scenarios. SPEED, on the other hand, merges input programs into a single OBS and then slice them into stages and assigns them into switches. That way SPEED assumes that all switches in the topology have the same amount of stages, while Switch(De)Compose could have support for receiving the number of stages as a property. SPEED also cannot support constraints, like policies, differently from what we proposed for Switch (De)Composer. A more similar direction is seen in P²GO (WINTERMEYER et al., 2020) uses insights gathered from network flows to generate switches optimizing for resource usage, but it does not target the OBS model.

Table 4.2 allows a more visual comparison of our solution with all related works presented. We can see that the solutions provide switch code optimization and most also use the OBS model in order to do so. However, none of the related works used the network policies to optimize the switch code deployment except our solution.

Table 4.2 – Related Work Comparison

Work	Optimize switch code	Use OBS	Use policies as constraint
SNAP	X	X	
Flightplan	X	X	
SPEED	X	X	
P²GO	X		
Switch (De)Composer	X	X	X

5 OUR EXTENSION: PROPOSED ARCHITECTURE

This chapter will discuss the software solution Switch (De)Composer++ provides. We begin with Section 5.1 where we elapse about what we wanted to achieve with the evolution of the original tool and why. Then in Section 5.2 we describe in detail the necessary modifications and extensions done to the project. After showing the changes done to the project we will describe the typical use case in Section 5.3. In this example, we will also mention the optional customizations made possible by this project. To finalize, in Section 5.4 we will present a quantitative analysis based on lines of codes changed.

5.1 Proposed Architecture

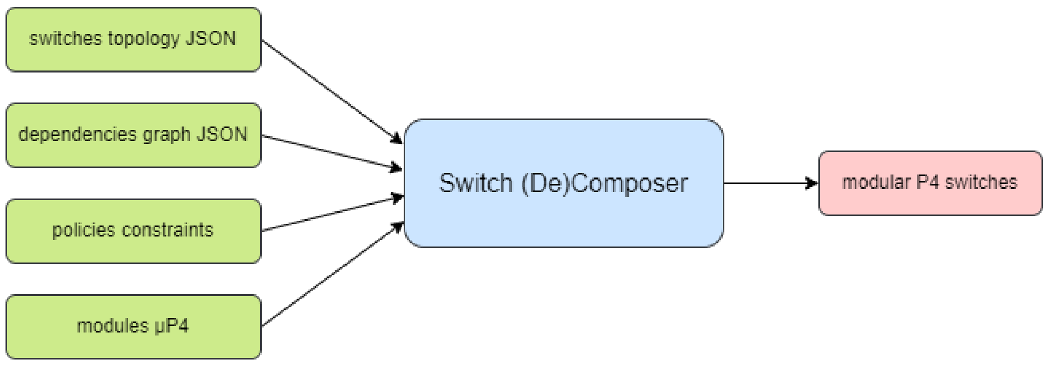
The first proof-of-concept of Switch (De)Composer had a lot of shortcuts that needed to be fully developed before it could become e suitable tool. Because of these limitations, we aim to make improvements to the architecture and remove some hard-coded dependencies in order for it to accept more diversity in constraints and topologies.

The technologies chosen to be used in Switch (De)Composer++ were Python 3 language and Shell script for the tool itself and μP4 for the modules. The choice of using μP4 language had already been made for the proof-of-concept, the reason behind it was to benefit from the abstractions it provides, which enable more modularity, portability, and reusability of code compared to the P4 language. The use of Python 3 and Shell scripts is because those are famous tools for automation, and that is something we needed to do on Switch (De)Composer++ to simplify the workflow. Also about Python specifically, the project already used Python 2 in the main module, but we opt to upgrade the language version to Python 3 since Python 2 was discontinued in 2020.

The objective of this undergrad dissertation is to make a new version with some major improvements to the switch development process. Figure 5.1 shows the expected inputs and outputs from Switch (De)Composer++ as if it was a black box. The only inputs we want to have are: a JSON file with the switches topology; a JSON file that explicitly the dependency graph for the One Big Switch file used; policies files for each switch in the topology; and, all the modules, including the base One Big Switch μP4 code. The expected output was for our project to generate a series of modular P4 switch programs according to the given topology input.

In the original proof-of-concept of Switch (De)Composer, it was defined a series

Figure 5.1 – Switch (De)Composer++ basic architecture overview



Source: The Author

of compilation steps needed to get from the μP4s to the final P4 switches, including testing the topology. Each step corresponds to a command that was previously executed in the terminal. The steps defined were:

1. The first step was to run the main Switch (De)Composer program that generates μP4 code for the switch
2. Then we need to compile the μP4 code for the submodules that are going to be imported by our switch code
3. Then we compile the μP4 into P4 code
4. After that we can take the resulting P4 program and compile it
5. This final step is not mandatory, but after doing the first four steps to all switches in the topology, we can run a Mininet to test the generated switches

In the proof-of-concept, the commands to run these steps were handwritten on the terminal or in a Shell file and then executed, this was prone to error, especially due to spelling errors or problems with file paths. The improvement proposed for this new version of Switch (De)Composer architecture is for it to automatically generate a Shell file with all commands to do the steps sequentially. This script file could be run right after its generation, but could also be used to test changes to the OBS and μP4 modules in the same topology since it does all generation and compilation processes.

We proposed to remove some hard-coded dependencies, such as the topology and the dependency graph. For the topology, we defined a JSON file where the network developer could declare the switches, hosts, and connections necessary for creating the topology. Another topology-related improvement needed is to create a Mininet script that uses these values to test the topology and P4 switches since the previous script only accepted the configuration from Figure 1.1. In relation to the dependency graph, we made also propose a JSON file that would contain an array of the modules, where each module will list its direct dependencies, direct dependencies being all submodules called inside the code, this array will later be turned into a directed graph used in the Python generator.

One of the promises of Switch (De)Composer was to be able to add modules according to the policies constraint, but this functionality was not implemented on the proof-of-concept (BOL et al., 2021a). So we plan on building a constraints interpreter that could read the policy document searching for specific expressions that indicate that a module should be included, then our program will also get all dependencies necessary to ensure all modules are implemented correctly. Nonetheless, we decided to keep providing the option to specify the wanted modules by picking them manually in case the network developer wants to have more control over the modules included.

We also cease using the idea of templates that the initial Switch (De)Composer (BOL et al., 2021a) had, because it could cause some confusion, so in the new version we only work with the base OBS program as the head module on our dependencies graph. That way instead of adding code from the OBS to a template file we already consider the main OBS module as the template, since it is the source of all modules. So in this new approach, when a submodule is not used we remove the declarations, instantiations, and invocation of this module, based on the topology definition and dependency graph, instead of adding code to the main OBS.

