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

MATEUS SAQUETTI PEREIRA DE CARVALHO
TIRONE

Programmable virtual Switches: Design and Implementation of the Forwarding Engine and Supporting Features

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Microeletronics

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

Porto Alegre
May 2020

CIP — CATALOGING-IN-PUBLICATION

Tirone, Mateus Saquetti Pereira de Carvalho

Programmable virtual Switches: Design and Implementation of the Forwarding Engine and Supporting Features / Mateus Saquetti Pereira de Carvalho Tirone. – Porto Alegre: PGMICRO da UFRGS, 2020.

87 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica, Porto Alegre, BR–RS, 2020. Advisor: José Rodrigo Azambuja; Coadvisor: Weverton Cordeiro.

1. Programmable forwarding planes. 2. P4. 3. Virtualization. 4. NetFPGA SUME. I. Azambuja, José Rodrigo. II. Cordeiro, Weverton. III. Programmable virtual Switches: Designand Implementation of the ForwardingEngine and Supporting Features

“If I have seen farther than others,
it is because I stood on the shoulders of giants.”

— Sir Isaac Newton

ACKNOWLEDGMENT

I would like to thanks my parents, Maria and Luis, that have always taken care of me. I am grateful for all support and learning that allowed me to be who I am. Thank you for believing in me even when I thought that I was not capable. Thank you for all unconditional love.

Mônica, thank you for all love and support, mainly on the hard days. Thank you for being my safe harbor, for being at my side and do not give up on me. You are so much more than a wife. You are my love, my partner, and my best friend, and I am so thankful for that. Thank you for making me so happy. I love you.

Many thanks to all my friends and family that even far supported me.

Finally, I would like to express my special thanks for gratitude to my advisors José and Weverton who allowed me to work on this project. Thank you for trusting in me and for all your dedication and guidance. You are an inspiration for me.

ABSTRACT

Virtualization has become the powerhouse for several networking concepts, from virtual local area networks (VLANs) to software switches, software-defined control plane, etc. Recent proposals like HyPer4, HyperVDP, and P4Visor brought the concept to the forwarding plane, by enabling *emulation* of several network contexts and/or composing several functions through a single program. In spite of the progress achieved, the real potential of forwarding plane virtualization remains untapped. In this work, we present PvS, a reconfigurable architecture for data plane virtualization. PvS provides parallel execution and hot-swapping of virtual switch instances, without requiring switch source code (for either emulation or program composition). We experimented PvS on a NetFPGA-SUME board with four virtual switches: a layer-2 switch, a simple router, a firewall, and an in-band telemetry. Area occupation measurements evidence the feasibility of running up to eight virtual switches in parallel. Compared to existing work, performance data show an improvement of up to two orders of magnitude for bandwidth and six orders for latency.

Keywords: Programmable forwarding planes. P4. virtualization. NetFPGA SUME.

Switches virtuais Programáveis: Projeto e Implementação do Mecanismo de Encaminhamento e Recursos de Suporte

RESUMO

A virtualização se tornou a potência de vários conceitos de rede, de redes locais virtuais (VLANs) a switches definidos por software, plano de controle definido por software, etc. Propostas recentes como HyPer4, HyperVDP e P4Visor levaram o conceito ao plano de encaminhamento, permitindo *emulação* de vários contextos de rede e/ou compondo várias funções através de um único programa. Apesar do progresso alcançado, o verdadeiro potencial da virtualização do plano de encaminhamento permanece inexplorado. Neste trabalho, apresentamos PvS, uma arquitetura reconfigurável para virtualização do plano de dados. PvS fornece execução paralela e hot swap de instâncias de switches virtuais, sem a necessidade de código-fonte do switch (para emulação ou composição de programa). Experimentou-se PvS em uma placa NetFPGA-SUME com quatro switches virtuais: um switch de camada 2, um roteador simples, um firewall e um switch de telemetria de rede(*in-band telemetry*). As medições de ocupação da área evidenciam a viabilidade de executar até treze switches virtuais em paralelo. Comparado aos trabalhos existentes, os dados de desempenho mostram uma melhoria de até duas ordens de magnitude para largura de banda e seis ordens para latência.

Palavras-chave: plano de encaminhamento programável, P4, virtualização, NetFPGA SUME.

LIST OF ABBREVIATIONS AND ACRONYMS

ASIC	Application Specific Integrated Circuit
BEL	Basic Element of Logic
BMv2	Behavioral Model version 2
BOQ	BRAM Output Queues
BRAM	Block Random Access Memory
CAM	Content Addressable Memory
CDPI	Control-Data-Plane Interface
CLB	Configurable Logic Blocks
CLI	Command Line Interface
CPLD	Complex Programmable Logic Device
CvSI	Control virtual Switch Interface
DMA	Direct Memory Access
DSL	Domain-Specific Language
FPGA	Field-Programmable Gate Array
GTH	Gigabyte Transceiver
HDL	Hardware Description Language
HLIR	High-Level Intermediate Representation
IA	Input Arbiter
IP	Intellectual Property
IvSI	Input virtual Switch Interface
JTAG	Joint Test Action Group
LAN	Local Area Network
LUT	Look-Up Table
NFV	Network Function Virtualization

ONF	Open Networking Foundation
OPL	Output Port Lookup
OvSI	Output virtual Switch Interface
P4	Programming Protocol-Independent Packet Processors
PCIe	Peripheral Component Interconnect Express
POF	Protocol-Oblivious Forwarding
PvS	Programmable virtual Switches
PR	Partial Reconfiguration
QSH	High-Speed Socket
QTH	High-Speed Terminal
RDMA	Remote Direct Memory Access
RISC	Reduced Instruction Set Computer
RMT	Reconfigurable Match-action Tables
RM	Reconfigurable Modules
SFP	Small Form Factor Pluggable
SSS	Simple Sume Switch
TCAM	Ternary Content Addressable Memory
VI	Virtual Internet
VSS	Very Simple Switch
VLAN	Virtual Local Area Network
VXLAN	Virtual Extensible LAN
vS	virtual Switch
WAN	Wide Area Networks

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

The concept of programmable forwarding planes has seen a renewed interest by industry and academia with the advent of Domain-Specific Languages (DSL) like POF (SONG, 2013a) and P4 (BOSSHART et al., 2014). These languages unleashed innovation in the data plane (CORDEIRO; MARQUES; GASPARY, 2017), helping break further the "network ossification".

Resource sharing through virtualization has become commonplace in modern public and private clouds, powering virtual abstractions for servers (KOPONEN et al., 2014), network functions (MARTINS et al., 2014; HWANG; RAMAKRISHNAN; WOOD, 2015; LI et al., 2016; YI et al., 2018), forwarding devices (DOBRESCU et al., 2009; SHERWOOD et al., 2009; PFAFF et al., 2015; SHAHBAZ et al., 2016; KUMAR et al., 2019), and even entire control planes (JIN et al., 2015; BLENK et al., 2015; AFOLABI et al., 2018) running atop a shared infrastructure. These virtual abstractions are highly convenient, as they enable efficient resource usage for network providers and resource isolation for network tenants.

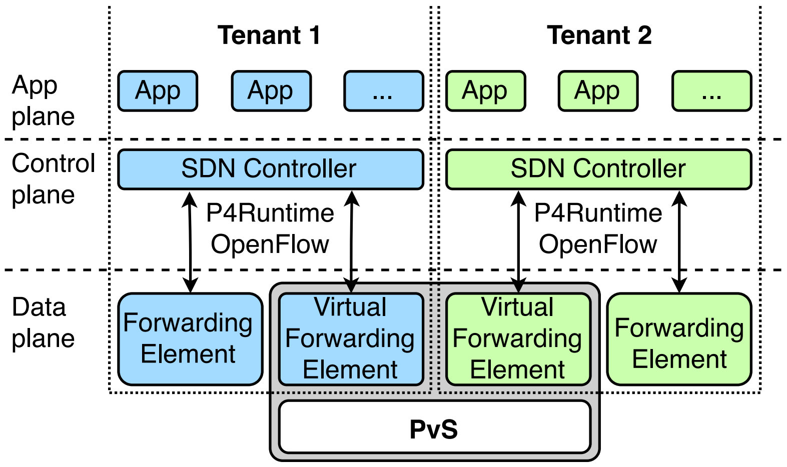
Thus the motivations for virtualization in the data plane are manifold, including network slicing and snapshotting, network function composition, and switch program profiling and debugging (HANCOCK; MERWE, 2016). Yet, existing "hypervisors" for P4-based switches have only enabled switch *emulation* (HANCOCK; MERWE, 2016; ZHANG et al., 2019) or composition of P4-based switch functions into a single program (ZHENG; BENSON; HU, 2018). As a result, they may pose a severe performance penalty and a large memory footprint. More importantly, they require access to switch source code for merging and custom compilation, thus violating the intellectual property of the programmable switches developers and vendors.

In this work, we argue in favor of a virtualization model for programmable forwarding elements that offers switch developers (and vendors) the possibility to distribute switch bytecode while preserving their intellectual property. In such a model, network operators would deploy on the hypervisor the same switch bytecode that otherwise would run directly on bare metal. For being a competitive model, the hypervisor should ensure that each virtual switch delivers the performance similarly, regardless of being run atop the switch hypervisor or hardware.

Delivering such a model is inherently challenging. The fundamental challenge is decoupling virtual switch instances from the hypervisor, while providing a virtual switch abstraction that offers strict performance and isolation guarantees. Also, deploying a virtual switch should not interrupt other running virtual instances, and switch development/compilation should be agnostic of any particularities of the hypervisor. These requirements should be met while ensuring that virtual switch instances retain the same capabilities of a traditional programmable forwarding element. The approach taken is slicing the physical switch into a set of disjoint pipelines of match-action tables for independent control by distinct tenants. As a result, we become able to accommodate multiple switch pipelines running in parallel. One key advantage of this model is that any switch bytecode for the target architecture could be deployed on top of the switch hypervisor, regardless of its internal switching pipeline. It means, for example, that one could write a switch using P4 (BOSSHART et al., 2014), and deploy it in parallel to another switch written in POF (SONG, 2013a). Even a switch written in Hardware Description Language (HDL) could be deployed in the hypervisor, as long as it follows the Reconfigurable Match-action Tables (RMT) (BOSSHART et al., 2013) architecture.

We realize this model in PvS, a system for running *Programmable virtual Switches* in a shared network environment. Figure 1.1 presents an overview of PvS containing two virtual switches from two different tenants. Each virtual switch has its pipeline of match-action stages, along with separate control channel for secure management. Each tenant in the shared infrastructure can deploy its virtual switch and use the most convenient management interface (e.g., Openflow, P4Runtime, or command-line) for making network apps in the application plane operate it.

Figure 1.1: PvS Overview



Source: From the author

1.1 Motivation

There are several motivations for bringing virtualization to the forwarding plane, as exhaustively discussed in prior work (HANCOCK; MERWE, 2016; ZHANG et al., 2019; KRUDE et al., 2019): networking slicing, multi-tenancy in private and public clouds, switch debugging, snapshotting, stateful switch migration, etc. Next we discuss some potential use cases.

Network slicing. Resource slicing is one main motivation for virtualization, and *per se* justifies the need for virtualization of programmable forwarding planes (AFOLABI et al., 2018). Consider for example the roll-out of novel communication technologies like 5G. In order to reduce risks with huge capital expenditure, telecommunication companies might opt to hire slices of radio-base stations (CABALLERO et al., 2017) on a pay-as-you-go basis. The radio-base service provider would need to offer not only the abstraction of a virtual radio-base station, but also abstractions of the networking hardware substrate – including programmable virtual switches – required for data communication between the provider and the telecom company network. A programmable virtual switch would make it easier to interface the provider network with that of the telecom company.

Multi-tenancy in cloud infrastructures. The deployment of a virtual networking infrastructure requires not only abstractions of servers and network functions, but also forwarding devices that will interconnect the virtual infrastructure. The use of software switches to materialize virtual forwarding devices might impose in some cases a performance penalty that violates defined service level objectives. An approach to overcome the performance penalty due to software switching is running programmable switches directly on a hypervisor that offers strict performance guarantees. A public cloud could even offer the notion of a "programmable switch as a service" (KRUDE et al., 2019), which could serve for interconnecting networks from different domains or running in various public cloud infrastructures.

Switch profiling and debugging. A network operator might want to evaluate a home-brewed switch in a semi-experimental scenario, using production workloads. A programmable switch hypervisor would enable that operator to instantiate a virtual switch, and steer (part of) the production traffic to it. The operator could run, for example, A/B acceptance testing, in which the performance of the home-brewed switch is compared with that of the production switch. The operator could also take advantage of such scenario to inspect switches for security vulnerabilities, for example, by replicating input traffic to independently developed switch instances and comparing output packets.

Switch snapshotting and migration. A switch hypervisor, having privileged access to the virtual switch memory, could snapshot the switch state and configured set of match-action table entries. The network operator could then create a catalog of switch configurations, and quickly deploy virtual switch instances capable of handling network flows during peak and off-peak hours, for example. Switch snapshotting could also be useful in scenarios in which a switch must be migrated, along with its state and table configurations, for handling fluctuations in network traffic demands, just like any virtual network function migration (GEMBER-JACOBSON et al., 2014) as seen in the context of Network Function Virtualization (NFV).

1.2 Contributions

This work presents PvS, discussing the approaches adopted to conceive a hypervisor capable of running multiple programmable virtual switches in a single hardware substrate with operation performance comparable to a single switch running as standalone. However, the main focus of this dissertation is the PvS's forwarding engine architecture, exposing its design, implementation, and supporting features. The main finding with this architecture is that it is possible to run virtual switch instances directly from switch bytecode while ensuring predictable performance and isolation. It does so by slicing physical switch resources into multiple pipelines and control. As a direct implication of this finding, we make the following contributions:

- **The design and implementation of PvS's forwarding engine:** Provides a virtual switch operation performance comparable to that of a single switch running as standalone directly on top of the physical programmable switch hardware. To achieve this goal, specific modules were designed and implemented to promote fair and efficient allocation of physical switch resources to virtual instances, in order to maximize the number of instances that can run in parallel, and prevent running instances interfere in the performance of each other;
- **High-Level Synthesis (HLS) flow to virtual switch bytecode:** Enables deployment of a switch bytecode in a Field-Programmable Gate Arrays (FPGAs) board from a program code written in any Domain Specific Language (DSL) for programmable forwarding planes, like POF and P4. A framework to run the HLS flow

was developed, allowing generation and deployment of virtual switches described in P4. The proposed flow facilitates the development of virtualized switches with features to simulate and manage multiple switches in the hypervisor. Code changes are not necessary, which provides protection to the Intelectual Property (IP) and enables the support of the same switch bytecode compiled for the target without the hypervisor;

- **The proposal of a methodology for switch reconfiguration for FPGA architectures:** Enable virtual switch hot-swapping. In other words, a tenant should be able to (un)deploy switch bytecode from/to a programmable virtual switch assigned to that tenant. A methodology was developed to run partial and full FPGA reconfiguration. The partial reconfiguration provides a structure to replace virtual switches without affecting other running virtual switches, while the full methodology delivers the best optimization in terms of resource utilization and performance. Each approach has an optimal utilization case that can be explored to better adapt PvS to different scenarios;
- **A public, open-source, implementation of PvS on GitHub[1]:** PvS was built as a fork of the canonical NetFPGA reference design (IBANEZ et al., 2019). This repository contains the forwarding engine modules, the HLS program to build P4-based virtual switches, the supporting library to test virtual switches execution, examples of different virtual switches implementations, a demo to run the full and partial reconfiguration methodology, and tutorials to enable network operators to use our proposed architecture.

1.3 Outline

The rest of this dissertation is organized as follows. Chapter 2 presents the background knowledge, including basics concepts of domain-specific languages, virtualization, and FPGAs. Also, it overviews switch program emulation, composition, and other reference architectures. Chapter 3 describes PvS, exposing the design principles and the system overview. Chapter 4 details the forwarding engine design and implementation with the modules that support switch virtualization. Chapter 5 describes the proposed HLS flow able to generate virtual switches from P4 descriptions. Chapter 6 explains our

[1]PvS forwarding engine slicing GitHub repo: <https://github.com/pvs-sigcomm/pvs-forwarding-engine>

reconfiguration methodology with pros and cons of using full or partial reconfiguration. Chapter 7 discusses our evaluation in terms of resource usage and performance, measuring reconfiguration times and the impact of virtual switches management, and discussing the in-band telemetry case-study. Finally, Chapter 8 draws the final remarks and possible future works that can be done over the presented forwarding engine and features.

2 BACKGROUND AND STATE OF THE ART

This Chapter discusses background on topics that are fundamental to understanding the work presented in this work, including DSLs, virtualization, Protocol Independent Switch Architecture (PISA), FPGA architectures, and related work addressing switch virtualization.

2.1 Domain-Specific Language (DSL)

A Domain-Specific Language (DSL) is a computer programming language of limited expressiveness focused on a particular domain. There are four key elements to this definition: *Computer programming language*: A DSL is used by humans to instruct a computer to do something. As with any modern programming language, its structure is designed to make it easy for humans to understand, but it should still be something executable by a computer. *Language nature*: A DSL is a programming language, and as such should have a sense of fluency where the expressiveness comes not just from individual expressions but also from the way they can be composed together. *Limited expressiveness*: A general-purpose programming language provides lots of capabilities: supporting varied data, control, and abstraction structures. All of this is useful but makes it harder to learn and use. A DSL supports a bare minimum of features needed to support its domain. You can not build an entire software system in a DSL; rather, you use a DSL for one particular aspect of a system. *Domain focus*: A limited language is only useful if it has a clear focus on a small domain. The domain focus is what makes a limited language worthwhile (FOWLER, 2010).