Besides that, we aimed to make improvements to the organization of the repository, making it easier to run and maintain. Among those proposed modifications many were simple changes like adding folders to improve the organization of the programs, enhancing the Python code using language best practices, and changes to the git repository to make it easier to contribute. The most impactful among those minor changes was that, on the proof-of-concept version, we needed to copy the repository code into a frozen version of the *obs-microp4*[1] repository forked from the original μP4 repository (*MicroP4*[2] (SONI et al., 2020a)). So we aim to invert this dependency in order to improve this process of installation and make Switch (De)Composer easier to implement and use.

[1]https://github.com/pauladbol/obs-microp4 (BOL et al., 2021b)
[2]https://github.com/cornell-netlab/MicroP4

5.2 Architecture Improvements

One of the first necessary improvements was to make a program that could create a Shell script to automate the execution of all steps needed in order to build the μP4 and turn it into P4. So based on this necessity *generate_distribute_programs.py* was made using Python3, the Python language was chosen since the project already used Python to make the main functionality of combining the μP4 modules. The output Shell objective was to be able to run all the steps and print helpful messages to the terminal, such as which step is currently executing and the occasional error messages.

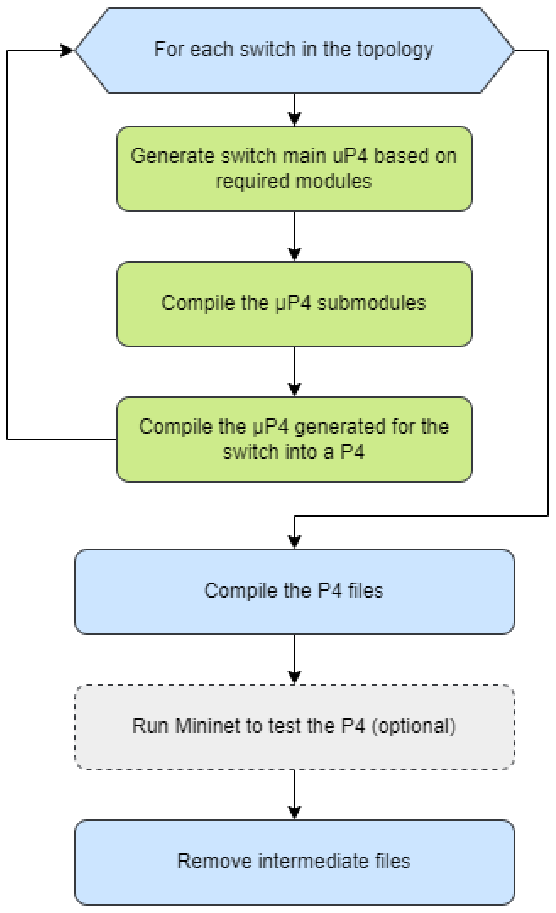
This change to Switch (De)Composer saves a lot of time, since, before it, all steps were run manually on the terminal, additionally, it also decreases the chance of making a mistake by typing the wrong command during the building process. This is especially true as the number of switches in your topology increases since you'd need to run at least 3 commands per switch, but this number can be way larger depending on how many modules are necessary for each switch. Figure 5.2 shows the steps that are now done by the resulting Shell file. In addition to the steps discussed in the previous chapter, there is a new one to remove the intermediate files generated during the compilation and testing.

The reason for adding the new step to remove the intermediate files was identified during the development and testing of this project since there were many files such as JSON files, p4i files, p4rt files, and μP4 files, created during the build process. After the build was finished all of them became useless, but none of them were deleted originally. This would crowd the output folder, especially after the Mininet test, thus making it harder to find the actual P4 programs containing the output switches. This step was simple to make, we just use a command to remove files with the extension previously mentioned.

After this, the next big improvement made was to change the way we accepted the topology inputs. Before it was all made by arguments passed to the commands in the terminal, but this was prone to error since one typo could be the cause of the error on multiple steps, and that was not easy to catch. So we changed the input to be a JSON file, which describes the switches, host, and how they are connected.

The JSON needs to have 3 properties, one called *switches* that has an array of objects, where each object describes a switch in the topology, with attributes like name, modules needed, policies file name, MAC address, and the dependencies graph used, some of them will be further explained later. Another property of the topology file is *hosts*, just like *switches* it has an array of objects, where each object describe a host,

Figure 5.2 – Switch (De)Composer++ steps to generate the switches


For each switch in the topology
Generate switch main uP4 based on required modules
Compile the µP4 submodules
Compile the µP4 generated for the switch into a P4
Compile the P4 files
Run Mininet to test the P4 (optional)
Remove intermediate files

Source: The Author

with attributes like hostname, MAC address, IPv4 address, IPv6 address, the switch it is connected to and in which switch port. The last JSON parameter is the *switchlinks* which describes the connections among the switches. Listing 5.1 presents an example of a topology JSON file with 3 switches that have the same configuration as in Figure 1.1.

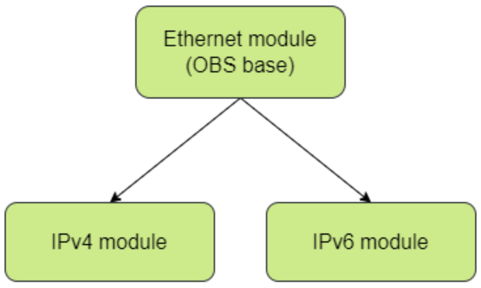
Listing 5.1 – Topology JSON example

```
{
    "switches": [
        {
            "switchname": "s1",
            "modules": "all",
            "dependencies": "dependencies_e1.json",
            "mac": "00:aa:bb:00:00:01"
        },
        {
            "switchname": "s2",
            "policies": "s2_policy_e1.txt",
            "dependencies": "dependencies_e1.json",
            "mac": "00:aa:bb:00:00:02"
        },
        {
            "switchname": "s3",
            "modules": "ethernet,ipv6",
            "dependencies": "dependencies_e1.json",
            "mac": "00:aa:bb:00:00:03"
        }
    ],
    "hosts":[
        {
            "hostname": "h1",
            "switchname": "s1",
            "mac": "00:00:00:00:00:01",
            "ipv6": "2021::1/64",
            "ipv4": "10.0.1.1/24",
            "port": "1"

        },
        {
            "hostname": "h2",
            "switchname": "s1",
            "mac": "00:00:00:00:00:02",
            "ipv6": "2022::1/64",
            "ipv4": "10.0.2.1/24",
            "port": "2"
        },
        {
            "hostname": "h3",
            "switchname": "s2",
            "mac": "00:00:00:00:00:03",
            "ipv6": "2023::1/64",
            "ipv4": "10.0.3.1/24",
            "port": "3"
        },
        {
            "hostname": "h4",
            "switchname": "s3",
            "mac": "00:00:00:00:00:04",
            "ipv6": "2024::1/64",
            "ipv4": "10.0.4.1/24",
            "port": "4"
        }
    ],
    "switchlink":[
        ["s1", "s2"],
        ["s1", "s3"]
    ]
}
```