The adoption of a DSL involves pros and cons, working with this approach means finding the balance between these two. The benefits of DSLs include: DSLs allow solutions to be expressed in the idiom and at the level of abstraction of the problem domain. Consequently, domain experts themselves can understand, validate, modify, and often even develop DSL programs (DEURSEN; KLINT; VISSER, 2000); Programs are concise, self-documenting to a large extent, and can be reused for different purposes (LADD; RAMMING, 1994); DSLs enhance productivity, reliability, maintainability (KIEBURTZ et al., 1996); Allow validation and optimization at the domain level (BRUCE, 1997); DSLs improve testability following approaches such as (SIRER; BERSHAD, 1999); Still according to Deursen, Klint e Visser (2000), the disadvantages of the use of a DSL are:

The costs of designing, implementing and maintaining a DSL; The costs of education for DSL users; The difficulty of finding the proper scope for a DSL; The difficulty of balancing between domain-specificity and general-purpose programming language constructs; The potential loss of efficiency when compared with hand-coded software; And according Krueger (1992), the limited availability of DSLs ;

Operators can use domain-specific languages to effectively program network behavior, from control plane applications to forwarding devices. In the following sections, we describe two prominent DSLs that enable data plane programmability: POF (SONG, 2013b) and P4 (BOSSHART et al., 2014). POF is designed to inter-operate with OpenFlow (MCKEOWN et al., 2008a) and enable greater flexibility in packet processing definition, whereas P4 provides higher-level abstractions to define protocol headers, header parsing logic and packet to achieve the same main goal as POF. Although there are other DSLs that enable data plane programmability, P4 and POF are the only that are recognized by the Open Networking Foundation[1] (ONF), responsible for fostering open networking standards.

2.1.1 Protocol-Oblivious Forwarding (POF)

As one of the initial implementations of Software-Defined Networking (SDN), OpenFlow provides a powerful tool set for network operators to program and manage their networks adaptively (MCKEOWN et al., 2008b). However, its protocol-dependent nature still limits the programmability of the forwarding plane. Specifically, OpenFlow defines the matching fields in flow tables according to existing network protocols (e.g., Ethernet and Internet Protocol). Therefore, OpenFlow switches need to understand the protocol headers to parse packets and perform flow matching, which may cause serious compatibility issues when new protocols try to add or remove header fields. Hence, it is desirable that the network programmability can be further enhanced such that the forwarding plane is protocol-independent and can be dynamically reprogrammed to support new protocol stacks seamlessly. Following this idea, Protocol-Oblivious Forwarding (POF) was proposed, a new SDN technologies to try to decouple network protocols from packet forwarding and make the forwarding plane reconfigurable, programmable, and future-proof (LI et al., 2017).

More specifically, POF is an OpenFlow extension that introduces a protocol-inde-

[1]https://www.opennetworking.org/

pendent instruction set, which allows a network operator to define the protocol stack and packet processing procedure in a much more flexible manner than that in the current OpenFlow specifications. This extension is designed to enable rules to reference arbitrary packet header fields and to control the table matching behavior. Header fields are referenced within table entries in POF using type, offset, length tuples, where type is set to 0 to indicate a header field or 1 to indicate a meta-data field. Meta-data fields may contain arbitrary data and serve the purpose of carrying packet information through the processing pipeline. The offset is the field starting position relative to the first bit of the header. The field length is specified in bits (SONG, 2013b).

Using POF to support new protocols, the operators need to download some flow rules with associated instructions into the network elements. This instruction set is named POF-FIS. POF-FIS is an enhancement and extension of the instructions and actions defined in OpenFlow 1.x. In the POF network element, network flows are handled by flow instructions in the form of POF Flow Instruction Blocks (POF-FIB). POF-FIBs are deployed by a controller through the POF southbound interface. All the flow instructions in POF-FIBs are defined in POF-FIS. The POF controller can use a Command-Line Interface (CLI), a Graphical User Interface (GUI), or a high-level programming language compiler as the northbound interfaces to users and applications (YU et al., 2014).

The packet parsing process is done through table redirection and specially defined decision tables. These tables match on header fields that indicate the next protocol type, for example, Ethernet *EtherType*, and their entries are populated to redirect the packet parsing appropriately. For example, suppose two decision tables dedicated to Internet Protocol version 4 (IPv4) and Internet Protocol version 6 (IPv6) routing, respectively. A third table could be defined to match the Ethernet *EtherType* field, and populated with two entries. One of them would have a matching value of 0x0800, and redirect the packet parsing, via Goto-Table instruction, to the IPv4 table. The other entry would have a matching value of 0x86DD, and redirect the parsing to the IPv6 table. The packet parsing is thus analogous to OpenFlow, but with the ability to match and modify arbitrary header fields (CORDEIRO; MARQUES; GASPARY, 2017).

2.1.2 Programming Protocol-Independent Packet Processors (P4)

Programming Protocol-Independent Packet Processors (P4) is a high-level language that provides an abstract model suitable for programming the network data plane with the definition of its behavior (BOSSHART et al., 2014). Its main aspect goals are reconfigurability, protocol independence, and target independence. The reconfigurability proposes that packet parsing and processing may be redefined by the controller. The protocol independence suggests that the controller should be able to extract header fields by specifying the packet parser and a collection of header processing match-action tables. The target independence focuses on not getting irrelevant information about the switch.

A P4 program defines essentially six items: (i) headers, specify the names and width of the protocol fields which the program operates on; (ii) metadata, structures that provide specific information of the packet; (iii) parser, group of state machines to analyze headers and extract data for the metadata structures; (iv) match-action table, identifying packets fields and metadata that will be compared and the possible actions being executed in response; (v) execution pipeline, stage actions, determined control flow, to define how packets are processed; (vi) deparser, process to rebuild packets.

The task flow of the device definition with capacity for P4 starts with the operators writing and compiling a P4 program through a front-end compiler in a High-Level Intermediate Representation (HLIR). Then, a back-end compiler will adapt the program to different targets, including FPGAs. Finally, through an execution time controller, operators can fill entries in match-action tables in the data plane and make destination devices process packets like they are preset.

The main difference between these two DSLs is that POF is an extension of OpenFlow, and P4 is a protocol-independent language created to describe how the packets are processed. While POF is a more direct evolution of OpenFlow, designed to inter-operate with it and enable greater flexibility in packet processing definition, P4 provides higher-level abstractions to define protocol headers, header parsing logic, and packet processing. Both have the same main goal, to make the data plane programmable and reconfigurable.

2.2 Virtualization

The work presented in (ANDERSON et al., 2005) conceptualizes virtualization as nothing more than a high-level abstraction that hides the underlying implementation details, for example, as used in virtual memory, virtual machines, and elsewhere. With virtualization, nodes can treat an overlay as if it is the native network, and multiple overlays can simultaneously use the same underlying infrastructure.

The application of virtualization has become common in modern public and private clouds, powering virtual abstractions for servers. The high costs involved in the acquisition and maintenance of servers have caused the migration of local servers to remotely accessed virtualized servers. In the past fifteen years, server virtualization has become the dominant approach for managing computational infrastructures, with the number of virtual servers exceeding the number of physical servers globally (BITTMAN et al., 2013; KOPONEN et al., 2014). This approach shares resources to deliver a multi-tenant scheme, in which a user receives an individual software abstraction to run its processes.

The idea of a Virtual Internet (VI) presented by (TOUCH et al., 2003) is a network composed of tunneled links among a set of virtual routers and virtual hosts. The architecture provides an abstraction that hides the complexity of the underlying network and provides isolation-based protection that encourages resource sharing. The network virtualization completely decouples its component hosts and routers from the underlying network to support recursive VIs and to allow a node to participate multiple times in a single overlay, known as re-visitation. This architecture provides a consistent multihoming paradigm, including end-to-end overlays, naming and addressing, virtual host requirements, and virtual gateway requirements.

Network Function Virtualization (NFV) has been recently proposed to turn middleboxes into software-based, virtualized entities (MARTINS et al., 2014; HWANG; RAMAKRISHNAN; WOOD, 2015); provide high performance and flexible networking using virtualization on commodity platforms (HWANG; RAMAKRISHNAN; WOOD, 2015) and reconfigurable hardware (LI et al., 2016); and the comprehensive survey of network function virtualization (YI et al., 2018). These virtual abstractions are highly convenient, it provides virtualization, isolation and non-interrupted execution of network functions.

Resource sharing through virtualization in forwarding devices take the VI concept closer to reality. The research community has been proposing solutions to decouple hosts and routers from the underlying network. In the work of the authors DOBRESCU et al. is presented a software router architecture to parallelize router functionality across multiple cores within a single server. Different approaches using SDN hypervisor architectures, centralized and distributed also are presented to run on general-purpose computing platforms with special-purpose network elements (SHERWOOD et al., 2009). Forwarding systems with predictable performance have been proposed in the literature and are able to share resources efficiently, while rapidly reacting to ensure isolation in public and private clouds (KUMAR et al., 2019).

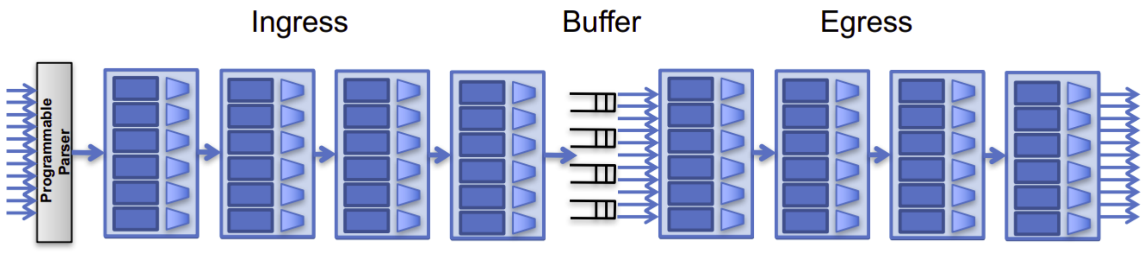
This work advocates in favor of network virtualization with the ability to run virtual instances of routers, switches, links, and network functions on top of a physical network substrate. In addition to enabling experimentation with novel network architectures, this virtual networking also promotes the sharing of physical resources, powering a whole new class of networking-on-demand services.

2.3 Protocol Independent Switch Architecture (PISA)

Programmable switches enable the implementation of many complex network functions directly in the data plane. The Protocol Independent Switch Architecture (PISA) proposes the Reconfigurable Match Tables (RMT) model, a Reduced Instruction Set Computer (RISC) inspired pipelined architecture for switching chips, through the identification of the essential minimal set of action primitives to specify how headers are processed in hardware. RMT allows the forwarding plane to be changed on-the-fly without modifying the hardware. As in OpenFlow, the programmer can specify multiple match tables of arbitrary width and depth, subject only to an overall resource limit, with each table configurable for matching on arbitrary fields. (BOSSHART et al., 2013).

The PISA architecture consists of a large number of physical pipeline stages that a smaller number of logical RMT stages can be mapped to, depending on the resource needs of each logical stage. After entering a PISA switch, packets first go through an ingress pipeline, then enter the traffic manager that maintains multiple queues, and are finally processed by an egress pipeline (POKORNY et al., 2008).

Figure 2.1: The Basic PISA Pipeline



Source: Programming The Network Data Plane (KIM, 2017)

Figure 2.1 shows the processing pipeline stages in more detail. Note that there are three fundamental elements, the Ingress match-action, Buffer, and the Egress match-action. The Ingress and Egress match-action stages are composed of memory-implemented modules, such as lookup tables, counters, meters, and generic hash tables; and processing modules, such as standard boolean and arithmetic operations, header modification operations, hashing operations, etc. The packets are recirculated through these stages up until the processing is complete.

The implementation of the PISA is frequently prototyped in Application Specific Integrated Circuits (ASIC) - i.e. Barefoot Tofino ASIC (BAREFOOT, 2019). The ASIC provide their users the ability to manufacture products having a proprietary design without having to begin the design at the device level (CHAN; BIRKNER; CHUA, 1992). The main language used to program these devices is P4, that allows developers to describe how packets are processed by a programmable data plane, spanning ASICs and processors, implementing PISA (LUINAUD et al., 2020). However, the hardware implementation is non-trivial and imposes constraints on the number of bytes that can be examined in the packet header and on the number of match-action stages in a pipeline (JEPSEN et al., 2019).

2.4 Field-Programmable Gate Arrays (FPGA)

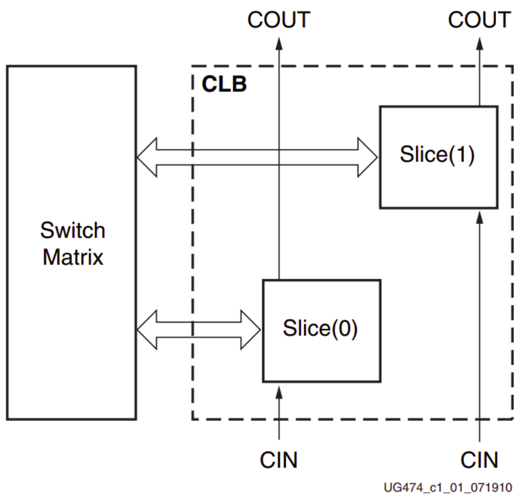
The use of hardware-accelerated network devices in switches and routers are enabling a rapid growth of the Internet. Today, gigabit ethernet switches are widely deployed to switch traffic within Local Area Networks (LAN) and route Internet Protocol packets across Wide Area Networks (WANs). Commercial vendors use ASICs and/or FPGAs to accelerate switching, routing, and processing of network data (LOCKWOOD et al., 2007). As elucidated in the previous sections, the newer generation of SDN-related solutions introduced the notion of data plane programmability - e.g., P4 and POF. They enable faster development/provisioning of novel and/or home-brewed protocols, as opposed to the long wait for the release of fixed-function ASIC switches supporting standardized protocols (SIVARAMAN et al., 2015). The intrinsic characteristics of Field-Programmable Gate Arrays (FPGAs) are aligned with the data plane programmability which can be a powerfully tool to network developers make the forwarding plane reconfigurable, programmable, and future-proof.

More specifically, the FPGAs are pre-fabricated silicon devices that can be electrically programmed to become almost any kind of digital circuit or system (KUON et al., 2008). They provide a number of compelling advantages over fixed-function ASICs, such as standard cells (CHINNERY; KEUTZER, 2002): ASICs typically take months to fabricate and cost hundreds of thousands to millions of dollars to obtain the first device; FPGAs are configured in less than a second (and can often be reconfigured if a mistake is made) and cost anywhere from a few dollars to a few thousand dollars.

The flexible nature of FPGA comes at a significant cost in area, delay, and power consumption: an FPGA requires approximately 20 to 35 times more area than a standard cell ASIC, has a speed performance roughly 3 to 4 times slower than an ASIC, and consumes roughly 10 times as much dynamic power (KUON; ROSE, 2007). These disadvantages arise largely from an FPGA's programmable routing fabric that trades area, speed, and power for "instant" fabrication.

To create the circuit to be loaded into a FPGA board, a designer traditionally describes the circuit using Hardware Description Languages (HDL). HDLs describe the behavior of physical devices and processes, a task commonly called modeling. Models written in HDL are used as input to a suitable simulator to analyze the behavior of the devices before the design implementation (CHRISTEN; BAKALAR, 1999). Verilog and VHDL are hardware description languages that provide means of specifying a digital system at a wide range of levels of abstraction. Both languages support the early conceptual stages of design with its behavioral level of abstraction, and the later implementation stages with its structural level of abstraction. The hierarchical modeling of the system can be achieved with top-down and bottom-up design methodologies, and the system and its subsystems can be described at any level of abstraction ranging from the architecture level to gate level (BHASKER, 1999; THOMAS; MOORBY, 2008).

Figure 2.2: Arrangement of Slices within the CLB of the FPGA Xilinx Virtex 7



Source: Xilinx 7-Series Manual (XILINX, 2018)

The FPGA fabric is composed by fundamental building block named Configurable Logic Blocks (CLBs) that provide the physical support for an implemented and downloaded design. The CLB is the FPGA's main logic resource for implementing combinatorial and sequential logic functions. Each CLB is connected to the general routing fabric through a switch matrix (CHANDRAKAR; GAITONDE; BAUER, 2015). The Figure 2.2 shows how the arrangement of slices form the CLB structure of the Xilinx Virtex 7, the FPGA embedded in this work's target board. The slice is built from Basic Elements of Logic (BELs) such as Look up Tables (LUTs), flip-flops, carry logic, and wide-function multiplexers (XILINX, 2018).

The configuration data of a FPGA is stored in a file called bitstream (.bit), which defines the FPGA's logic functions and circuit connections. This configuration remains active in the FPGA until it is erased or overwritten. For the NetFPGA SUME board, erasing can be accomplished by resetting the board and overwriting can be accomplished by writing a new configuration file to the Joint Test Action Group (JTAG) port or by triggering the on-board Complex Programmable Logic Device (CPLD) to load a new bitstream from the parallel flash memory (Digilent Inc., Accessed: July 2019).

The use of FPGAs allows us not only to rapidly prototype PvS, but also to take advantage of its flexibility, high performance, and module reconfiguration. The main difference in the use of FPGAs, when compared to the use of microprocessors, is the ability to make substantial changes to the hardware, including modifications to the data and control flows (YANG et al., 2014), in addition of better efficiency in terms of performance and power consumption, since it implements a dedicated circuit. When compared to using a custom ASIC, the advantage of using FPGAs is the possibility of changing the hardware at run time by loading a different circuit in the reconfigurable matrix.

2.5 Related Work

This work posit that a data plane hypervisor must allow network operators to deploy and hot-swap truly independent virtual P4-based switch instances, deliver a performance similar as if the virtual instances were running directly on hardware, and allow vendors to distribute switch byte-code while protecting their intellectual property. Although previous investigations partially fulfill some of these requirements, satisfying them simultaneously poses the following research challenges: (i) decouple virtual switch instances from the virtualization environment, (ii) provide network flow isolation, (iii) ensure hardware resource isolation, (iv) support virtual networking within the hypervisor, and (v) deliver an implementation with feasible performance and memory footprint. In the following section, we discuss the state-of-the-art related works to elucidate our proposed approach.

2.5.1 Switch Program Emulation

Hancock and van der Merwe (HANCOCK; MERWE, 2016) were the first to propose the idea of a general purpose P4 switch program (called Hyper4) that exposes a set of match-action tables that can be configured in a way to emulate other P4 programs. The Hyper4 compiler is responsible for parsing switch programs and generating the necessary Hyper4 match-action table entries required to emulate them. Hyper4 makes extensive use of P4 packet `recirculate` primitive, which imposes high performance penalty to emulated switches. Zhang et al. (ZHANG et al., 2017; ZHANG et al., 2019) followed with HyperVDP, a solution that also emulates multiple P4 program switches using a single general purpose program. Similarly to Hyper4, HyperVDP also suffers from performance penalty due to emulation. In addition, both require several match-action stages to emulate a single match-action stage, thereby incurring in high memory footprint.

SR-PVX (LI et al., 2017) follows a similar approach to Hyper4, but focuses on POF (SONG, 2013a). SR-PVX enables instantiating virtual POF switches over designated substrate switches, and uses source routing to realize virtual links in virtual SDN slicing. The deployment flow of virtual switches involves (i) deployment of tenant-defined flow-table(s) in the substrate switch pipeline, and (ii) configuration of match tables in the substrate switch so that packets belonging to a specific SDN are processed by the corresponding tenant-defined flow-tables. In a follow-up work PVFlow (LI et al., 2018), the authors unify flow-tables that contain arbitrary matching fields with various lengths, enabling them to share the Ternary Content Addressable Memory (TCAM) of a matching stage in a substrate switch.

2.5.2 Switch Program Composition and Parallel Execution

P4Visor (ZHENG; BENSON; HU, 2018), Dejavu (WU et al., 2019), and P4SC (CHEN et al., 2019) are solutions that compose multiple switch codes into a single program, and that optimize resource usage through merging of similar switch program functionalities. The composed programs can then be configured in a way that emulates various switch functions running in the substrate switch. However, they do not support switch isolation and interruption-free deployment of new switch instances.

Krude et al. (KRUDE et al., 2019) discussed research avenues towards providing *Programmable Switches as a Service*, including switch hot-pluggability in the data plane, which have been partially addressed in prior work like P4Visor (ZHENG; BENSON; HU, 2018), VirtP4 (SAQUETTI et al., 2019), and P4VBox (Saquetti et al., 2020). VirtP4 (SAQUETTI et al., 2019) and P4VBox (Saquetti et al., 2020) propose running multiple switch instances on a NetFPGA board, but lack management abstractions for controlling virtual switch instances. In our work, we show how the architectural requirements discussed in (KRUDE et al., 2019) are realized to deliver a hypervisor for programmable virtual switches running on a NetFPGA board.

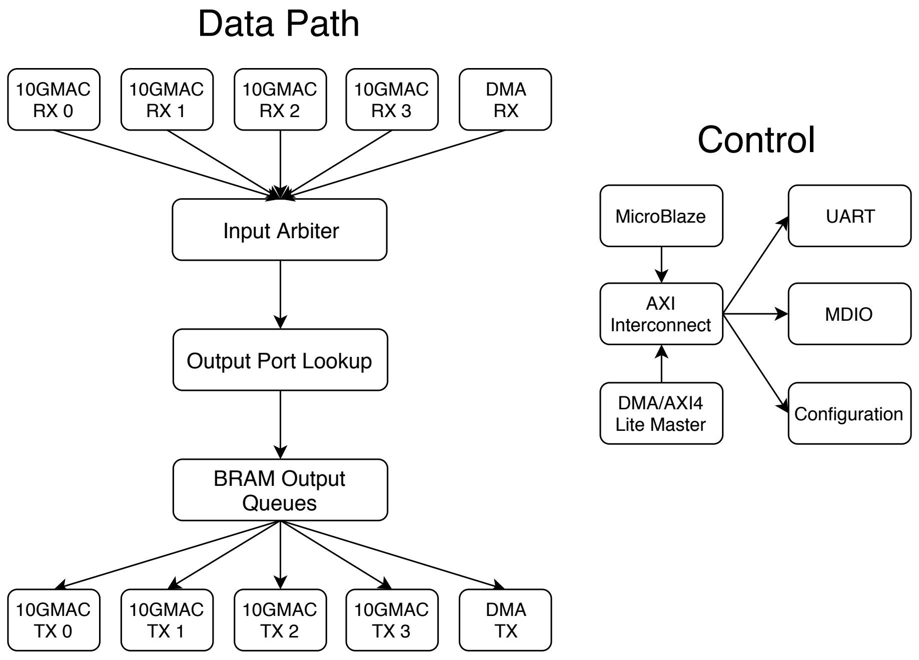
2.5.3 Software Switches

PISCES (SHAHBAZ et al., 2016) follows a best-of-breed approach for enabling customization of software switches through a high-level domain-specific language for programming packet parsing and processing logic. Derived from Open vSwitch (PFAFF et al., 2015), it provides a protocol-independent virtual switch whose behavior can be modified at will. Although multiple software switch instances could run in parallel, it is unclear whether each virtual switch instance would perform under intense competition for shared resources like CPU and physical switch ports. Still, such approach would require a supervisor program to steer packets between virtual switch instances and ensure network isolation.

2.5.4 Canonical NetFPGA Reference Design Architecture

The canonical NetFPGA design is based on the Simple Sume Switch (SSS) architecture, which supports a single switch description, its structure is showed in Figure 2.3. It consists of four 10Gbps and DMA I/O ports, Input Arbiter (IA), Output Port Lookup (OPL), and BRAM Output Queues(BOQ). The IA is responsible for serializing the packet frames received from the input ports and delivering them to the OPL module. The OPL is a placeholder to accommodate an implementation of an independent P4-based switch pipeline, which performs packet processing and determines the output port. The BOQ buffers packets before they are copied to the output ports. The Control enables system monitoring and configuration.

Figure 2.3: Canonical NetFPGA reference design



Source: From the author

Table 2.1 describes the format of the SSS's *sume_metadata* bus and the functionality of each field. The architecture could also be extended by more advanced users, or, for the most adventurous users, it could be completely replaced by writing a new architectural model (IBANEZ et al., 2019).

Table 2.1: Simple Sume Switch sume_metadata Fields Description

Field Name	Size (bits)	Description
pkt_len	16	Size of the packet in bytes (not including the Ethernet preamble or FCS)
src_port	8	Port on which the packet arrived (one-hot encoded)
dst_port	8	Set by the P4 program - which port(s) the packet should be sent out of (one-hot encoded)
send_dig_to_cpu	8	Set the least significant bit of this field to send the digest_data to the CPU
*_q_size	16	Size of each output queue at P4 processing start time, measured in of 32-byte words

Source: (IBANEZ et al., 2019)

For generating a switch implementation from a P4 description, the P4-NetFPGA uses a HLS methodology applied to a SSS architecture creating an OPL. The SSS is similar to the Very Simple Switch (VSS) reference model and supports switch logic composed of a single packet parser, match-action pipeline, and packet deparser.

2.5.5 Summary

Most prior work have focused on *emulating* programmable switch pipelines through a single switch program, or *composing* several switch functions into a single program (see Table 2.2). Observe that prior work on virtualization of the forwarding plane lack mechanisms for ensuring predictable performance and tenant isolation. A common approach for virtualization has been using switch program composition, for example, through a merged packet parser and a set of match-action tables for emulating multiple pipelines.

Table 2.2: Existing work on virtualization of forwarding elements.

Solution	Underlying hypervisor	Abstraction	Model	Target DSL	Compilation / Merging
Hyper4 (HANCOCK; MERWE, 2016)	Software switch	Virtual switch	Emulation	P4	Compilation
HyperVDP (ZHANG et al., 2019)	Software switch	Virtual switch	Emulation	P4	Compilation
P4Visor (ZHENG; BENSON; HU, 2018)	Single switch program	Switch function	Emulation	P4	Merging
Dejavu (WU et al., 2019)	Single switch program	Switch function	Emulation	P4	Merging
P4SC (CHEN et al., 2019)	Single switch program	Switch function	Emulation	P4	Merging
SR-PVX (LI et al., 2017)	Single switch program	Virtual switch	Emulation	POF	Compilation
PV-Flow (LI et al., 2018)	Single switch program	Virtual switch	Emulation	POF	Compilation
PvS	Hard virtualization	Virtual switch	Virtualization	Independent	–

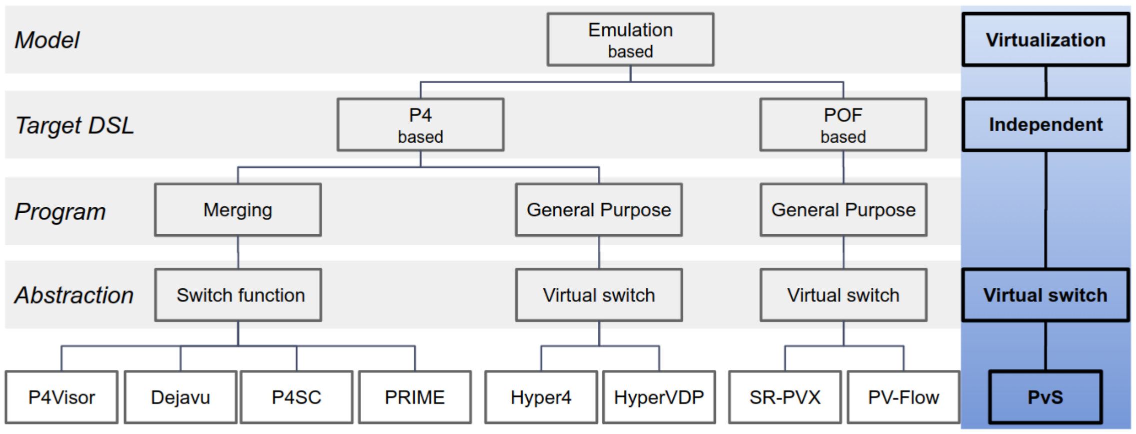
Source: From the author

The main drawbacks of these techniques are that: (i) they do not provide true switch isolation, as switch programs are merged into a single one, and therefore share physical resources, (ii) any modification to a switch program requires full recompilation, (iii) updating a single switch program requires full system halt, (iv) incur in performance degradation in terms of increased latency a decreased bandwidth, or (v) not provide a virtualization features.

3 PVS PROPOSAL

This Chapter presents an overview of PvS, a new solution of virtualization for programmable data planes that aims to fill the existing virtualization gap in the literature, as depicted in Figure 3.1. This system provides true virtualization running multiple pipelines of programmable virtual switches on top of the same hardware platform, providing performance comparable as a single switch running as standalone. The support to multiple DSLs allows the co-existence with legacy solutions and is capable of making the network infrastructure more extensible and reusable, while the hot-swapping process enables the easy (un)deploy of new virtual switches to do debugging or migration without interfering in the already running pipelines. In Section 3.1, we describe the design principles that a hypervisor should satisfy, which guided the PvS implementation, whereas Section 3.2 provides a PvS overview and define the main properties to programmable switch abstraction.

Figure 3.1: PvS proposal in the literature



Source: From the author

3.1 Design Principles

As a general goal, any hypervisor for the data plane should offer the network developer an abstraction of a programmable virtual switch. In other words, the hypervisor should enable the network developer to program a virtual switch placeholder as if the physical switch hardware itself were being programmed.

As a direct consequence of the goal above, any suitable hypervisor for the data plane should offer the following set of features:

- Promote fair and efficient allocation of physical switch resources to virtual instances, in order to maximize the number of instances that can run in parallel, and prevent running instances interfere in the performance of each other (Chapter 4);
- Provide virtual networking between virtual switch instances, enabling them to exchange flows between each other. This is the case for virtual networking within the hypervisor, a useful feature in a scenario where multiple switch instances belong to a same tenant, and are part of a same domain (Chapter 4).
- Ensure that virtual switch instances do not process network flows or read/write memory blocks or virtual ports beyond their control, e.g., from other virtual switches running in parallel. Such feature is essential for ensuring that a malicious virtual switch does not tamper with other running instances (Chapter 4 and 5);
- Enable the deployment of a switch bytecode compiled from a program written in any Domain Specific Language (DSL) for programmable forwarding planes, like POF and P4 (Chapter 5);
- Support the same switch bytecode compiled for the target it is running on. For example, if the hypervisor is running on NetFPGA, then it should run virtual switch instances from a switch bytecode compiled for a NetFPGA target (Chapter 5);
- Enable virtual switch hot-swapping. In other words, a tenant should be able to (un)deploy switch bytecode from/to a programmable virtual switch assigned to that tenant (Chapter 6);
- Provide a virtual switch operation performance comparable to that of a single switch running standalone directly on top of the physical programmable switch hardware (Chapter 7);
- Ensure that network apps running in the SDN application plane only gain any access to virtual switches if they provide valid access credentials. Such feature is critical for ensuring secure operation of virtual switch instances and tampering from a malicious tenant.

In addition to these features, the deployment of a virtual switch instance must not require access to switch source code or custom switch program compilation. Such feature is important for protecting the intellectual property of switch developers and vendors. In contrast, prior work require custom compilation or switch program composition for running it as a virtual switch instance.

To satisfy these features, we propose the following design principles for a programmable forwarding plane virtualization solution:

P1 *Forwarding engine slicing.* For predictable performance and tenant isolation, physical switch resources should be sliced proportionally to the desired performance, and committed to virtual switch placeholders. In particular, resources like memory space for holding the switch bytecode, match-action table entries, and counters, as well as packet queues and buffers, should be allocated per deployed switch instance, to ensure their feasible operation.

P2 *Control engine abstraction.* For ensuring secure management and compatibility with open control-data plane interfaces, the hypervisor should offer an abstraction of a single control engine and management channel to network apps running in the application plane.

The forwarding engine slicing (P1) enables accommodating virtual switch instances with predictable performance with regard to packet processing capabilities, and known storage capacity for handling match-action table entries and packet counters. In turn, the use of slicing requires accessory modules for ensuring that (i) incoming packets are steered to correct virtual switch instances, (ii) output packets are forwarded to the correct physical switch ports, and (iii) match-action table entry and register reading/writing operations do not overflow. More importantly, such modules should provide virtual port and ingress/egress buffer and queue abstractions for virtual switch placeholders, so that these placeholders can accommodate the same switch bytecode one would deploy directly on the physical switch. While P1 is necessary for predictable performance and enforce secure virtual switch operation at the forwarding engine, it does not cover control to data plane management channels, neither is capable of enforcing secure virtual switch management.

The control engine abstraction (P2) provides a proper abstraction of a virtual switch control engine and satisfies the last mentioned design principle. Such abstraction should intercept message requests sent through the control channel, and ensure that they are accepted and processed by the virtual switch only if proper credentials are supplied beforehand. It should also ensure that tenants do not perform read/write requests to match-action table entries and counters beyond their control. Observe also that regardless of the control engine abstraction used, the hypervisor must ensure that it is not reprogrammed by a malicious tenant so that these security properties are subverted, and a malicious tenant becomes able to access restricted memory regions in the forwarding engine.

In this dissertation, we focus on the PvS Fowarding Engine, a programmable forwarding plane hypervisor that implements the forwarding engine slicing (P1) and all-but-the-last design principles. It enables compiled switch bytecode to run as virtual switch instances, with a performance comparable to what would be achieved if the switch bytecode were running directly on the physical programmable switch hardware. With the use of this virtualization system, we can have, for example, a layer-2 switch, a simple router, a firewall switch, and an in-band telemetry running on top of PvS. The control engine abstraction (P2), which satisfies the last design principle, is left out of the scope of this dissertation.