The way Switch (De)Composer++ handles the dependencies also changed, it was added the idea of a dependencies JSON file that has a list of the module's objects, each object needs to have 3 mandatory parameters: the module name, the module file name, and an array of direct dependencies. In this context, direct dependencies are all the submodules called inside the module described. The dependencies JSON file also has three optional parameters: function, regex, and head, the first and second will be further explained later, while the third represents if a file is the head of the dependency graph. The head module of the dependency graph is also called the OBS base code since it is from it that all switches are built, therefore should only exist one head per dependency graph. Figure 5.3 has an example of the dependency graph used in our test cases, the base module is the switch module that just implements the ethernet protocol.

Figure 5.3 – Switch (De)Composer++ dependency graph example



Source: The Author

This dependencies graph needs to be a directed graph with no cycles and needs to have a head node, so it may be also called a dependencies tree. Because of that we sometimes use nomenclature related to trees, such as the head module and leaf modules. The dependencies graph is used in the generation of the switches to check if all modules necessary exist because Switch (De)Composer needs to resolve all the annotations, for each module in the switch. For this purpose, we created a function to sort and get the dependencies by searching the tree to see if all modules listed exist and adding missing modules in order to make a connected graph, since it is not connected one of the dependencies will not be properly included in the code. Listing 5.2 presents an example of a dependencies JSON file with 3 modules that have the same configuration as in Figure 5.3.

Listing 5.2 – Dependencies JSON example

```
[
    {
        "name": "ethernet",
        "file": "obs_main.up4",
        "function": "ethernet",
        "directDependencies": ["ipv4", "ipv6"],
        "regex": [
            "ethernet",
            "([0-9A-Fa-f]{2}[:-]){5}([0-9A-Fa-f]{2})"
        ],
        "head":true
    },
    {
        "name": "ipv4",
        "file": "ipv4.up4",
        "function": "ipv4",
        "directDependencies": [],
        "regex": ["ipv4", ...]
    },
    {
        "name": "ipv6",
        "file": "ipv6.up4",
        "function": "ipv6",
        "directDependencies": [],
        "regex": ["ipv6", ...]
    }
]
```

As a way to let different switches on the topology have completely different codes, we decide to allow each switch to have a different dependencies graph. To do that in the topology JSON you need to define the topology file name as a switch parameter. Then if the network developers want to use their own modules and dependency graph, they can do it by putting their code and files in the respective folders on the project root, *modules* and *dependencies-json*, or by passing a customizable folder as a solution parameter.

A new piece of code implemented was the constraints interpreter that analyses policy documents, like Open Flow, to identify which modules should be included on a given switch. For this, we created a function that uses regular expressions to match patterns on the policy file with modules on the dependencies graph. To allow users to add new expressions according to their policies, we added a property *regex* array that can be passed to each switch object to the dependencies JSON, where they can add new regex. We can see how to pass the regex on Listing 5.2, although some of them were ellipsed since their content has large and would take too much space.

This modification is really important because it automates the switches generation, based on predefined policies, which can save a lot of time since the developer does not need to remember all necessary models. In order to use this interpreter, instead of directly listing the modules in the topology JSON, we added a new property *policies*, which represents the name of the policy file used to extract the necessary modules to include.

The policies files must be added to a folder named *policies* on the root of the repository, or passed as a customized parameter of the Switch (De)Composer++. On the Listing 5.1 we can see the second switch, named *s2*, uses a police file to get the dependencies, while the other two use the older method of listing the modules.

Another challenge was related to changing the forwarding tables, the proof-of-concept had a fixed topology so all the forwarding tables used these constant IPs and MACs values to forward the packet. So we needed to find a way to replace this part of the code according to the switch topology. The way we find to solve this problem was to create another annotation @TableInstantiate() that must surround the table, then a new table is mounted using a function that receives topology data and then send it to replace the original table code. These functions are placed in a new python file called *specific_functions.py* so the user can change them or add new ones as well. In order for Switch (De)Composer++ to know which function is used to generate a table, the function name has to be declared in the dependencies JSON.

A major flaw related to Switch (De)Composer usability, was that in order for it to work we needed to copy the repository into the folder *extensions/csa/* of a frozen version of the obs-microp4 repository[3] which contains the μP4 source code. Besides that, the user still had to run multiple configuration commands to set environment variables before being able to successfully use Switch (De)Composer. So what we decided to do was to invert that logic and have the copy of the frozen repository as a git submodule of the Switch (De)Composer++ repository, this was one of the hardest challenges since we needed to configure the git submodule correctly and set the path to all environment variable according to project path.

This change was made so that the user doesn't need to worry about doing multiple configurations before even running our solution. This was a successful modification because following it the user only needs to download our repository and run a few scripts we left ready to run on the help folder before they are ready to generate P4 switches. Although these improvements deeply facilitated the process, we don't eliminate all requisites from Switch (De)Composer++, anyone that wants to use our solution still needs to have the P4 language and Python 3 language already installed on the machine. Our solution may also not run on all Operational Systems, since we only tested so far on Ubuntu 18.04 because that is the version μP4 recommends [4].

We also decided to make our program more customizable to better fit the necessi-

[3]https://github.com/pauladbol/obs-microp4 (BOL et al., 2021b)
[4]https://github.com/cornell-netlab/MicroP4#11-dependencies (SONI et al., 2020a)

ties of all network developers, for this purpose we made a series of optional parameters to customize Switch (De)Composer++. The first one is the option to insert the output path for the switches, so it could be saved where the user wants. If this parameter is not passed, the output P4 switch programs will be placed inside a Switch (De)Composer++ folder appropriately called *outputs*. Other parameters related to paths on the operating system are the location of the folder that has the dependencies JSON (*dependencies-folder-path*) and the folder that has the policies (*policies-folder-path*), already mentioned before.

Another optional parameter, called *not-run-mininet*, disables the automatic run of Mininet, as shown in Figure 5.2 this step is the second last on our workflow, however, we understand it may not be necessary for all scenarios, so it was added a flag to disable the default run of Mininet to test the P4 modules generated. We also included the option *separate-mininet* to put the Mininet script in a separate Shell file, so it could be run apart from the generation of the switches. It was also defined that the Shell script, which gets all the modules and does the compilation, would be automatically run after its creation, therefore only a single command will be necessary to generate the switches, so a flag *not-auto-run* was added to disabled this automation.

A minor improvement done to the project was a refactor on the main code from Switch (De)Composer, the *generate_switch_program.py*, to make it able to run on Python 3 instead of Python 2, since that version was discontinued and will not receive further updates. There were some additional refactors that remove a class definition and some specialized functions from inside the main file to their own files and move all Python source code to a *src* folder to improve code readability, reusability, and organization.

5.3 Switch (De)Composer++ Usage Process

So far it was discussed how the solution works and the improvements made to its usability, but to finish let's paint a full picture of the process that a network developer needed to use Switch (De)Composer++, from start to finish. First, the developer needs to have a set μP4 modules, with their OBS switch and all the necessary annotations, the process of adding the annotations can take a while but it would only need to be done once. If this is a customized set of μP4 modules used the user also needs to provide their own dependency JSON file, specifying all dependencies between modules. Or the developer could use the modules we already provide, which so far only include IPv4, IPv6, and the main OBS code that implements ethernet, therefore also using our dependency graph.

After adding the modules and dependencies graph, the final input they need to have before using our solution is a JSON file with the desired topology wanted for the network. For each switch in the topology, the network developer can manually add the modules they want to include or link to a policies document that will be automatically interpreted to get the module's dependencies. With all this they can finally run our tool, opting to add the optional parameters, such as output path, dependencies folder path, policies folder path, disable Mininet, separate Mininet, and automatically run the generation Shell script.

If the option was chosen to allow the automatically run of the generation and compilation script, it already will be executed inside the chosen output folder. In the opposite case, the developer can manually run the Shell script, which will be placed inside the chosen output folder after its generation. Then the script will execute following the steps from Figure 5.2, in the case a policies file was added to a switch in the topology JSON the interpreter will be run inside on the first step that generates the switches. Finally, after the Shell script is finished, for each switch in the topology there will be a P4 program with the modular switch code ready to be deployed.

Something that is also very important to notice is that if the Mininet step is enabled, it will take over the terminal. This will allow the user to run Mininet commands and make their test on the fly. When they are satisfied, is possible to finish the Mininet by typing *exit* or pressing *ctrl + D*, and the script will do the final cleaning step. If the terminal is closed during the Mininet step nothing will be lost, all P4 switches will be available, and the only part that will not be done is cleaning the intermediate files. If the option to run the separate Mininet is selected, it will create a new file named *mininet.sh* with the last three steps represented in Figure 5.2.

Another important usage scenario to consider is when a network of switches was previously generated by Switch (De)Composer++ and is necessary to make modifications to a few of the switches. In this case, the best thing will be to keep the original inputs given to Switch (De)Composer++, and then run it again making the required changes to the switches on to the topology and policies files. The code generated for the unchanged switches the second time should be the same, since Switch (De)Composer++ is deterministic. Because of that tests and simulations can be run on top of the generated switches to check the behavior of the topology after the changes. At the end of the process, the network developer can take only the P4 programs generated for the modified switches and deploy them individually instead of needing to make changes to the whole network.

5.4 Quantitative Analysis

As a quantitative analysis of the software improvements, the original Switch (De) Composer repository[5] contained around 1400 Lines-of-Code in total this included the Python automation code, the μP4 modules, and also some example and git files. Considering only the Python used for switch generation, the proof-of-concept had approximately 320 LoC. Our complete repository [6] has 2400 LoC including python, Shell script the modules, topologies, dependencies graph, helpers, and git configurations files. Considering only the Python programs to automate the generations the newer version has approximately 610 LoC, which is almost twice the size of the original code.

Using the *git diff* command we gather some additional data regarding the changes, there were 45 files changed, among those 28 new files were added and only two were deleted. About lines of code, git shows us there were 1103 code line insertions and only 162 line deletions. There are also 8 new folders added to the root repository, 9 when counting the *obs-microp4* dependency, which helps with the organization.

Running the *git diff* with a filter for only the python files specifically does not show useful statistics, because the two original Python files were renamed, so the git considers them different files counting the lines of code of the original files as deletion and the new ones as insertions. But it does shows there are 6 newer files, including the *generate_distribute_programs.py* which represents almost a third of the Python code. Analyzing all these quantitative results, we could build on top of the existing solution by adding important automation to enhance the software significantly.

[5]https://github.com/pauladbol/SwitchDeComposer (BOL, 2021)
[6]https://github.com/JuDCraide/Switch-De-Composer- (CRAIDE, 2023)

6 EXPERIMENTAL EVALUATION

A crucial part of proposing any new tool, especially the ones related to networking, is to test it in realistic situations, so this Chapter exposes the testing efforts done. To test our solution on the available hardware, a few modifications to the output switches P4 code needed to be done, in Section 6.1 those are gonna be explained. Then the obtained results and calculations will be presented in Section 6.2