3.2 PvS: Programmable virtual Switch Hypervisor for the Data Plane

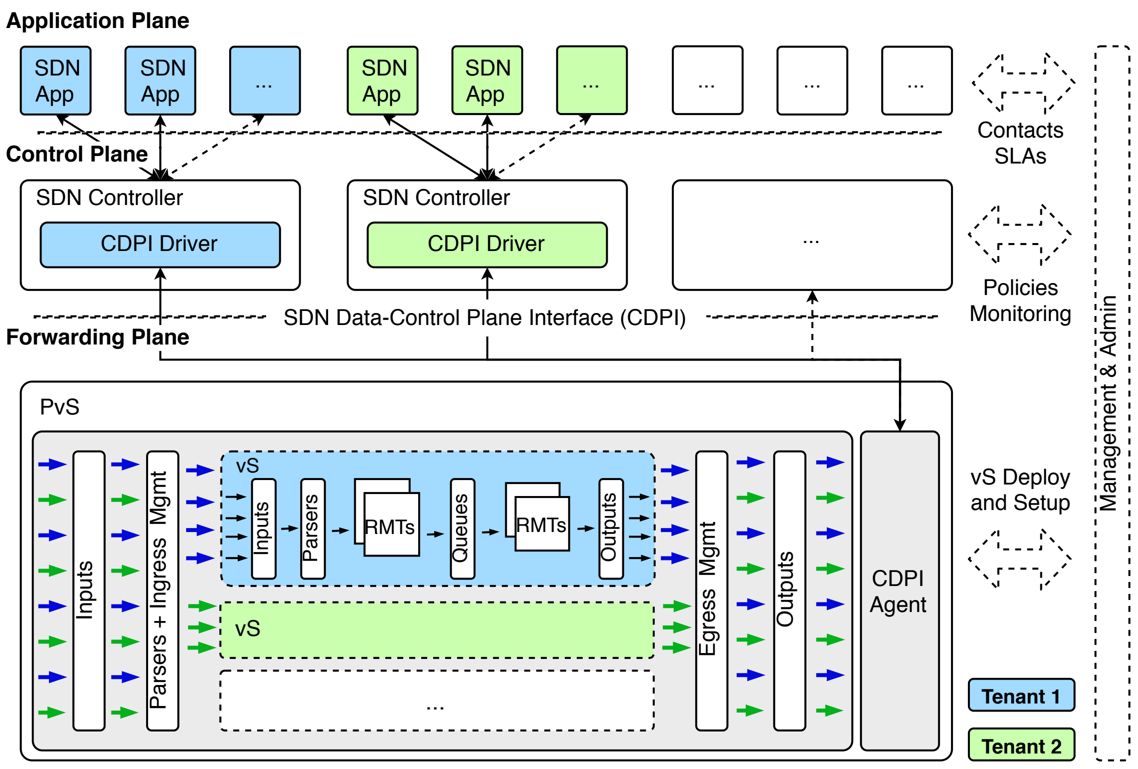
The PvS design is guided by the notion that a hypervisor for programmable forwarding planes should offer to network developers a proper abstraction of programmable virtual switches. Under such notion, and considering a scenario in which the hypervisor runs on top of a NetFPGA board, then the hypervisor should offer to each network developer the illusion of having their own dedicated NetFPGA board – which is, in practice, a slice of resources of the physical NetFPGA board. The hypervisor manager should be able to create as many slices for virtual switches as possible, and network developers should be able to instantiate any switch program on a slice. The operation of virtual switch instances should not interfere on switches running in other slices.

A physical switch may be characterized, for example, by (i) amount of memory available for storing the switch bytecode and table entries, (ii) number of network ports and their speed, (iii) switch throughput capacity, (iii) packet processing capacity per second, (iv) latency, and (v) number of dedicated management interfaces[1]. In this context, we define a programmable switch abstraction along properties like memory capacity, virtual switch ports, and dedicated management interfaces. The goal of PvS, as a programmable switch hypervisor, is supporting such abstractions while ensuring virtual switch performance and isolation.

Figure 3.2 provides an overview of PvS, and its relationship with a software-

[1]The physical switch characterization discussed in this paper is based on the datasheet of a STORDIS BF2556X-1T Barefoot Tofino-based programmable switch: <https://www.stordis.com/wp-content/uploads/2019/12/STORDIS_BF2556X-1T-A1F.pdf>

Figure 3.2: An overview of a conceptual architecture of a programmable virtual switch hypervisor, and its relationship with key elements that compose a high-level Software Defined Networking (SDN) architecture.



Source: From the author

defined networking architecture. PvS is composed of a *forwarding engine* and a *control engine*. The forwarding engine is responsible for accommodating virtual switch pipelines, whereas the control engine provisions the functionalities of a Control-Data-Plane Interface (CDPI) agent, as discussed in the SDN technical reference specification (FOUNDATION, 2013). The CDPI agent enables network applications, belonging to different tenants (colored in the figure for illustration purposes), interact with virtual switch instances through standard management interfaces like OpenFlow and P4Runtime.

3.2.1 Memory slicing

The network manager may define the amount of memory allocated to a virtual switch instance either considering the specific memory requirements of the switch bytecode to be deployed, or according to a desired arbitrary amount. The memory space allocated should consider both the required space to store the switch bytecode and the match-action table entries plus counters. The memory region belonging to a virtual switch instance should be protected from unauthorized access, in order to ensure security and privacy of processed flows.

3.2.2 Virtual ports

After entering the physical switch input queue, network flows must be delivered to a virtual switch port. The network manager may define an arbitrary number of virtual ports for each switch instance, and a mapping of physical to virtual ports. The PvS parser and ingress management module is responsible for identifying the tenant associated to an incoming packet, and to which virtual switch pipeline an incoming packet must be steered to (note that a tenant may own multiple virtual switch instances). Such identification must be done based on the input physical port and values matched from the incoming packet header fields. Existing solutions considered custom packet tagging (HANCOCK; MERWE, 2016; ZHANG et al., 2017) to assist that identification; traditional protocols for network segmentation, like VLAN and Virtual Extensible LAN (VXLAN) may also be used to this end.

Once processed by a virtual switch, packet flows must be steered out of the switch (if not marked for drop). Similarly to input port mapping, the network manager must determine the virtual to physical output port mapping. The PvS egress management module is responsible for determining to which physical output port the packet must be copied to. To this end, it considers the virtual switch and port from which the packet is coming from. For ensuring secure packet steering, the egress management module should also check if the packet tag identifier (for example, VLAN/VXLAN id) matches the tenant that owns the virtual switch that forwarded the packet.

3.2.3 Management interfaces

The hypervisor must provide open management interfaces to virtual switches, so that they can be easily coupled to a software-defined network substrate. At the same time, the hypervisor must guarantee that the management interface is not subverted so that a malicious tenant gain unauthorized access to virtual switches. For this reason, PvS also implements the control engine abstraction, an abstraction of management channel to tenants, that is materialized through a single Control-Data-Plane Interface (CPDI) agent.

The agent receives requests from the CPDI driver, identifies the virtual switch target of the request, and then process it accordingly. Note that providing a management channel abstraction requires extending management specifications to support tenant authentication.

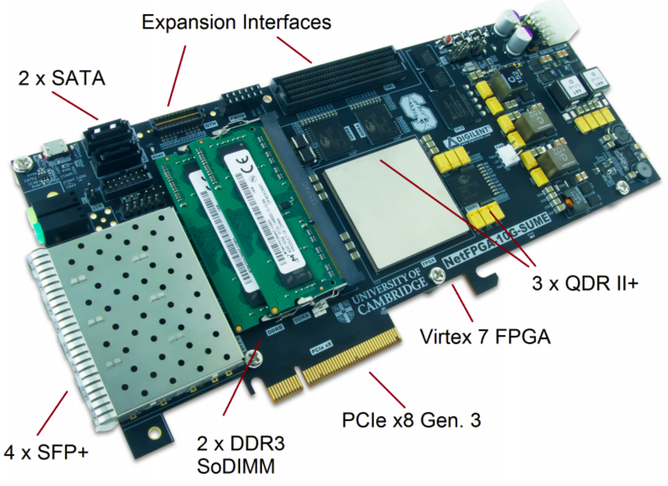
As previously mentioned, we focus on the forwarding engine slicing (P1) (Chapter 4), its supporting features (Chapter 5), and its reconfiguration capabilities (Chapter 6). Thus, we do not directly address the control engine abstraction (P2), only its interactions with the forwarding engine. The hypervisor seeks to accommodate the switch abstraction properties above, while ensuring that a network developer remains free to devise switch programs according to one's requirements and needs. More importantly, it seeks to deliver optimal memory occupancy per virtual switch instance and negligible overhead, both in terms of flow processing and operational management.

4 FORWARDING ENGINE IMPLEMENTATION

This work proposed a forwarding engine architecture for virtualization multiple virtual switch (vS) instances in a single hardware subtract. In typical switch system, only one switch can run on hardware and this system has a specific language supports. To make a target independent platform, this approach enable the portability to any FPGA-based architecture with minor adjustments in the Intellectual Property (IP) library. The real virtualization make the hardware target more flexible and reusable. So its can implement on FPGA cards or any other Verilog supported architecture, for example the 32-Port Programmable Switch from New Wave DV (DV, 2017).

The FPGA-based architecture chosen to target the PvS prototype was the NetFPGA SUME board. This is an FPGA-based PCI Express board with I/O capabilities for 10 and 100 Gbps operation, an x8 Gen3 PCIe adapter card incorporating Xilinx's FPGA with multiple memory and high-speed expansion interfaces (Digilent Inc., Accessed: July 2019). These attributes make SUME the ideal card for high-performance and high-density networking design.

Figure 4.1: NetFPGA SUME Board Peripherals Subsystems



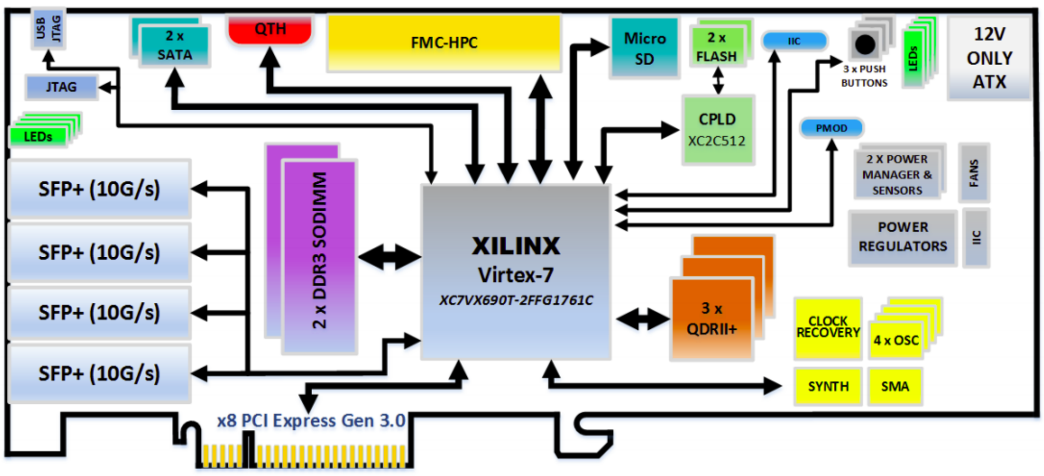
Source: (Digilent Inc., Accessed: July 2019)

The board is manufactured by Digilent Inc and implemented as a dual slot, full-size PCIe adapter, that can operate as a stand-alone unit outside of a PCIe host presented. This operation mode use a microblaze soft-core processor, e.g. (WOODRUFF et al., 2014) considerable on-chip memory that can serve for onchip cache, the peripheral devices on board as a local RAM of between 8GB and 16GB running at 1866MT/s and two hard drives connected through SATA enable the card to operate as a powerful stand-alone computing unit. The Figure 4.1 show the NetFPGA SUME and expose its peripheral.

The core of the board is powered by Virtex-7 690T FPGA device. There are five peripheral subsystems that complement. A high speed serial interfaces subsystem composed of 30 serial links running at up to 13.1Gb/s. These connect four 10Gb/s SFP Ethernet interfaces, two expansion connectors and a PCIe edge connector directly to the FPGA. The second subsystem, the latest generation 3.0 of PCIe is used to interface between the card and the host device, allowing both register access and packet transfer between the platform and the motherboard. The memory subsystem combines both SRAM and DRAM devices. SRAM memory is devised from three 36-bit QDRII+ devices, running at 500MHz. In contrast, DRAM memory is composed of two 64-bit DDR3 memory modules running at 933MHz (1866MT/s). Storage subsystems of the design permit both a MicroSD card and external disks through two SATA interfaces. Finally, the FPGA configuration subsystem is concerned with use of the FLASH devices. A block diagram of the board is provided in Figure 4.2 (Zilberman et al., 2014).

The documentation support is provided by the NetFPGA project, a collaborative effort between Digilent, Xilinx, the University of Cambridge, and Stanford University. Its goal is provides software, hardware and community as a basic infrastructure to simplify design, simulation and testing, all around an open-source high-speed networking platform.

Figure 4.2: NetFPGA SUME Board Block Desig

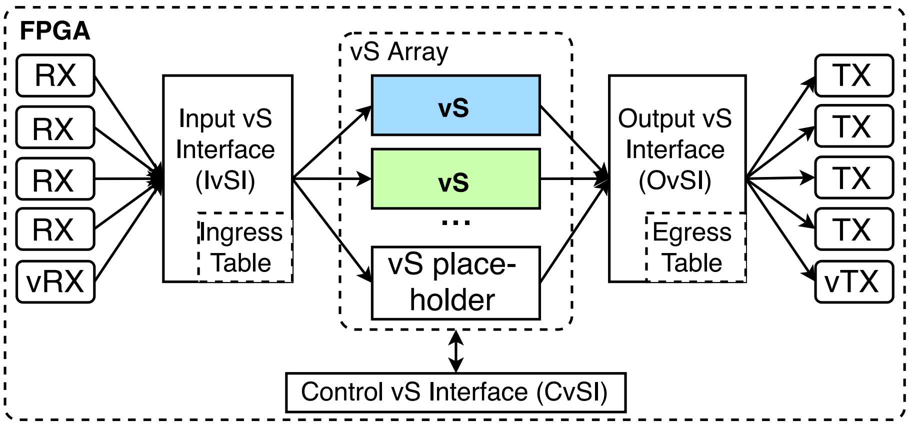


Source: (Zilberman et al., 2014)

4.1 Forwarding Engine Design Overview

The PvS forwarding engine is responsible for hosting virtual switch (vS) instances and performing flow steering between them. A conceptual view is shown in Figure 4.3. It contains five main elements: (4.2) Tx/Rx ports, which is the interface between the architecture and physical input and output layer, (4.3) Input vS Interface (IvSI), which receives incoming packets from the data plane, (4.4) vS Array, which hosts parallel vS instances, (4.5) Output vS Interface (OvSI), which outputs outgoing packets back to the data plane, and (4.6) Control vS Interface (CvSI), which provides access from the vS Array to the control engine.

Figure 4.3: PvS Forwarding Engine Architecture



Source: From the author

These modules have been developed in Verilog and implemented in the NetFPGA SUME board at 200 MHz. The communication interface with the control engine (at Host) occurs through PCI-e and USB interfaces. All design structures are based AXI4, released by ARM and later refined by Xilinx. The data flow transmission interface can be resumed as two AXI4 buses:

AXI Lite *Control & Status Interface.* The AXI Lite protocol is the control signals interface. It's basically a simple read/write interface, single beat access, with 32bit address and 32bit data. Implemented through Xilinx standardized AXI4 lite with 5 channels: read data, read address, write data, write address and write response. Components with AXI lite interface can be connected to each other via AXI interconnect module in which all peripherals reside within a slice address space. The master on this interconnect is both PCIe and microblaze, they can write and read all peripherals.

AXI Stream *Data path interface.* The AXI Stream protocol is the transit bus for SUME packets and metadata signals. This bus permeates the modules instantiated in the project sequentially conveying five main signals: `tvalid` - master to slave flow control signal, `tdata` - data signal, `tready` - slave to master flow control signal, `tlast` - indicates end of packet, `tuser` - carries meta data in the first data beat of the packet. A data path interface is implemented with Xilinx AXI4 Streaming Protocol compliant stream whereby meta-data is conveyed through the TUSER field.

4.2 Tx/Rx Ports

The Transmitter and Receiver physical ports are responsible for sending and receiving data, which can be encapsulated in different protocols, and are interpreted as data frames. The 10G interface block receives incoming serial data from the optical SFP transceiver ports and converts it to a 256-bit AXI Stream interface toward the PvS data path. It is imported from NetFPGA-SUME library and its implementation is based on the Xilinx' AXI 10G Ethernet subsystem IP. The IP includes the PMA, PCS, and MAC of the 10G interface, which operates the physical ports on a 156.25 MHz clock. This IP may include shared logic and, when this happens, this module acts as a fifth module, responsible for generating clocks and resets to all other 10G interfaces.

Data going out from the Xilinx's 10G subsystem, enters a 64-b data FIFO, and from there enters a submodule which adds metadata to every packet. The metadata is written into the TUSER field and includes the source and (default) destination port of a packet. Once the metadata is added, the 64b-wide AXI steam bus, is converter to 256b, matching the NetFPGA internal data path, and the information is forwarded toward the input arbiter (NetFPGA Project, Accessed: December 2019).

The Direct Memory Access (DMA) module provides the abstraction of extra virtual ports to interface with the Controller, each real port has a virtual port associated, allowing the architecture to load and store data frames from and into the memory. By doing so, the Controller is able to send and receive data to and from the network. Virtual ports enable the architecture to forward packets between vS instances.

4.3 Input vS Interface (IvSI)

The introduction of multiple vS instances creates the need to steer incoming flows to their corresponding deployed vS instances. To this end, the IvSI module receives packets from the RX ports (including the virtual ones) and distributes them to deployed vS instances. Every data frame that travels into the IvSI must be delivered to its correct P4 instance, otherwise the packet must be dropped. The IvSI buffers and serializes input packets through a multiplexer running in a round-robin fashion into a packet parser, which decides the vS instance to forward the packet to. In order to do so, it has to go through the frame until it finds an identifier capable of identifying the destination P4 instance, which should be encapsulated in the packet information.

The identifier can be a VLAN tag, for example, if that is the case the packet parser reads the data packet until it finds a valid 32-bit 802.1Q VLAN tag (`pkt.vlan_id`) and combines it with the packet source port (`sume.src_port`), extracted from the sume metadata (`sume_metadata_t`) in `tuser` signal, forming a tuple. The tuple is then matched against an abstraction of an `Ingress` match-action table, which can be configured by the network administrator through the control engine.

This table supports two actions: `forward` the incoming packet to a vS, or `drop` it. A `Ingress.forward` requires as parameter the `device_id`, indicating the identifier of the vS that will receive the packet. It also modify the `tuser` signal from the data communication channel to include the `device_id`, for indicating to the subsequent modules in the forwarding engine pipeline which vS a packet belongs to (to prevent packet leakage, for example). Note that, with this approach, vS instances may belong to the same VLAN, as long as they do not share the same physical ingress ports.