6.1 Test Modifications

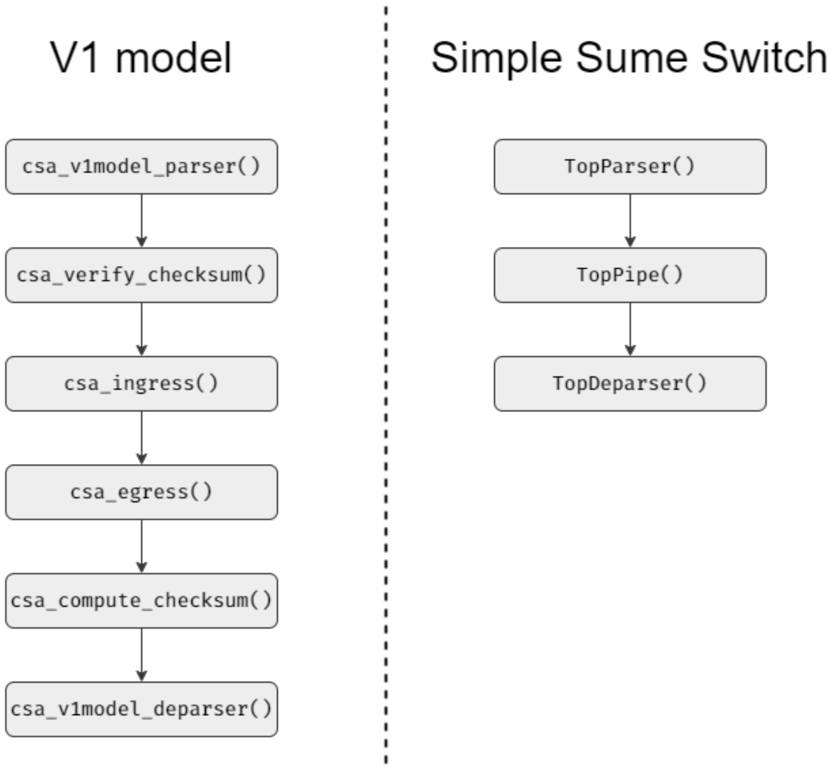
Switch (De)Composer generates switches that follow the V1Model, so since we did not have a Tofino switch to make our tests, it was needed to make adaptations to the generated P4 code in order to test it on the board we had, a NetFPGA-SUME that work with the Simple Sume Switch architecture. A comparison between the V1Model and the Simple Sume Switch code is shown in Figure 6.1, we can see that the V1Model has 6 basic P4 control functions that execute in order, while the Simple Sume Switch only has 3. The main difference in the code is that Simple Sume Switch doesn't have the two checksum functions in the packet way in and out, the ingress and egress, all those functions are united in a single TopPipe control.

So in order to make the P4 output from Switch (De)Composer++ run in the board, changes were needed to adapt it to the Simple Sume Switch model. The first change we encounter was that SUME doesn't accept arrays of structs, in our V1Model P4 we had a *msa_packet_struct_t* struct that has an array of *msa_byte_h* structs as an attribute called *msa_hdr_stack*. So instead of a single array we needed to have n attributes, like *msa_hdr_stack_0*, *msa_hdr_stack_1*, . . . , *msa_hdr_stack_n*, with *n* being the array size, which varies between switches that have IPv4 module or IPv6 module, because of the change of the size of the packet header.

Another important change was that the main struct *standard_metadata_t* needed to be replaced. The problem was that this metadata struct was defined on the *v1model.p4* package, so is not compatible with the SUME metadata, so we need to change it to the appropriate metadata for the board. The fix for this problem was simple, we replaced references to it with the official SUME metadata struct, the *sume_metadata_t*.

The P4 V1Model annotation call *@name(".NoAction")*, also needed to be removed, since its name wasn't accepted by the Simple Sume Switch architecture. We

Figure 6.1 – V1Model vs Simple Sume Switch



Source: The Author

initially believed the problem was that the annotation name starts with a dot symbol, which we believe confused the compiler. Searching about it on the official P4 NetFPGA repository wiki [1], we realized Simple Sume Switch model doesn't support *name* annotations at all, so we went ahead and remove all annotations of this type.

The P4 match_kind *ternary* operator didn't work on NetFPGA SUME, we could not exactly pinpoint why, thankfully in our match action every time the code uses the ternary operator all the table entries had the same default value. The word default (also equivalent to _) in this context represents a *don't care*, which means this value is not used to choose between entries. So we could change the match_kind to an *exact* since an *exact* comparison with a *don't care* accept the universal set of values.

Specifically, on the switches that implemented the IPv6 module, there were changes needed because the V1Model struct for IPv6 header uses the word *class* as the name of an attribute. On the other hand SDNET compiler, used to compile the Simple SUME Switch, has class as a reserved word. So to solve the problem we needed to replace the *IPv6_h* struct attribute name, and this was the last modification that needed to be made.

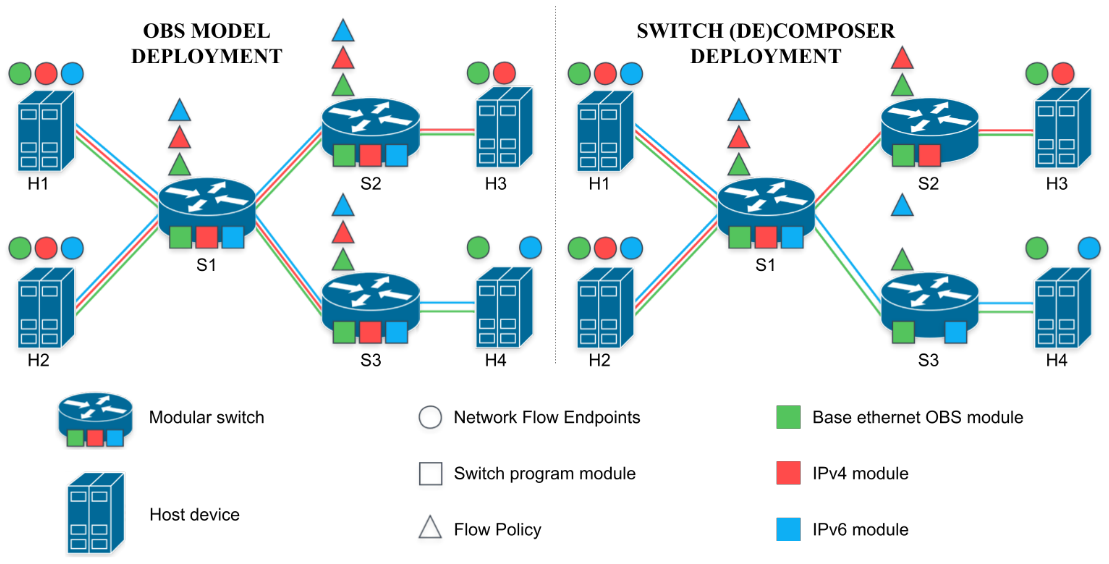
[1]https://github.com/NetFPGA/P4-NetFPGA-public/wiki/Workflow-Overview#annotations (IBANEZ; ZILBERMAN, 2018)

Lastly, the entries from all match-action tables needed to be moved to a secondary *commands.txt* file since this is the way architecture handles this workflow[2]. The table entries format also had some changes, but every entry line in the V1Model became another line in the commands, so we do not believe this would have much impact on the metrics. One more small change related to the match-action is that on the command entries, it does not accept boolean type, so the boolean logic was converted to numeric 0 and 1. So after all these changes to the switches P4 programs, we were able to compile all of them successfully and start the steps for the practical evaluation.

6.2 Practical Evaluation

In this evaluation, we focused on a test case that corresponds to scenario 1 from the original Switch (De)Composer(BOL et al., 2021a) preliminary evaluation as shown in Chapter 4. Figure 6.2 shows the topology used by our tests, which is a specialization of the one shown 1.1. This is a simple topology with three switches connected in a V-shape, with only two constraint policies, IPv4 and IPv6. Switch 1 (S1) needs to support both IPv4 and IPv6 protocols and is connected to two hosts Host 1 (H1) and Host 2 (H2). Switch 2 (S2) needs to support only IPv4 and is connected to S1 and Host 3 (H3). Finally, Switch 3 (S3) needs to support only IPv6 and is connected to S1 and Host 4 (H4).

Figure 6.2 – OBS vs. Switch (De)Composer detailed deployment



Source: The Author, adapted from (BOL et al., 2021a)

[2]https://github.com/NetFPGA/P4-NetFPGA-public/wiki/Workflow-Overview#workflow-steps (IBANEZ; ZILBERMAN, 2018)

We are going to compare two equivalent solutions regarding latency and occupation, in the OBS model deployment, all switches have the full OBS code with both protocols. On the other hand, in the Switch (De)Composer deployment, S1 has the full OBS code, S2 has the base OBS code plus the IPv4 module, and finally, S3 has the base OBS code plus the IPv6 module. To simplify the nomenclature for the following tables we are going to be using the following nomenclature: *IPv4 Switch* equivalent to S2 switch from the Switch (De)Composer deployment, *IPv6 Switch* equivalent to S3 from the Switch (De)Composer deployment, and *OBS Switch* equivalent to S1 from the Switch (De)Composer and S1, S2, and S3 from the OBS deployment.

In order to do the test for our evaluation, we need to connect to a computer with a Xilinx license server to authenticate the SDNet Compiler and Vivado tools. There was a script to test if the license server is properly working, that needed to be run before anything else, because if the license failed it may lead to many errors. Something important to notice is that all commands used to run the practical tests must be executed as root user (su -), otherwise, they will not execute correctly. This is because all the software needed was installed in the root, so we cannot run as a normal user.

P4VBox (SAQUETTI et al., 2020) provides a CLI to run all steps needed between compiling P4 and generating the bitstream. In this test workflow, there are 5 stages, the first one compiles the P4 code, the next one generates the testbench, the third simulates the switch behavior according to the testbench, the fourth one does the bitstream implementation, and the last one is to open the implementation in Vivado. Steps one, two, and four are done exclusively in the command line, step three can be done both by the terminal or on Vivado graphical interface, and step five is done on the Vivado graphical interface.

Before doing any step all the changes mentioned in the previous Chapter were done to each switch program. The testbench used to test the switches was based on the P4VBox template, first, we define packets to be sent to the board entry ports, then we compare the switch output to the expected headers, payload, and exit port. There were actually two different versions of the testbench, one that sends IPv6 packets and the other IPv4 packets. In the testbench results, the latency can be seen by analyzing the waveforms from the signals that indicate the packet was entering (t0) or exiting (t1) the board.

Based on the measured in-out times, gather from the testbench analysis, we calculated the packet round trip time by doubling the difference between t1 and t0. The clock used by the testbench simulation was 5ns, with that we can compute the clock cycle by dividing the round trip by the clock. Then finally to calculate the switch latency, instead of the actually used clock we use the minimum clock allowed by the SUME NetFPGA, that according to (SAQUETTI et al., 2020) is 1,826 ns as the critical path occupies the fixed parts of the FPGA architecture, then we multiply that value with the previously obtain clock cycle. Table 6.1 shows the result of all these time measures for each switch in our topology. It is important to notice that the *OBS Switch*, was tested twice, one for each protocol testbench, but the latency was the same between the tests.

Table 6.1 – Switches Time measurements

Switch	t0 (ns)	t1 (ns)	Round Trip (ns)	Clock Cycles	Latency (ns)
IPv4 Switch	26816.5	28300	2967	593.4	1083.5484
IPv6 Switch	26816.5	28492	3351	670.2	1223.7852
OBS Switch	26816.5	28956	4279	855.8	1562.6908

In the final step, we can see all the reports generated from the implementation. The most important to our study is the occupation report. Table 6.2 shows the most important occupancy measures, the sliced LUTs, and sliced registers occupancy. These two metrics are so essential because commercial FPGAs, such as the ones made by Xilinx and Altera, use LUTs and Flip-flops to compose the Basic Logic Element (BLE) (FAROOQ; MARRAKCHI; MEHREZ, 2012). And those units are then used on the FPGAs configurable logic blocks, which can consist of a cluster of BLEs or a single BLE to provide the logic and storage functionalities.

Table 6.2 – Switches Occupation

Switch	Slice LUTs	Slice Register
IPv4 switch	67436	157634
IPv6 switch	77766	204837
OBS switch	109063	378567

After gathering the latency and occupation data for each of the generated P4 programs, Table 6.3 shows a comparison between the two solutions. OBS solution deployment is the one where all three switches use the full OBS code (*OBS Switch*). In contrast, Switch (De)Composer generates the 3 switches with only modules needed (one *OBS Switch*, one *IPv4 Switch*, and one *IPv6 Switch*) for its deployment.

In order to obtain occupancy measurements shown for the Switch (De)Composer solution, we add one of each switches occupancy from Table 6.2, and, for the OBS solution, we multiply by three the *OBS Switch* from the same table. For the latency we measure two scenarios the first one goes from Host 1 to Host 3 (H1 to H3), therefore

Table 6.3 – Switches Deployment Latency and Occupation

	Occupation		**Latency (ns)**	
	Slice LUTs	Slice Register	H1 to H3	H1 to H4
OBS	327189	1135701	3125.3816	3125.3816
Switch (De)Composer	254265	741038	2646.2392	2786.4760

passing through S1 and S2, and the second goes from Host 1 to Host 4 (H1 to H4), therefore passing through S1 and S3. For these latency scenarios, we only sum the latency of the switches in the flow path from Table 6.1, consequently disregarding the latency of the network cables. A better way of measuring the latency between hosts would be to implement physically all switches the topology on the NetFPGA SUME, we do intend to test this, but could not finish this experiment on time for the deadlines.