Therefore, this module can behave in the following ways: In case the IvSI is not able to detect the destination vS instance, it drops the frame. In case the IvSI detects that the destination vS instance is not implemented, it routes the frame to the Controller. Finally, if the destination vS instance is implemented, the packet is forwarded through an independent route that provides true switch isolation.

The IvSI interact with the virtual port abstraction, as it parses input packets before forwarding them to the corresponding vS instance. Therefore, we take advantage of the `Ingress` match-action table, originally created to extract the vS device id, and improve it to also return the corresponding virtual source port, which updates the original packet source port (`sume.src_port`) in the sume metadata (`sume_metadata_t`).

4.4 Virtual Switch Array (vS Array)

The forwarding engine provides slicing of resources through an array of placeholders for virtual switch (vS) instances (vS Array), in other words the vS array is an abstract module that corresponds to a group of placeholders for vS instances. The vS instances can be the same as the one used by the canonical reference design originated from a P4 code 2.5.4 or any other Verilog switch implementation. The architecture difference is that PvS can support as many vS as one can physically fit in the board, instead of only one.

The vS placeholders are preallocated areas with a set of predefined resources capable of implementing a single vS instance. Each placeholder contains three communication channels, to receive data packets from the IvSI, send data packets to the OvSI, and process control requests from the CvSI. They all have individual logic to implement the configuration file of a vS instance with an individual set of networking protocols, switch metadata, variable scope, and control flow. To do so, they also have a private memory space for storing match-action tables and configuration registers, which can only be accessed by the deployed vS instance. The (un)deployment of vS instances is performed through either full or partial reconfiguration (Chapter 6).

For each of the four implemented TX/RX ports, we also implement a virtual TX/RX port to allow virtual networking between vS instances. Virtual interfaces are created by connecting a virtual TX port to a virtual RX port through the DMA 4.2. By implementing virtual ports, a vS instance can create a communication channel and forward packets to other implemented vS instances.

In order to ensure non-interference between vS instances, we also adopt queues at the input/output of vS instances. By using independent vS queues instead of a single shared one we prevent, for example, that a switch handling heavy incoming traffic affects the latency or packet loss at the input of incoming traffic of other switches. Similarly, having independent queues for vS output prevents a malicious switch from saturating other switch queues and preventing them from sending packets.

4.5 Output vS Interface (OvSI)

Once the data packet is successfully delivered to its corresponding vS instance and processed, the forwarding engine must forward it to an output port. It does so through the

Output vS Interface (OvSI) module, which interfaces all vS instances' output buffers to the TX ports (including the virtual ones). The OvSI iterates over all vS instances' output buffers in a round-robin fashion, forwarding packets as soon as the corresponding TX port is available.

The OvSI also implements a memory element named Egress Table that stores a match-action table. The Egress Table is accessible by the Control Engine and has as the match input a tuple composed by the packet destination port (`sume.dst_port`), from the sume metadata (`sume_metadata_t`), the device id (`tuser.device_id`), from the data communication channel, and the VLAN tag (`pkt.vlan_id`), from the data packet. As output action, it returns the PvS physical destination port, which is updated in the sume metadata as `sume.dst_port`. The Egress Table allows the implementation of TX virtual ports on vS instances that enables network slicing.

The interaction with virtual ports abstraction 4.2 of a `Egress` match-action table. This table matches the output virtual port from which the packet is leaving the vS (from the `sume_metadata.dst_port` field), the VLAN tag within the packet (e.g., from `pkt_out.vlan_id`), and the `device_id` from the `tuser` signal. Possible actions are `forward` and `drop`. `Egress.forward` receives as parameter the physical TX port to which the packet must be sent, which is updated in the sume metadata as `sume.dst_port`.

Note that `Ingress.drop` and `Egress.drop` can be triggered to enforce vS isolation and to prevent unauthorized traffic steering from/to vS instances. For example, if a packet leaves a vS with a VLAN ID different from that configured in the `Egress` for it, PvS may drop the packet. Note also that maintaining the `device_id` information as part of the `tuser` signal prevents vS instances from modifying it. Otherwise, a vS could modify that id to that of another switch, and thus output its packets through the same physical ports assigned to that switch.

4.6 Control vS Interface (CvSI)

Place multiple vS instances in only one architecture result in a challenge to pass the control signal to each vS. The control signals should initialize the table and register to an unique switch for time, the delivery information must affect only the specific slice memory of the pertinent vS. The Control vS Interface (CvSI) is an interface between the vS Array and the Control Engine. It is responsible for delivering requests from the control engine to vS instances implemented on vS placeholders using the AXI lite protocol.

The requests from the control engine are grouped into a memory address, a data frame, and a read/write flag, the CvSI issues the request into the control communication channel shared with all vS placeholders. All requests are delivered to a corresponding vS placeholder, and therefore, the security credentials of the request are checked previously in the control engine. In this scheme the PCIe is a master in interconnect module and can send the correspondent data to vS array.

This module implement a buffer multiplexer oriented to address. When the deployment process occurs the vS receive the unique slice of memory, its begin and end space address of switch is shared with the Control Engine and stored to future access. When the control engine initialize a entry, the address called belongs to a resource threshold of the vS, so the CvSI module can access directly the pertinent vS instance memory area. The access takes place through a dedicated bus to prevent any leakage of data from occurring to any other allocated placeholder.

The board control path is composed by the microblaze system and PCIe system, both structures is a master in AXI lite protocol, so they are a master in CvSI communication. When the Control Engine needs replace, delete or check any match-action entry or register status, this is do which the Command Line Interface (CLI) accessing the switch module. The CvSI pass the address to the vS instance to request the allocated data, the answer is delivery to Control Engine using the PCIe communication channel.

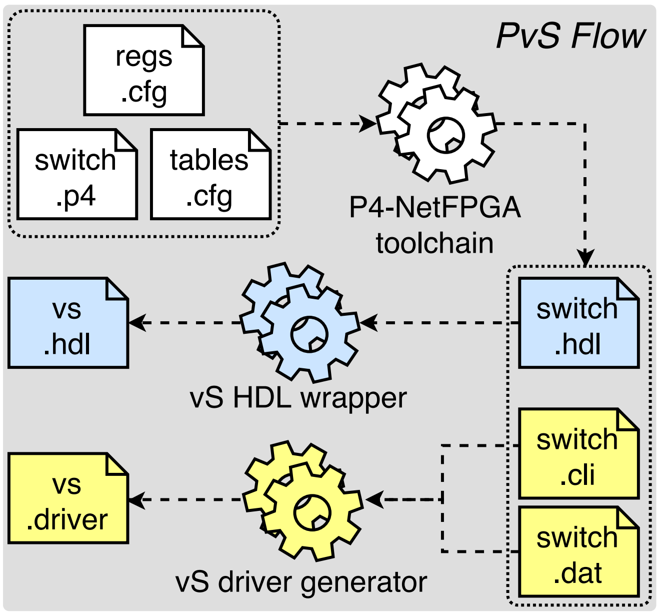
5 HIGH-LEVEL SYNTHESIS FLOW

This Chapter describes our proposed High-Level Synthesis (HLS) framework to generate a vS for PvS. This framework, which was based on the P4-NetFPGA Flow (IBANEZ et al., 2019), enables easy development of virtualized switches to be deployed in PvS, as well as supporting tools to perform simulation and testing.

The forwarding engine runs multiple vS instances in parallel which, together, form the vS Array. As previously mentioned in Chapter 4, each vS is an individual pipeline circuit implemented in the FPGA fabric with a dedicated register and memory spaces. The use of a hardware language to describe vS instances allows the deployment of a switch bytecode compiled from a program written in any DSL through this process.

The design of a vS requires the generation of two files: the vS HDL and the vS driver. The vS HDL describes the packet processing of the vS. The vS driver is used by the Control Engine for accessing the tables and registers of the switch. It is interesting to notice that the vS HDL is mandatory, but the vS driver is optional, as a vS does not require a control. Therefore, this implementation flow is able to generate a vS composed of a single wire as well as a full stack router with multiple tables and configuration registers.

Figure 5.1: High-Level Synthesis flow overview



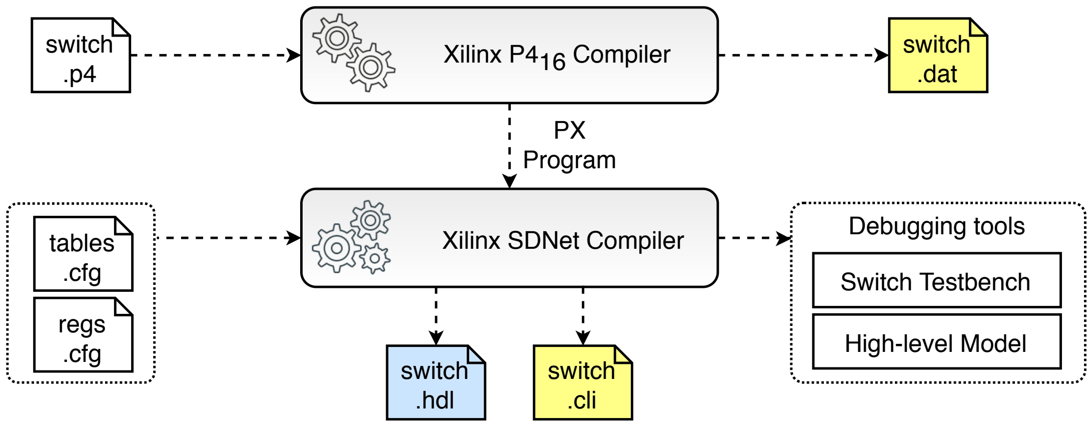
Source: From the author

To generate the vS HDL and the vS driver, our HLS flow is divided in three separate internal transformations flows: (i) P4-to-HDL HLS flow, which synthesizes a P4-described switch into a HDL-described switch, (ii) PvS wrapper generation flow, which creates a wrapper that fits the HDL-described switch into PvS, and (iii) the driver generation flow, which creates drivers and memory allocation used by the Control Engine to interface with the wrapped vS in PvS. It is important to notice that one could also design different implementation flows. For example, one could replace the P4-to-HDL HLS flow by a POF-based one or, alternatively, generate the inputs for the PvS wrapper generation flow by hand. We chose P4 as our DSL because it is currently the most used language in the literature, and therefore has a large variety of available documentation and supporting tools. An example is the Xilinx SDNet, the HLS tool to create HDL modules from P4 programs used in the canonical NetFPGA reference design (Section 2.5.4), and also in the P4-to-HDL HLS flow.

Figure 5.1 shows an overview of our proposed HLS flow, highlighting the internal flows (i-iii) in white, blue, and yellow, respectively. As one can notice, the P4-to-HDL HLS flow is the entry point, generating temporary files for the PvS wrapper and driver generator flows (ii and iii). In the following sections, each internal flow is described.

5.1 P4-to-HDL HLS flow

Figure 5.2: Xilinx P4-SDNet compilation flow



Source: From the author

The P4-to-HDL HLS flow uses procedures from the P4-NetFPGA toolchain library to generate the inputs to both vS HDL (`switch.hdl`) and vS driver (`switch.cli` and `switch.dat`) implementation flows, one for each switch in the vS array. It uses the HLS tool Xilinx SDNet Packet Processor to first translate P4 to PX and create the `switch.dat` file. It then compiles the PX program into the vS HDL and generates the `switch.cli` file.

Figure 5.2 depicts this process. The Xilinx P4 Compiler translates the input $P4_{16}$ program (`switch.p4`) into an intermediate language called PX and creates the `switch.dat` that contains information about the tables organization in the design for the Control Engine. The PX program is passed along with configuration parameters (tables and configuration registers description - `tables.cfg` and `regs.cfg`, respectively), to the Xilinx SDNet Compiler, which then produces an encrypted HDL module that implements the input $P4_{16}$ program `switch.hdl` and the set of function to manipulate the design registers through a CLI *switch.cli*. Xilinx SDNet Compiler also produces the `Switch Testbench`, a simulation environment that creates an interface able to communicate with the vS, and the `High-level Model` of the PX program that, when executed, produces a verbose output for debugging, capable of describing how the packets are processed.

At this point, the developer can use the debugging tools to simulate the HDL module behaviour and ensure the processing capacity of the design. To do so, a python script is used to generate custom network flows with varying number of packets to be applied by simulation. Using the packet manipulator methods of the Scapy library (BIONDI, 2010), the network data generator creates four files: (1) a set of packets to feed the vS (`src.pcap`), (2) the expected output of the vS (`dst.pcap`), (3) the metadata associated with each packet in `src.pcap` (`Tuple_in.txt`), and (4) the metadata associated with each packet in `dst.pcap` (`Tuple_out.txt`). Additionally, one could create simulation files for the board containing multiple vS instances. The next section covers more details about the SUME simulation.

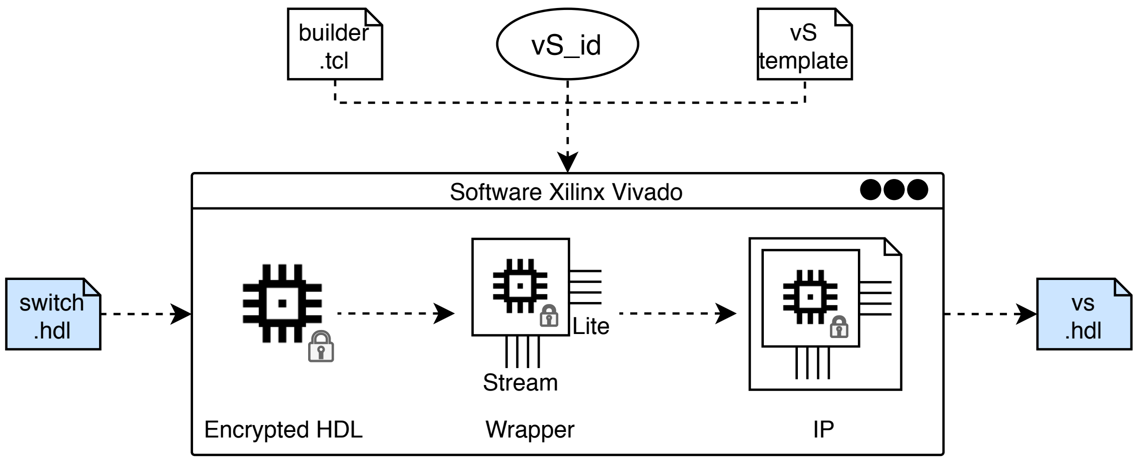
Once finished, the HLS flow results in an encrypted hardware description, so this process can be carried out independently of the platform design and the developer guarantees the protection of its intellectual property. This feature allows developers to create vS instances and deliver the bytecode to a third party, which can then integrate it with the hypervisor design platform. Despite the apparent complexity, all this process is guided by an online command tool and has transparent steps for the developer to facilitate its usage. The Appendix A shows an example of commands to execute the HLS flow from P4 to HDL of a project with three vS instances.

5.2 PvS Wrapper Generation Flow

The PvS wrapper generation flow consists of wrapping a generic HDL-described switch (`switch.hdl`) with data and (optionally) control communication channels. To do so, the vS HDL wrapper parses the switch.hdl, inserts I/O communication interfaces compatible with a vS placeholder template, and creates the vS HDL IP file, which can then be translated into a target board configuration file. The vS placeholder template exposes an AXI Stream interface for packet data traffic, an AXI Lite interface for communication with the control. The former takes the headers and payload of the package for processing, as well as the associated metadata, while the latter is used to initialize and change the registers associated with the vS tables during processing.

Our generation flow takes advantage of the SSS model, available from the P4-NetFPGA project, which supports the sume_metadata, extending it to make each vS unique, to do so, a unique switch identifier (`vS_id`) is assigned to the `Wrapper` through the template file (`vS template`) during the IP building process. Currently, the identifier used is a number ranging from 0 to the total number of switches embedded in the design of the PvS Forwarding Engine, normally associated with the VLAN tag. If the vS was developed by a third party, it is up to the design that will implement the project to adjust the vS identifier so that there are no conflicts in the architecture.

Figure 5.3: Building a vS IP



Source: From the author

The construction of the IP is performed through a script in Tool Command Language (TCL) that runs on top of the Xilinx Vivado software. It uses the folder resulting from the switch compilation and simulation process `switch.hdl` as input, defines the unique vS identifier and then calls this script. In turn, the IP builder applies the vS placeholder template, creates the IP (vS.hdl) and links it to the user library in Vivado, for later added to projects. Figure 5.3 summarizes this step in the HDL Wrapper flow.

The SUME simulation adds the vS IPs available in the library and builds the architecture design around them, including the modules covered in the Chapter 4 to support virtualization. In addition, the standard I/O port modules and a simple control module are instantiated to provide communication channels to packet data and control commands. The embedded control enables simple reading and writing functions in registers to enable communication via PCI-e with the vS array through the CvSI, which in turn uses the communication channel formed by AXI Lite.

Finally, the files applied for each of the four simulated network interfaces generated during P4-to-HDL flow are applied in the SUME simulation. The register structure that represents the switch tables is initialized and the architecture is stimulated with all vS running in parallel. With the end of the execution, the output packages are compared to the expected packages.

5.3 Driver Generation Flow

The driver generation flow consists of four steps that, together, generate all drivers for PvS (`vs.driver`). The first step is to generate the standard NetFPGA SUME board driver [1]. This driver is used for reading and writing board registers via PCI-e at runtime. The second step is the generation of functions used to access multiple CLIs. This driver is derived from all individual switch CLIs. The thirty driver is a map to global memory to guide the implementation flow to place the vS instances address in design. The final step generates drivers for the board reconfiguration wizard, that enables the initialization of multiple switch tables and reprogramming capabilities.