6.3 Results discussion

The results obtained from the practical evaluation were very positive, regarding the latency shown in Table 6.1 the *IPv4 Switch* was around 31% lower than the *OBS Switch* on the IPv4 testbench and the *IPv6 Switch* was 22% lower than *OBS Switch* on the IPv6 testbench. This is also aligned with the occupancy results shown in Table 6.2 since, compared to the *OBS Switch*, *IPv4 Switch* occupied around 31% fewer LUTs and 50% fewer registers, regarding *IPv6 Switch* it occupied around 22% fewer LUTs and 37% fewer registers. This was expected since there is a correlation between the robustness of the switch code, the occupation of the FPGA, and the increase in latency.

In Table 6.3 we compare the switches generated from a trivial OBS deployment with our solution based on a topology describe in Figure 6.2. Regarding the latency Switch (De)Composer was 15% lower at sending IPv4 packets through two switches (one *OBS Switch* and one *IPv4 Switch*) and 11% lower sending IPv6 packets by two switches (one *OBS Switch* and one *IPv6 Switch*), compared to the OBS deployment. Our solution had significant improvements in the FPGA occupation over the OBS, using 18% fewer LUTs and 29% fewer registers. Those results are comparable to the occupancy approximation made on the proof-of-concept (BOL et al., 2021a) by counting the switch tables, there the equivalent scenario (scenario 1) had a 26% reduction in the occupancy using Switch (De)Composer, which is close to the 18% reduction in LUTs and 29% reduction in registers obtained practically on the NetFPGA SUME.

7 FINAL CONSIDERATIONS AND OUTLOOK

This work addresses the problem of generating modular switch codes for multiple switches in a topology following the given constraints while taking advantage of the OBS abstraction. Our main objectives were to generate programs that could be reusable, easily extended, and not tightly coupled due to the OBS model. Another important metric we aimed to achieve a more efficient resource usage and lower latency, due to the reduced number of modules our solution produces compared to the trivial OBS implementation, section 7.1 will discuss in depth these metrics and results. Lastly, Section 7.2 will present ideas for future iterations of this work.

7.1 Considerations on Results Achieved

We present significant changes to the Switch (De)Composer architecture, most importantly the automation of the switch generation and the compilation process that used to be manually made. Due to this new process, we also created a new way of inputting the topology requirements via JSON file, as before the user needed to manually generate each switch in the console by passing the modules and constraints. The way we handle the OBS module and submodules was completely changed, now all modules must be described in a dependency JSON file to generate the dependency graph and we made it possible for switches in the same topology to be based on different dependency graphs.

A different automation issue tackled was how Switch (De)Composer gets the necessary modules to include in a switch, so we create a constraints interpreter that can decide which module to include by analyzing Open Flow policies. We also implemented a solution to allow network developers to customize the forwarding tables for each switch program based on the topology by adding an annotation to the code. Finally, there were several improvements made to the repository and the code itself using best practices, and all these changes overall made Switch (De)Composer++ easier to use and customize.

A quantitative analysis was also made comparing the code of Switch (De)Composer with Switch (De)Composer++, which shows that approximately 1000 lines were added to the repository. From those around 600 LoC were Python programs created for the automatization efforts, this show how the solution was extended, while still leveraging the functionalities from the existing code. Therefore, on the software side, we deliver a complete solution that can generate multiple switches from a topology, based on policies constraint, a dependencies graph, and the given OBS μP4 modules.

Our conclusions from the practical evaluation were that using the modular switches generated by Switch (De)Composer++ there were relevant reductions both in latency and occupation in all the tests we made. We saw greater improvements comparing just among two switches (Table 6.1 and Table 6.2), one with a single module and the other with the full OBS, than comparing our entire topologies scenario (Table 6.3). This is explained since in an average topology there will probably be some switches with all modules and some switches with fewer modules, so the topology is a limiting factor to latency and occupation improvements. As a hypothetical example, a topology with 10 switches generated by Switch (De)Composer where nine are *IPv4 Switch* and one is *OBS Switch* will have better metrics than a topology with nine *OBS Switch* and one *IPv4 Switch*.

In our tests, the IPv6 module was the most complex of the network policy modules, so when we compare switches that include only IPv6 (*IPv6 Switch*) with switches that only include IPv4 (*IPv4 Switch*), the *IPv6 Switch* is slightly worst in both latency and occupancy. We also believe that the impact could be even greater with more modules in the topology since the gains will not be so limited by the larger module as it happened with IPv6. We also acknowledge that there were many changes needed to be able to test our output modular switches on the NetFPGA-SUME board, which could have an impact on the result obtained, but we tried to do only the necessary changes to interfere with the least possible and get reasonable results.

7.2 Future Work

Despite the advancements to Switch (De)Composer solution in this project, there are still many opportunities for future work to be developed on top of this solution. Firstly due to time constraints, we could only test one scenario, so, in the future, we intend to test more robust scenarios to improve our results. Also due to a lack of equipment, we could not test the switches in the Intel Tofino, they were originally designed for.

A few promises made on the original Switch (De)Composer article (BOL et al., 2021a) have yet to be developed. The most important one is to add more modules and constraints, including constraints of different types, as an example resources available, or continuously changing contains, which would require changing the modules on the fly. And although we did run some basic testbenches to test if the Switch (De)Composer++ switches programs had the correct behavior, they were far from complete. So there is still a great effort ahead to create a verification strategy that can ensure all behaviors are equivalent between the custom modular switches and the initial OBS program.

Lastly, our idea for the work made to the project is to write a paper for publication at a symposium, like the ACM SIGCOMM Symposium on SDN Research (SOSR). We also want to make a full analysis of the topology in terms of latency using a real testbed with the switches implemented on NetFPGA SUME. Finally, we would like to compare the same test on a Tofino, since the academic laboratory this project is part of just received a new Barefoot Tofino.

REFERENCES

ARASHLOO, M. T. et al. Snap: Stateful network-wide abstractions for packet processing. In: **Proceedings of the 2016 ACM SIGCOMM Conference**. New York, NY, USA: Association for Computing Machinery, 2016. (SIGCOMM '16), p. 29–43. ISBN 9781450341936. Available from Internet: <https://doi.org/10.1145/2934872.2934892>.

BENZEKKI, K.; FERGOUGUI, A. E.; ELALAOUI, A. E. Software-defined networking (sdn): a survey. **Security and Communication Networks**, v. 9, n. 18, p. 5803–5833, 2016. Available from Internet: <https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.1737>.

BOL, P. D. **SwitchDeComposer**. 2021. Accessed: March 31, 2023. Available from Internet: <https://github.com/pauladbol/SwitchDeComposer>.

BOL, P. D. et al. Modular switch deployment in programmable forwarding planes with switch (de)composer. In: **Proceedings of the SIGCOMM '21 Poster and Demo Sessions**. New York, NY, USA: Association for Computing Machinery, 2021. (SIGCOMM '21), p. 30–32. ISBN 9781450386296. Available from Internet: <https://doi.org/10.1145/3472716.3472856>.

BOL, P. D. et al. **obs-microp4**. 2021. Accessed: March 31, 2023. Available from Internet: <https://github.com/pauladbol/obs-microp4>.

BOSSHART, P. et al. P4: Programming protocol-independent packet processors. Association for Computing Machinery, New York, NY, USA, v. 44, n. 3, p. 87–95, jul 2014. ISSN 0146-4833. Available from Internet: <https://doi.org/10.1145/2656877.2656890>.

BYTE, N. **Intel demonstra o primeiro switch Ethernet óptico co-empacotado do setor (English Only)**. 2017. Accessed: March 31, 2023. Available from Internet: <https://newsroom.intel.com.br/news/intel-demonstra-o-primeiro-switch-ethernet-optico-co-empacotado-do-setor-english-only/#gs.ssbo4h>.

CHEN, X. et al. Speed: Resource-efficient and high-performance deployment for data plane programs. In: **2020 IEEE 28th International Conference on Network Protocols (ICNP)**. [S.l.: s.n.], 2020. p. 1–12.

CONTRIBUTORS, M. P. (Ed.). **Mininet Overview**. 2022. Accessed: March 31, 2023. Available from Internet: <http://mininet.org/overview/>.

CRAIDE, J. D. **Switch-De-Composer++**. 2023. Accessed: April 16, 2023. Available from Internet: <https://github.com/JuDCraide/Switch-De-Composer->.

FAROOQ, U.; MARRAKCHI, Z.; MEHREZ, H. Fpga architectures: An overview. In: ____. **Tree-based Heterogeneous FPGA Architectures: Application Specific Exploration and Optimization**. New York, NY: Springer New York, 2012. p. 7–48. ISBN 978-1-4614-3594-5. Available from Internet: <https://doi.org/10.1007/978-1-4614-3594-5_2>.

GAO, J. et al. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In: . New York, NY, USA: Association for Computing Machinery, 2020. (SIGCOMM '20), p. 435–450. ISBN 9781450379557. Available from Internet: <https://doi.org/10.1145/3387514.3405879>.

GOBATTO, L. et al. Programmable data planes meets in-network computing: State of the art, challenges, and research directions. **Journal of Integrated Circuits and Systems**, v. 16, n. 2, p. 1–8, 2021.

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.

IBANEZ, S.; ZILBERMAN, N. **P4-NetFPGA-public**. 2018. Accessed: March 31, 2023. Available from Internet: <https://github.com/NetFPGA/P4-NetFPGA-public>.

KANG, N. et al. Optimizing the "one big switch" abstraction in software-defined networks. In: **Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies**. New York, NY, USA: Association for Computing Machinery, 2013. (CoNEXT '13), p. 13–24. ISBN 9781450321013. Available from Internet: <https://doi.org/10.1145/2535372.2535373>.

KAUR, K.; SINGH, J.; GHUMMAN, N. Mininet as software defined networking testing platform. In: **International Conference on COMMUNICATION, COMPUTING & SYSTEMS (ICCCS–2014)**. [S.l.: s.n.], 2014. p. 139–142.

KUON, I.; ROSE, J. Measuring the gap between fpgas and asics. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 26, n. 2, p. 203–215, 2007.

KUON, I.; TESSIER, R.; ROSE, J. Fpga architecture: Survey and challenges. **Foundations and Trends® in Electronic Design Automation**, v. 2, n. 2, p. 135–253, 2008. ISSN 1551-3939. Available from Internet: <http://dx.doi.org/10.1561/1000000005>.

LARA, A.; KOLASANI, A.; RAMAMURTHY, B. Network innovation using openflow: A survey. **IEEE Communications Surveys & Tutorials**, v. 16, n. 1, p. 493–512, 2014.

MCKEOWN, N. et al. Openflow: Enabling innovation in campus networks. **SIGCOMM Comput. Commun. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 38, n. 2, p. 69–74, mar 2008. ISSN 0146-4833. Available from Internet: <https://doi.org/10.1145/1355734.1355746>.

NGUYEN, X.-N. et al. Rules placement problem in openflow networks: A survey. **IEEE Communications Surveys & Tutorials**, v. 18, n. 2, p. 1273–1286, 2016.

OLIVEIRA, R. L. S. de 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.

P4LANGUAGE (Ed.). **Behavioral Model (bmv2)**: The bmv2 simple switch target. 2022. Accessed: March 31, 2023. Available from Internet: <https://github.com/p4lang/behavioral-model/blob/main/docs/simple_switch.md>.

P4.ORG (Ed.). **P4 Language Tutorial**: P4 developer day fall 2017. 2017. Accessed: March 31, 2023. Available from Internet: <https://opennetworking.org/wp-content/uploads/2020/12/P4_tutorial_01_basics.gslide.pdf>.

SAQUETTI, M. et al. Hard virtualization of p4-based switches with virtp4. In: **Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos**. New York, NY, USA: Association for Computing Machinery, 2019. (SIGCOMM Posters and Demos '19), p. 80–81. ISBN 9781450368865. Available from Internet: <https://doi.org/10.1145/3342280.3342314>.

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

SONI, H. et al. **MicroP4**. 2020. Accessed: March 31, 2023. Available from Internet: <https://github.com/cornell-netlab/MicroP4>.

SONI, H. et al. Composing dataplane programs with μp4. In: . 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>.

SULTANA, N. et al. Flightplan: Dataplane disaggregation and placement for p4 programs. In: **18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)**. USENIX Association, 2021. p. 571–592. ISBN 978-1-939133-21-2. Available from Internet: <https://www.usenix.org/conference/nsdi21/presentation/sultana>.

WINTERMEYER, P. et al. P2go: P4 profile-guided optimizations. In: **Proceedings of the 19th ACM Workshop on Hot Topics in Networks**. New York, NY, USA: Association for Computing Machinery, 2020. (HotNets '20), p. 146–152. ISBN 9781450381451. Available from Internet: <https://doi.org/10.1145/3422604.3425941>.

YANG, H. et al. Review of advanced fpga architectures and technologies. **Journal of Electronics**, v. 31, p. 371–393, 10 2014.

ZILBERMAN, N. et al. Netfpga sume: Toward 100 gbps as research commodity. **IEEE Micro**, v. 34, n. 5, p. 32–41, 2014.