The stardard NetFPGA SUME board driver is based on the DMA, which supports PCI-e Gen.2 x8 as well as packet transactions. It receives incoming transactions (both packets and registers access) from the PCI-e and converts them to AXI-Stream transactions towards the data path (for packets) and AXI-Lite transactions toward the control path (for registers access) (AUDZEVICH, 2015). In other words, in addition to accessing the registers, it is possible to insert packages directly through the PCI-e. Such functionality is used to support packet-in and packet-out delivering messages to

[1]https://github.com/NetFPGA/NetFPGA-SUME-public/wiki

the Control Engine. To do so, the Forwarding Engine, upon receiving a packet with `sume_metadata.dst_port= 0x10000000`, using the OvSI to copy the packet for the reserved virtual port number three (`nf3_dma`), realizes the packet-out when sending a data plane packet for the controller. In the packet-in message, a reserved register is written with the `device_id` with the virtual port from which the packet was received and the OvSI routes the packet to its corresponding vS.

The second step creates an individual CLI for each vS. It generates functions to improve the `switch.cli` using information about the switch structure from `switch.dat`. To do so, a new environment variable is declared to define a property relationship between the vS instance and the respective CLI, which is then passed as a parameter during the CLI building process to the basic `switch.cli`. This process extracts the switch structure pipeline from the `switch.dat` and generates functions to access data in match-action tables and registers. It also describes a memory allocation requirement for the CLI's memory space, used in the next step.

The third step is responsible for producing the address map drivers, which are guided by the memory allocation requirements informed in the previous step. The memory allocation requirements are collected, adjusted with the SUME register structure, and distributed in slices of memory, forming a global memory allocation structure. The slices are based on the number of vS tables, so that the memory space allocated to each vS fits twice the size of its initial tables. This process creates an address map of all vS instances, containing the start and length of the initialized match-action tables in the global memory.

The fourth and final step generates drivers for the board reconfiguration wizard. This feature needs a the map of the vS Array, created in the previous step, and the initial values of the tables from the vS instances to perform the first write in the board register after the reconfiguration. In addition, the wizard realizes full or partial reconfiguration using the bitstream generated from the PvS design implementation. Environment variables define the type of reconfiguration and which bitstream that should be used by the wizard to reprogram the FPGA.

6 RECONFIGURATION METHODOLOGY

The current ossified state of the Internet imposes many barriers to meet the constant increase in demand and improvement of services already offered, aiming at a more sustainable model in the long term, the network system designers are exploiting the programmability to add new features to the forwarding plane, including telemetry (KIM et al., 2015), layer-4 load balancing (MIAO et al., 2017), encryption, and in-network caching (JIN et al., 2017). The new breed of programmable switches and NICs matching the performance, power, and cost of fixed-function devices (BROADCOM, 2018; CAVIUM, 2018; BAREFOOT, 2019). Among the benefits of using programmable hardware such as FPGAs is the flexibility and re-usability provided by the reconfiguration.

The reconfiguration of PvS can be achieved by means of full or partial board reconfiguration. Full reconfiguration downloads a bitstream that replaces whole FPGA logic with a new one, erasing previous configuration. Partial reconfiguration downloads a partial bitstream to the FPGA, overwriting only a predefined part of the full logic, without interfering with other parts. Both methodologies have advantages and disadvantages, which must be tailored to different scenarios. The Forwarding Engine takes advantage of reconfiguration to enable vS hot-swapping. In other words, this feature allows tenants to (un)deploy switch bytecode from/to a programmable vS.

Both full and partial reconfiguration methodologies start the procedure by programming the board with a standard full configuration bitstream. However, while the full reconfiguration methodology already programs the board with the final design, the partial one programs the board with a blank template, capable of later receiving partial modules. The following sections explain our proposed methodologies and present an evaluation of the best scenarios for applying each of them.

6.1 Full Reconfiguration Methodology

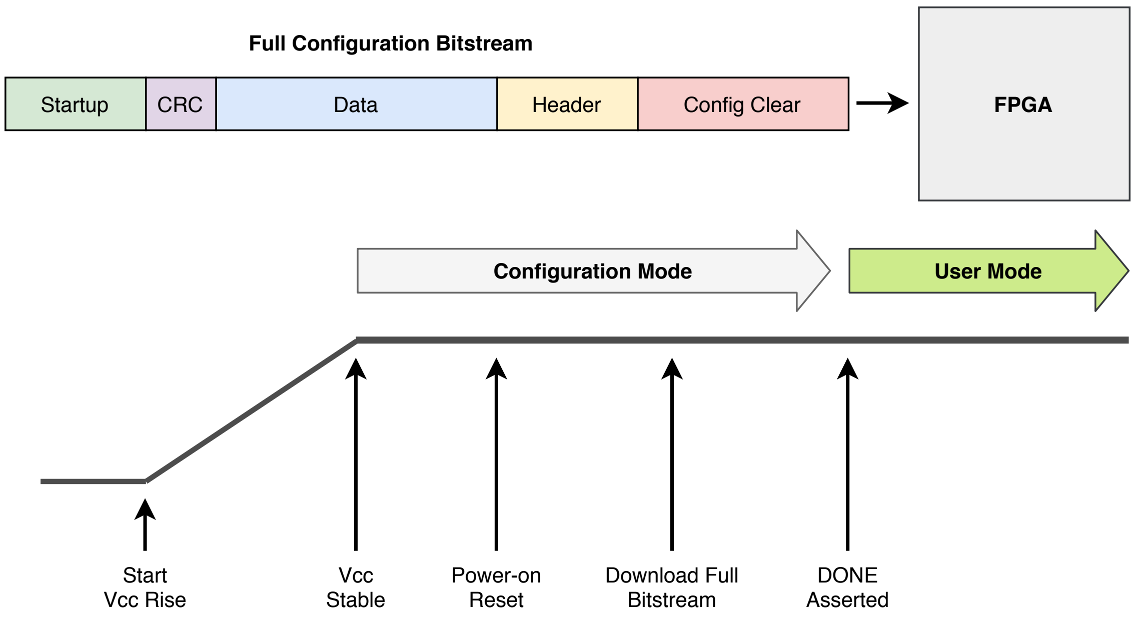
The full reconfiguration process downloads a full bitstream file, overwriting the current FPGA board configuration. The bitstream download can be done directly from an internal PROM memory or from an external general-purpose memory. This process is carried on by a microprocessor, which configures the FPGA board. A full bitstream is a file that contains all the information necessary to reset and configure the FPGA fabric with a complete design.

Our full reconfiguration methodology implementation includes three steps, which happen after the HLS Flow, described in Chapter 5, is performed: (i) full bitstream generation, (ii) full bitstream download to the FPGA board, and (iii) initialization.

The first step is responsible for grouping all vS instances, which are already wrapped to the PvS architecture and generating the full bitstream, describing the whole FPGA fabric. This is done in Xilinx Vivado through a set TCL scripts that build the virtualized design. This process is transparent for the user, as shown in appendix A.3. At the end of this step, the full bitstream file is ready to be downloaded to the board using the set of features provided by driver generator flow (5.3).

The second step, which downloads the full bitstream to the FPGA, is illustrated in Figure 6.1. It waits for a stable Vcc and then issues a global reset that erases all contexts across the FPGA board, including PvS and vS configurations, such as tables and configuration registers. After, a microprocessor downloads the bitstream from the source memory to the target FPGA board. The download process is then verified by performing a bitstream readback from the FPGA and, if the verification passes, a DONE signal is asserted, changing the configuration mode to user mode.

Figure 6.1: FPGA Configuration with Full Bitstream



Source: Author adaptation from (XILINX, 2019)

The third and final step starts in the user mode, where the FPGA is correctly programmed with the full bitstream. It then loads the drivers generated by the driver generation flow into each vS instance. In other words, this step initializes all vS instances with their respective registers and memory spaces and restarts the whole vS array.

This approach synthesizes and implements the vS instances embedded in the vS array. Consequently, the synthesis tools are able to better optimize the design, achieving better area occupation (more vS instances can fit in the board) and improved operating frequency (improved latency and throughput). On the other hand, the full reconfiguration takes longer to perform because reconfiguration times are dependant on configuration file sizes, and full bitstream files are as larger than partial ones.

6.2 Partial Reconfiguration Methodology

The partial reconfiguration process is supported by the newer FPGA devices, it allows the modification of a fraction of the FPGA fabric, instead of the whole FPGA. This process also commonly applied dynamically, being able to reconfigure the FPGA partially while the design is running in user mode. As mentioned previously, partial reconfiguration starts by programming the FPGA with a full bitstream containing a template for dynamic partial reconfiguration, following all steps from the full reconfiguration methodology.

After a full bitstream file configures the FPGA, partial bitstream files can be downloaded to modify reconfigurable regions in the FPGA, without compromising the integrity of the applications running on those parts of the device that are not being reconfigured. To do so, the logic in the FPGA design is divided into two different types: reconfigurable logic and static logic. The static logic is the fixed part of the design which remains functioning and is unaffected by the loading of a partial bitstream file. The reconfigurable logic is the dynamic part of the design, which can be replaced by different partial bitstream files.

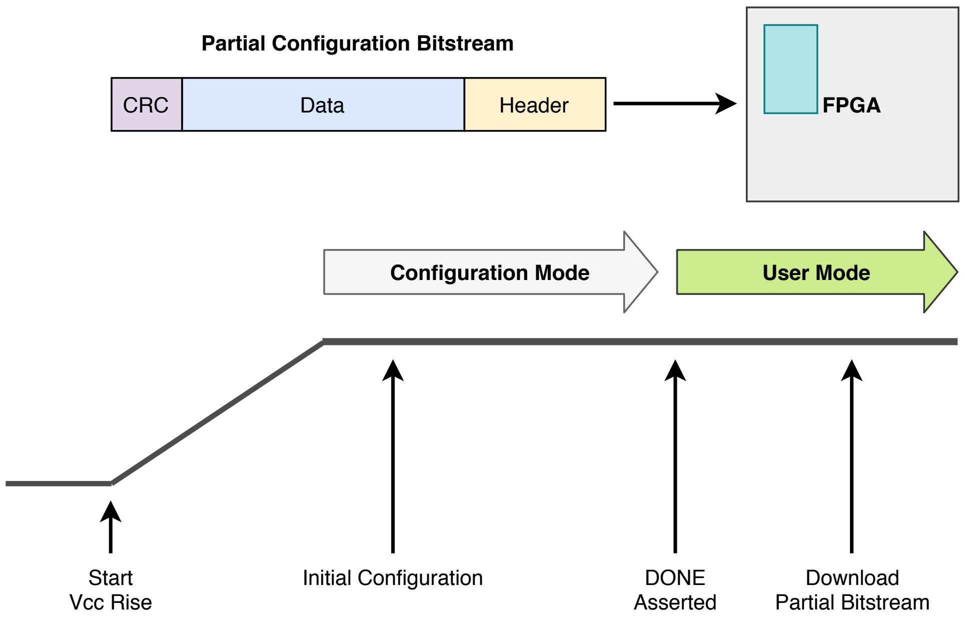
The static logic is generated following the full reconfiguration methodology with two adaptations: (i) a set of reconfigurable areas are defined and statically allocated in the FPGA fabric, and (ii) vS instances are marked as reconfigurable modules, enabling multiple vS instances that can be replaced in a single existing reconfigurable area. At the end of these steps, the static full configuration bitstream is generated and downloaded to the target FPGA board. It is interesting to notice that step (ii) from the full reconfiguration methodology is not necessary, as there are no vS instances to be initialized.

Following the download of the static logic in to the target FPGA, the partial reconfiguration methodology has three more steps to follow for each new vS: (i) partial bitstream generation, (ii) partial bitstream download to a reconfigurable area, and (iii) initialization.

The first step is responsible for creating partial bitstreams files for all vS instances. As reconfigurable areas might have different sizes and available resources, a single vS must have a specific partial bitstream file for each target reconfigurable area. Therefore, a fixed design with four reconfigurable areas can have up to four different partial bitstreams for each vS. It is also important to notice that a vS might not fit in a specific reconfigurable area, and this information must be taken into account during this step. Partial bitstream files are created in the same process as the full bitstream, but with area constraints. At the end of this step, a set of partial bitstream files is generated for each vS.

The second step downloads partial bitstream files to the FPGA during user mode. This process is illustrated in Figure 6.2. The partial bitstream file no has a startup sequence, because partial reconfiguration can only be performed in user mode. Therefore, the bitstream file contains a simplified header, frame address of the reconfigurable area, and configuration data of the vS. After downloading the partial bitstream file to the FPGA board, it remains in user mode, only returning to configuration mode when a new full reconfiguration is performed.

Figure 6.2: FPGA Reconfiguration with Partial Bit File



Source: Author adaptation from (XILINX, 2019)

As soon as the partial bitstream file is downloaded and checked for consistency, the final step can be performed, where the hot-swap is adjusted - i.e. the replacement of one vS instance with another through the dynamic partial reconfiguration. To do so, the initialization process first clears the memory range used by undeployed vS, and then load the table values. The memory clear is performed to avoid security breaches. The vS driver of the reprogrammed vS is then loaded, initializing its memory space (registers and tables).

Partial reconfiguration methodology improves full reconfiguration by allowing the dynamic change of vS instances on the hypervisor, without affecting the remaining modules on the board. In addition, as partial bitstreams are smaller than full ones, it is also faster to be performed. On the other hand, the restrictions in sizing of reconfigurable areas constraints the synthesis tool to place and route vS instances in a reduced area, with fewer resources, and therefore fewer optimization options across reconfigurable areas boundaries. These constrainings lead to waste of resources, as the unused resources from a reconfigurable area cannot be used by other instances, that means that the overall density and performance is lower for a partial reconfiguration design than for the equivalent design generated with the full reconfiguration methodology.

7 EVALUATION

This section substantiates our implementation claims by: (i) evaluating resource usage and performance, (ii) analyzing the proposed reconfiguration methodologies through resource allocation and reconfiguration time, (iii) measuring the impact of vS management, and (iv) discussing the in-band telemetry case-study. We prototyped PvS in the NetFPGA SUME board at 200 MHz operating frequency and, to serve as baseline, we also synthesized the P4-NetFPGA canonical reference design (IBANEZ et al., 2019) atop our FPGA, simulated the native P4 behavioral model version 2 (bmv2) (P4LANG-CONSORTIUM et al., 2019), HyPer4 (HANCOCK; MERWE, 2016), and HyperVDP[1] (ZHANG et al., 2019), and used data obtained from P4Visor (ZHENG; BENSON; HU, 2018).

Our experiments run PvS with four vS instances in parallel: a layer-2 switch (L2), a router, a firewall, and an in-band telemetry (INT). All case-study switches are from the state-of-the-art and have been generated with the complete High-Level Synthesis Flow (Chapter 5). We separate traffic between vS instances using VLAN tagging.

7.1 Resource Utilization and Performance

Here we discuss FPGA resource utilization and operating frequency, as well as three scenarios to evaluate performance in terms of throughput and latency: (i) simulated performance, where we simulate PvS, (ii) port saturation performance, where we deploy PvS and generate a packet flow that saturates one 10 Gbps RX port through multiple TCP connections, and (iii) FPGA saturation performance, where we deploy PvS and generate a loopback packet flow capable of saturating all PvS 10 Gbps ports.

7.1.1 Resource Utilization and Operating Frequency

Our implementations resulted in a maximum operating frequency of 547.6 MHz for both PvS and P4-NetFPGA. This value is directly related to the outgoing buffers of the vS instances, where the critical path resides. We confirmed these measurements by analyzing timing reports and testing a network hub described in HDL and implemented

[1]HyperVDP source code was not fully available by the time this dissertation was written. As a result, we could not run the benchmarks with HyperVDP for a fair comparison. For this reason, we omitted it from part of our analysis.

through the PvS wrapper generation flow (Section 5.2). Modifications to these structures could increase operating frequency, therefore improving overall throughput and latency.

In terms of area occupation, Table 7.1 shows Look-Up Table (LUTs) and Registers usage by PvS and P4-NetFPGA architectures without switch pipelines, as well resources used to implement the case-study switch pipelines individually. As one can see, both PvS and P4-NetFPGA architectures have resource usage comparable to a single switch instance, where PvS requires around 1.1% more resources than P4-NetFPGA, at 11.4% of the available board LUTs to provide the virtualization modules. Comparisons to the related work was not possible due to the lack of area occupation data.

Table 7.1: NetFPGA SUME resource usage

Module	LUTs		Registers	
PvS	48471	(11.4%)	74338	(8.6%)
P4-NetFPGA	44447	(10.3%)	67926	(7.8%)
L2 switch	28096	(6.5%)	40506	(4.7%)
Router	57704	(13.3%)	84149	(9.7%)
Firewall	45683	(10.5%)	74344	(8.6%)
INT	70011	(16.2%)	147851	(17.1%)

Source: From the author

PvS uses memory to store data packets in the I/O buffers and for storing vS match-action tables. As the number size of I/O buffers can be tailored to different scenarios, we present Table 7.2, which shows the number of match+action tables required for each case-study switch. Observe that PvS requires the same number of tables per switch as in bmv2, whereas HyPer4 and HyperVDP decouple match-action stages into a combination of several tables.

Table 7.2: Match-action table usage

Switch instance	bmv2	HyPer4	HyperVDP	P4-NetFPGA	PvS
L2 switch	2	13	5	2	2
Router	4	28	16	4	4
Firewall	3	22	8	3	3
INT	5	35	23	5	5
L2+Router	–	–	–	–	6
L2+Router+Firewall	–	–	–	–	9

Source: From the author

7.1.2 Scenario 1: Simulated Performance

In this scenario, we simulated PvS running at the theoretical maximum operating frequency of 547.6 MHz, rather than the clock available in NetFPGA SUME, which is 200 MHz. We measured throughput and latency by injecting packet loads emulating a datagram packet flow (512x 256-byte packets) and an ICMP request (2x 64-byte packets), respectively. For both packet flows, we measured the number of clock cycles between the arrival of the first packet and the completion of the last. We also simulated P4-NetFPGA and considered results from P4Visor.

Table 7.3 presents achieved throughput and latency. When considering throughput, PvS achieved around 100 Gbps, which translates to 37 Gbps at the deployed 200 MHz operating frequency. This value is enough for feeding four 10 Gbps ports and could be increased if needed. When compared to P4Visor and P4-NetFPGA, it is around 25 times higher than P4Visor and around 20% smaller than P4-NetFPGA. Latency showed around 0.8 μs, an increase in 8% when compared to P4-NetFPGA. Such results show that PvS is able to deliver enough latency and throughput to up 10x 10 Gbps ports at maximum operating frequency, but still performs slightly worse than P4-NetFPGA, showing the performance cost of implementing virtual parallel switch instances.

Table 7.3: Simulated throughput and Latency

Switch instance	Throughput ($Gbps$)			Latency (μs)		
	P4Visor	P4-NetFPGA	PvS	P4Visor	P4-NetFPGA	PvS
L2 switch	–	121.8	101.1	–	0.52	0.56
Router	4.6	119.6	100.2	109.1	0.79	0.84
Firewall	–	114.8	99.8	–	0.76	0.81
INT	–	113.7	98.8	–	0.86	0.92
L2+Router	4.4	–	118.3	–	–	–
L2+Router+Firewall	4.4	–	115.2	–	–	–

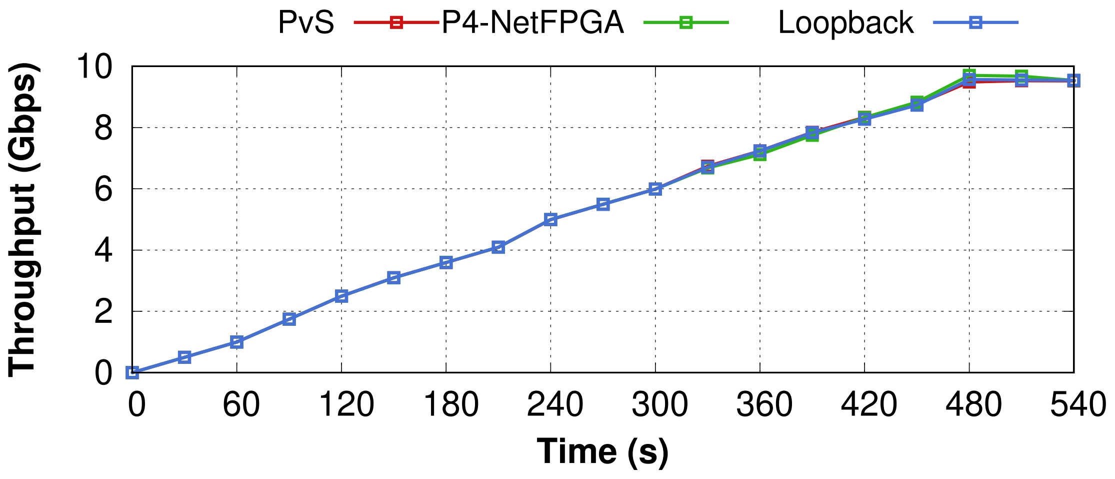
Source: From the author

7.1.3 Scenario 2: Port Saturation Performance

To saturate a single PvS port with multiple connections, we deployed sixteen virtual machines, each with a 1 Gbps network interface generating packet flows to an OpenFlow switch. The OpenFlow switch then either loopbacks or forwards to PvS the added bandwidth. In case of PvS forwarding, PvS processes the packet flow in a given vS instance and forwards it back to the OpenFlow switch, which then distribute the packet flow back to the paired virtual machines. The communication between the OpenFlow switch and PvS goes over a 10 Gbps Optical Ethernet port, and each 10 Gbps port in the OpenFlow switch is mirrored to a 1 Gbps port connected to the host computer. We generated packet flows and measured throughput with iperf3 through up to sixteen TCP connections.

Figure 7.1 shows throughput for the loopback, PvS, and P4-NetFPGA. Evaluated data shows that applications' throughput increased for each new TCP connection with less than a 2% difference up until saturation point, where they slightly differed due to TCP congestion control converging to a fair bandwidth share between flows. All applications stabilized throughput at 9.5 Gbps.

Figure 7.1: Port saturation throughput measurement



Source: From the author

7.1.4 Scenario 3: FPGA Saturation Performance

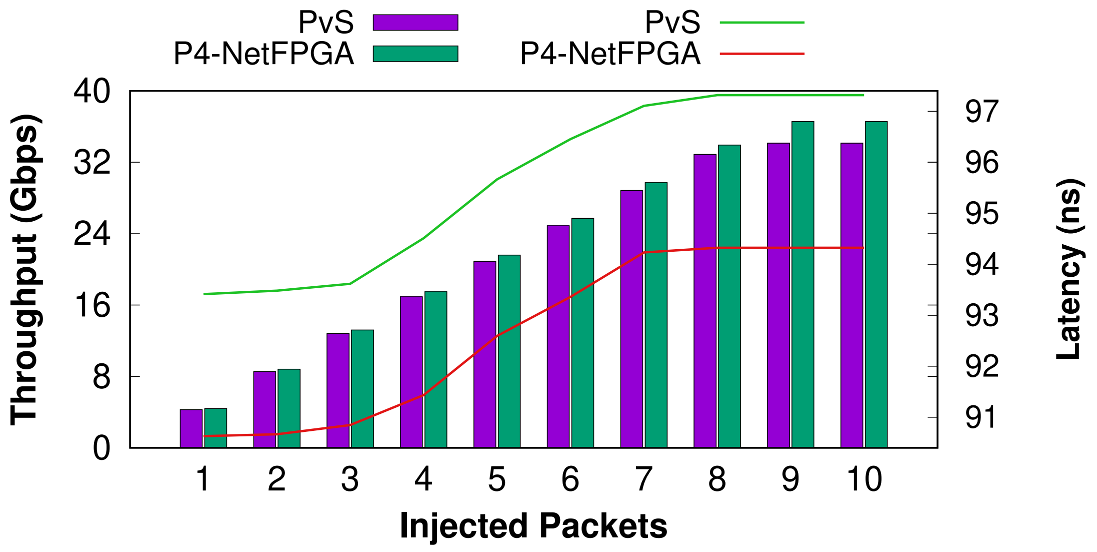
To measure FPGA saturation performance, we developed a HDL packet counter module and deployed it with PvS. The module was implemented to count all packets passing through the OvSI module of PvS spaced one second. At the end of each second, results were stored in an internal OvSI register and the counter reset. We also implemented internal register access through the CLI.

Having PvS deployed with four vS instances (two l2 switches and two routers) and the packet counter, we connected all four 10 Gbps I/O ports forming four looped pairs (TX_0/RX_1, TX_1/RX_0, TX_2/RX_3, and TX_3/RX_2), and configured all packets to undergo a single vS instance, forming four loopback packet flows.

To evaluate throughput and latency, we performed the following experiment: (i) inject a single 256-byte packet in all ports, (ii) read the packet counter, and (iii) repeat until FPGA saturation. Throughput is constantly measured by the packet counter by multiplying the number of packets per its size (2048 bits), and latency can be measured until port saturation by dividing the number of inserted packets by the throughput. It is important to notice that we disabled packet TTL verification in order to keep packets in the loopback.

Measured FPGA saturation performance can be seen in Figure 7.2 for PvS and P4-NetFPGA. It shows that PvS saturates throughput at 34.1 Gbps, 6.7% less than P4-NetFPGA. When considering latency, PvS showed an average of 1.8 ns more latency than P4-NetFPGA. PvS cannot achieve the same throughput and latency mainly because it implements four virtualized modules, but steers traffic to a single one. Therefore, the OvSI keeps iterating over four output buffers, whereas only one has data.

Figure 7.2: FPGA saturation throughput and latency



Source: From the author

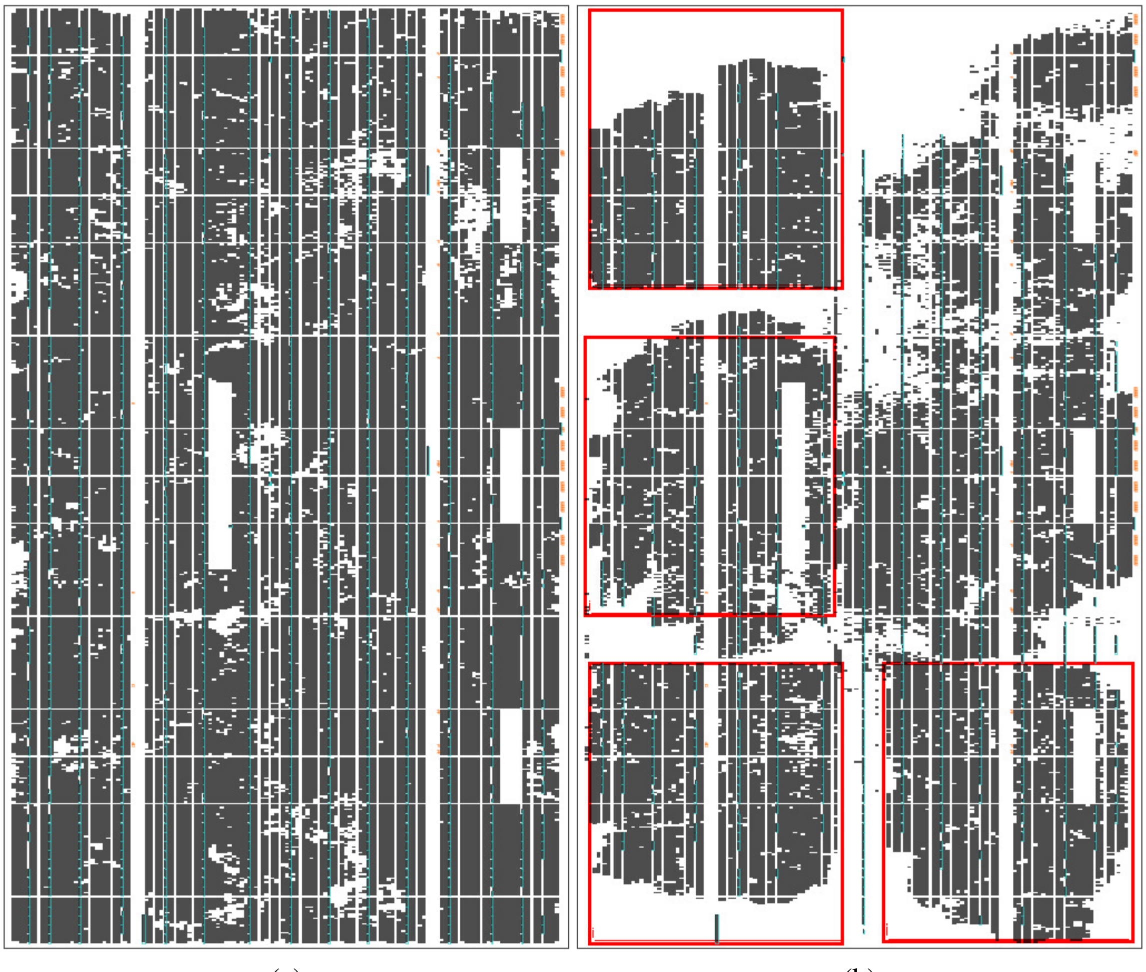
7.2 Reconfiguration Analysis

This section describes an analysis on the two proposed reconfiguration methodologies, considering the maximum resource allocation for designs with no performance degradation, and its reconfiguration times.

7.2.1 Resource Allocation

Figure 7.3 shows the maximum resource allocation for full and partial FPGA reconfiguration examples for the same SUME NetFPGA. Figure 7.3a shows a snapshot of a full reconfiguration example containing eight layer-2 switch instances, while Figure 7.3b shows the same layer-2 switch instances, constrained to 4 reconfigurable areas (highlighted in red). More vS instances could be allocated by porting PvS to a larger FPGA, such as the Programmable Network Switch FPGA-based 32-Port (2017), which embeds a Xilinx Ultrascale+ FPGA.

Figure 7.3: (a) Full reconfiguration. (b) Partial Reconfiguration.


(a) (b)

Source: From the author

In our experiments, the full reconfiguration setup showed no decrease in performance, while the partial reconfiguration showed decreases of 65% and 71% for three and four reconfigurable areas, respectively. In terms of occupation, the full reconfiguration was limited by the available onboard memory, which could be tailored to fit more layer-2 switch instances. The partial reconfiguration was limited by the sizing and placement of the reconfigurable areas, which could not occupy global resources, such as the PCI-e pins.

7.2.2 Reconfiguration Times

Reconfiguration speed is directly related to the size of the configuration file and the bandwidth of the configuration port (XILINX, 2019). Our generated bitstreams, which have been compressed by the synthesis tool, size around 14.6 MB (full) and 4.3 MB (partial). Our measurements have been performed through the USB-JTAG configuration port operating at 66 Mbps, on the host, and resulted in around 8 and 4 seconds for full and partial configuration files, respectively.

It is important to notice that, as our measurements have been performed on the host, there are hidden overheads related to the system and, therefore, one could optimize reconfiguration times on the same JTAG-USB down to 1.8 and 0.5 seconds. Additionally, one could store the configuration files in the onboard parallel flash memory and use the SelectMap interface (3.2 Gbps) to achieve configuration times around 36 and 11 milliseconds. Finally, newer boards support MCAP configuration ports operating at 6.4 Gbps over the PCI-e, which could further halve reconfiguration times to 18 and 6 milliseconds.

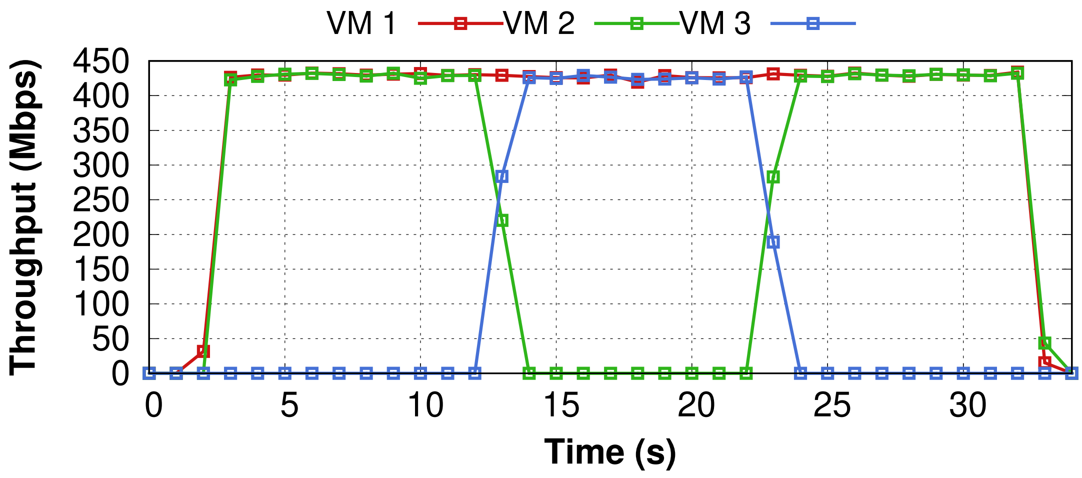
The full reconfiguration enables better resource allocation, but it takes longer to download the bitstream. Hence, it is recommended for scenarios where all vS instances fit on target board or few changes are made during operation, since packet loss and context loss impact other running vS instances. Context loss can be mitigated by the Control Engine through context snapshotting, but packet loss cannot. For all the other scenarios, partial reconfiguration is recommended due to the flexibility offered by the hot-swap and the low reconfiguration times.

7.3 Impact of vS Management

To measure the impact of vS management, we deployed PvS and started three virtual machines. The first one (VM1) generated a bandwidth of 100 Mbps to PvS. Initially we configured PvS to steer all incoming traffic to the second virtual machine (VM2). Then, after 10 seconds, we issued a CLI to update the match-action table and forward the incoming traffic to the third virtual machine (VM3). Finally, after another 10 seconds, we issued a second CLI to update the match-action table again, steering traffic back to VM1.

Figure 7.4 shows the throughput on the virtual machines, as well as the issue of the CLI. As one can see, throughput remained stable during the table update, and the CLI interaction with FPGA required around 0.2 seconds to take effect. We measured around 909 packets lost during traffic steering.

Figure 7.4: vS management impact



7.4 Use Case: In-Band Telemetry

We also used PvS to run a vS implementation that performs in-band telemetry. The main takeaway from this experiment is that we are able to deploy and run vS instances that perform in-band telemetry with a performance comparable to that of the switch running directly on the NetFPGA board, with the existing vS constructors.

Observe from Table 7.1 that our implementation of in-band telemetry uses 16.2% of LUTs and 17.1% of registers available. These results are around 21% and 75% higher than those observed for the router deployment (second more costly observed). With regard to match-action stages used, Table 7.2 evidences that running an instance of in-band telemetry on top of PvS is significantly more efficient than doing it over Hyper4 or HyperVDP.

8 CONCLUSION

The possibility of programming the forwarding plane has opened up several interesting possibilities for in-network computing (TOKUSASHI et al., 2019). As virtualization of programmable forwarding plane takes off, we may soon witness the emergence of *Programmable Switches as a Service* (KRUDE et al., 2019). However, realizing that vision requires decoupling the programmable virtual switch (vS) from the hypervisor, as well as enabling switch vendors and developers to distribute switch bytecode while preserving their intellectual property. In this work we presented PvS forwarding engine slicing, a architecture that provides the abstraction of programmable vS instances that satisfies those requirements while ensuring tenant isolation and delivering a performance similar as if the vS were running directly on bare-metal.

This work proposed a hypervisor for data plane capable of running multiple vS instances in a single hardware substrate with performance comparable as a single switch running. This solution enables the network operators to deploy vS instances directly from compiled switch bytecode protecting the intellectual property. The methodology for full and partial reconfiguration allows two distinct hardware utilization: a more static or dynamic model, with the switch hot-swapping feature without causing interference to other running instances.

A proof-of-concept was implemented based on the SSS model for a NetFPGA SUME target environment. The experiments run PvS with four virtual vS instances in parallel: a layer-2 switch, a router, a firewall, and an in-band telemetry. All case-study switches have been generated with the complete P4-to-HDL implementation flow (5). We segment vS packet flows using VLAN tagging.

The results provided evidence of the possibility to achieve near line-rate performance for switch instances running on top of a hypervisor, while satisfying typical virtualization requirements like hot-swapping and context and resource isolation. Implementation were able to advance the state-of-the-art in terms of bandwidth and latency and provide performance comparable as a single switch running on hardware.

8.1 Future Work

For future work, it is intended to achieve the following goals:

8.1.1 Network Segmentation

We currently support VLAN tagging as a mechanism to isolate traffic flows between switches. Extending PvS to support VXLAN or another user-defined tagging mechanism requires making the Forward Engine `Ingress` and `Egress` programmable. It is intended to investigate how to make their structure defined by a custom parser and match-action stage written in a DSL like P4, so that the network developer can specify its own match-action tables to separate traffic flows between vS instances, and therefore accommodate an arbitrary and user-defined flow tagging scheme.

8.1.2 Performance Guarantees

There is a trade-off to be considered when designing the switch queues. A design based on shared queues optimizes memory resource usage and opens up space for deploying more vS instances. However, such may cause a vS to impact the operation of other switches. In the case of a shared ingress queue, a switch handling elephant flows may increase the latency of traffic forwarded to other vS instances. In the case of a shared egress queue, suppose a malicious switch flooding the queue with fake output packets. That switch would quickly occupy the shared queue, and prevent other vS instances from forwarding traffic. An alternative design is allocating a dedicated egress queue for each vS. Although such design requires extra memory space for implementing the queue, it enables providing more strict performance and isolation guarantees to vS instances.

8.1.3 Security Implications

The forwarding engine slicing guarantees that vS instances only have access to their individual memory spaces. The control engine, on the other hand, has one single entry point: the Platform API. The platform API access the forwarding engine through the SUME driver and vS drive. The first has access to all tables and configuration registers on the board, while the second is limited to its deployed vS instance, through the CvSI module. The forwarding engine does not perform any security checking of control requests, always delivering the request to the addressed memory space. Therefore, the control engine is responsible for checking security credentials of all control requests.

Currently, the control engine only has the start address of each table from a vS, as contained in the `switch.cli` module generated by the P4-NetFPGA toolchain. The `cam_add_entry` does not have a way to know when a table is full, for example. Therefore, we need an external mechanism to prevent match-action table overflow.

8.1.4 Community Development

While this prototype demonstrated the feasibility of a programmable vS hypervisor, it opens up several research avenues for future investigation. Is it possible run multiple programmable vS instances in an array of NetFPGAs and support their management through a single control engine abstraction? Is it viable we use Remote Direct Memory Access (RDMA) to enable vS instances running in different NetFPGAs to share context for stateful packet processing and fine-grained in-band telemetry (KIM et al., 2016)? Is it possible offer the abstraction of predictable virtualized ports to vS instances (KUMAR et al., 2019), with distinct performance guarantees per tenant and per vS port? How can future hardware switches support programmable vS hypervisors? As a public and open source implementation, it is intended to develop a community of researchers and practitioners around PvS to understand the implications of these questions (and others that follow) on the design of a fully-fledged hypervisor for programmable forwarding planes.

REFERENCES

AFOLABI, I. et al. Network slicing and softwarization: A survey on principles, enabling technologies, and solutions. **IEEE Communications Surveys & Tutorials**, IEEE, v. 20, n. 3, p. 2429–2453, 2018.

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

AUDZEVICH, R. T. Y. **RIFFA DMA Engine**. [S.l.], 2015. Disponível em: <https://github.com/NetFPGA/NetFPGA-SUME-public/wiki/RIFFA-DMA-Engine>.

BAREFOOT. **Tofino 2**. [S.l.], 2019. Disponível em: <https://www.barefootnetworks.com/products/brief-tofino-2/>.

BHASKER, J. **A Vhdl Primer**. [S.l.]: Prentice-Hall, 1999.

BIONDI, P. **Scapy documentation**. [S.l.]: Release, 2010.

BITTMAN, T. J. et al. Magic quadrant for x86 server virtualization infrastructure. **Gartner, June**, 2013.

BLENK, A. et al. Survey on network virtualization hypervisors for software defined networking. **IEEE Communications Surveys & Tutorials**, IEEE, v. 18, n. 1, p. 655–685, 2015.

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

BOSSHART, P. et al. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In: **Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM**. New York, NY, USA: Association for Computing Machinery, 2013. (SIGCOMM '13), p. 99–110. ISBN 9781450320566. Disponível em: <https://doi.org/10.1145/2486001.2486011>.

BROADCOM. **Jerico2 Ethernet Switch Series**. [S.l.], 2018. Disponível em: <https://www.broadcom.com/products/ethernet-connectivity/switching/stratadnx/bcm88690>.

BRUCE, D. What makes a good domain-specific language? apostle, and its approach to parallel discrete event simulation. **Kamin [43]**, p. 17–35, 1997.

CABALLERO, P. et al. Multi-tenant radio access network slicing: Statistical multiplexing of spatial loads. **IEEE/ACM Transactions on Networking**, IEEE, v. 25, n. 5, p. 3044–3058, 2017.

CAVIUM. **XPliant Ethernet Switch Product Family**. [S.l.], 2018. Disponível em: <https://cavium.com/xpliant-ethernet-switch-xp60-and-xp70-family.html>.

CHAN, A. K.; BIRKNER, J. M.; CHUA, H.-T. **Programmable application specific integrated circuit and logic cell therefor**. [S.l.]: Google Patents, 1992. US Patent 5,122,685.

CHANDRAKAR, S.; GAITONDE, D.; BAUER, T. Enhancements in ultrascale clb architecture. In: **Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays**. [S.l.: s.n.], 2015. p. 108–116.

CHEN, X. et al. P4sc: Towards high-performance service function chain implementation on the p4-capable device. In: IEEE. **2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)**. [S.l.], 2019. p. 1–9.

CHINNERY, D.; KEUTZER, K. **Closing the gap between ASIC & custom: tools and techniques for high-performance ASIC design**. [S.l.]: Springer Science & Business Media, 2002.

CHRISTEN, E.; BAKALAR, K. Vhdl-ams-a hardware description language for analog and mixed-signal applications. **IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing**, IEEE, v. 46, n. 10, p. 1263–1272, 1999.

CORDEIRO, W. L. da C.; MARQUES, J. A.; GASPARY, L. P. Data plane programmability beyond openflow: Opportunities and challenges for network and service operations and management. **Journal of Network and Systems Management**, v. 25, n. 4, p. 784–818, Oct 2017. ISSN 1573-7705.

DEURSEN, A. V.; KLINT, P.; VISSER, J. Domain-specific languages: An annotated bibliography. **ACM Sigplan Notices**, ACM New York, NY, USA, v. 35, n. 6, p. 26–36, 2000.

Digilent Inc. Netfpga-sume reference manual. In: . [s.n.], Accessed: July 2019. Disponível em: <https://reference.digilentinc.com/reference/programmable-logic/netfpga-sume/reference-manual>.

DOBRESCU, M. et al. Routebricks: Exploiting parallelism to scale software routers. In: **Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles**. New York, NY, USA: Association for Computing Machinery, 2009. (SOSP ’09), p. 15–28. ISBN 9781605587523. Disponível em: <https://doi.org/10.1145/1629575.1629578>.

DV, N. W. **Datasheet Programmable Switch FPGA-based 32-Port Network Switch**. [S.l.], 2017. Disponível em: <http://www.ldatech.com/wp-content/uploads/2017/10/Newwave-Programmable-Switch-FPGA-Based-32-Port-Network-Switch-Datasheet.pdf>.

FOUNDATION, O. N. **SDN Architecture Overview**. [S.l.], 2013. Disponível em: <https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/SDN-architecture-overview-1.0.pdf>.

FOWLER, M. **Domain-specific languages**. [S.l.]: Pearson Education, 2010.

GEMBER-JACOBSON, A. et al. Opennf: Enabling innovation in network function control. **SIGCOMM Comput. Commun. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 44, n. 4, p. 163–174, ago. 2014. ISSN 0146-4833. Disponível em: <https://doi.org/10.1145/2740070.2626313>.

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

HWANG, J.; RAMAKRISHNAN, K. K.; WOOD, T. Netvm: High performance and flexible networking using virtualization on commodity platforms. **IEEE Transactions on Network and Service Management**, IEEE, v. 12, n. 1, p. 34–47, 2015.

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.

JEPSEN, T. et al. Fast string searching on pisa. In: **Proceedings of the 2019 ACM Symposium on SDN Research**. [S.l.: s.n.], 2019. p. 21–28.

JIN, X. et al. Covisor: A compositional hypervisor for software-defined networks. In: **12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)**. Oakland, CA: USENIX Association, 2015. p. 87–101. ISBN 978-1-931971-218. Disponível em: <https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/jin>.

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

KIEBURTZ, R. B. et al. A software engineering experiment in software component generation. In: IEEE. **Proceedings of IEEE 18th International Conference on Software Engineering**. [S.l.], 1996. p. 542–552.

KIM, C. **Programming The Network Data Plane: What, How, and Why?** [S.l.], 2017. Disponível em: <https://conferences.sigcomm.org/events/apnet2017/slides/chang.pdf>.

KIM, C. et al. In-band network telemetry (int). **technical specification P**, v. 4, p. 2015, 2016.

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

KOPONEN, T. et al. Network virtualization in multi-tenant datacenters. In: **11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)**. Seattle, WA: USENIX Association, 2014. p. 203–216. ISBN 978-1-931971-09-6. Disponível em: <https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/koponen>.

KRUDE, J. et al. Online reprogrammable multi tenant switches. In: **ACM CoNEXT Workshop on Emerging In-Network Computing Paradigms**. New York, NY, USA: ACM, 2019. (ENCP '19), p. 1–8. ISBN 9781450370004. Disponível em: <https://doi.org/10.1145/3359993.3366643>.

KRUEGER, C. W. Software reuse. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 24, n. 2, p. 131–183, jun. 1992. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/130844.130856>.

KUMAR, P. et al. Picnic: Predictable virtualized nic. In: **Proceedings of the ACM Special Interest Group on Data Communication**. New York, NY, USA: Association for Computing Machinery, 2019. (SIGCOMM '19), p. 351–366. ISBN 9781450359566. Disponível em: <https://doi.org/10.1145/3341302.3342093>.

KUON, I.; ROSE, J. Measuring the gap between fpgas and asics. **IEEE Transactions on computer-aided design of integrated circuits and systems**, IEEE, v. 26, n. 2, p. 203–215, 2007.

KUON, I. et al. Fpga architecture: Survey and challenges. **Foundations and Trends® in Electronic Design Automation**, Now Publishers, Inc., v. 2, n. 2, p. 135–253, 2008.

LADD, D. A.; RAMMING, J. C. Two application languages in software production. In: **USENIX Very High Level Languages Symposium Proceedings**. [S.l.: s.n.], 1994. p. 169–178.

LI, B. et al. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In: **Proceedings of the 2016 ACM SIGCOMM Conference**. New York, NY, USA: Association for Computing Machinery, 2016. (SIGCOMM '16), p. 1–14. ISBN 9781450341936. Disponível em: <https://doi.org/10.1145/2934872.2934897>.

LI, S. et al. Sr-pvx: A source routing based network virtualization hypervisor to enable pof-fis programmability in vsdns. **IEEE Access**, IEEE, v. 5, p. 7659–7666, 2017.

LI, S. et al. Pvflow: Flow-table virtualization in pof-based vsdn hypervisor (pvx). In: IEEE. **2018 International Conference on Computing, Networking and Communications (ICNC)**. [S.l.], 2018. p. 861–865.

LI, S. et al. Protocol oblivious forwarding (pof): Software-defined networking with enhanced programmability. **IEEE Network**, IEEE, v. 31, n. 2, p. 58–66, 2017.

LOCKWOOD, J. W. et al. Netfpga–an open platform for gigabit-rate network switching and routing. In: IEEE. **2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)**. [S.l.], 2007. p. 160–161.

LUINAUD, T. et al. Bridging the gap: Fpgas as programmable switches. **GAs**, v. 8, p. 10, 2020.

MARTINS, J. et al. Clickos and the art of network function virtualization. In: **11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)**. Seattle, WA: USENIX Association, 2014. p. 459–473. ISBN 978-1-931971-09-6. Disponível em: <https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/martins>.

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. Disponível em: <https://doi.org/10.1145/1355734.1355746>.

MCKEOWN, N. et al. Openflow: enabling innovation in campus networks. **ACM SIGCOMM Computer Communication Review**, ACM New York, NY, USA, v. 38, n. 2, p. 69–74, 2008.

MIAO, R. et al. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In: **Proceedings of the Conference of the ACM Special Interest Group on Data Communication**. [S.l.: s.n.], 2017. p. 15–28.

NetFPGA Project. Netfpga wiki documentation. In: . [s.n.], Accessed: December 2019. Disponível em: <https://github.com/NetFPGA/NetFPGA-SUME-public/wiki>.

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

PFAFF, B. et al. The design and implementation of open vswitch. In: **12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)**. Oakland, CA: USENIX Association, 2015. p. 117–130. ISBN 978-1-931971-218. Disponível em: <https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff>.

POKORNY, M. R. et al. **User interface for reporting event-based production information in product manufacturing**. [S.l.]: Google Patents, 2008. US Patent 7,380,213.

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: ACM, 2019. (SIGCOMM Posters and Demos '19), p. 80–81. ISBN 978-1-4503-6886-5.

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

SHAHBAZ, M. et al. Pisces: A programmable, protocol-independent software switch. In: **Proceedings of the 2016 ACM SIGCOMM Conference**. New York, NY, USA: Association for Computing Machinery, 2016. (SIGCOMM '16), p. 525–538. ISBN 9781450341936. Disponível em: <https://doi.org/10.1145/2934872.2934886>.

SHERWOOD, R. et al. Flowvisor: A network virtualization layer. **OpenFlow Switch Consortium, Tech. Rep**, v. 1, p. 132, 2009.

SIRER, E. G.; BERSHAD, B. N. Using production grammars in software testing. **ACM SIGPLAN Notices**, ACM New York, NY, USA, v. 35, n. 1, p. 1–13, 1999.

SIVARAMAN, A. et al. Dc. p4: Programming the forwarding plane of a data-center switch. In: **Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research**. [S.l.: s.n.], 2015. p. 1–8.

SONG, H. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In: **ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking**. New York, NY, USA: ACM, 2013. (HotSDN '13), p. 127–132. ISBN 978-1-4503-2178-5.

SONG, H. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In: **Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking**. New York, NY, USA: Association for Computing Machinery, 2013. (HotSDN '13), p. 127–132. ISBN 9781450321785. Disponível em: <https://doi.org/10.1145/2491185.2491190>.

THOMAS, D.; MOORBY, P. **The Verilog® Hardware Description Language**. [S.l.]: Springer Science & Business Media, 2008.

TOKUSASHI, Y. et al. The case for in-network computing on demand. In: **Proceedings of the Fourteenth EuroSys Conference 2019**. New York, NY, USA: Association for Computing Machinery, 2019. (EuroSys '19). ISBN 9781450362818. Disponível em: <https://doi.org/10.1145/3302424.3303979>.

TOUCH, J. et al. A virtual internet architecture. **ISI Technical Report ISI-TR-2003-570**, Citeseer, p. 73–80, 2003.

WOODRUFF, J. et al. The cheri capability model: Revisiting risc in an age of risk. In: IEEE. **2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)**. [S.l.], 2014. p. 457–468.

WU, D. et al. Accelerated service chaining on a single switch asic. In: **Proceedings of the 18th ACM Workshop on Hot Topics in Networks**. New York, NY, USA: Association for Computing Machinery, 2019. (HotNets '19), p. 141–149. ISBN 9781450370202. Disponível em: <https://doi.org/10.1145/3365609.3365849>.

XILINX. **7 Series FPGAs Configurable Logic Block User Guide**. [S.l.], 2018. Disponível em: <https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf>.

XILINX. **Vivado Design Suite User Guide: Partial Reconfiguration**. [S.l.], 2019. Disponível em: <https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug909-vivado-partial-reconfiguration.pdf>.

YANG, H. et al. Review of advanced fpga architectures and technologies. **Journal of Electronics (China)**, Springer, v. 31, n. 5, p. 371–393, 2014.

YI, B. et al. A comprehensive survey of network function virtualization. **Computer Networks**, Elsevier, v. 133, p. 212–262, 2018.

YU, J. et al. Forwarding programming in protocol-oblivious instruction set. In: IEEE. **2014 IEEE 22nd International Conference on Network Protocols**. [S.l.], 2014. p. 577–582.

ZHANG, C. et al. Hyperv: A high performance hypervisor for virtualization of the programmable data plane. In: IEEE. **Computer Communication and Networks (ICCCN), 2017 26th Int'l Conference on**. [S.l.], 2017. p. 1–9.

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

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

Zilberman, N. et al. Netfpga sume: Toward 100 gbps as research commodity. **IEEE Micro**, v. 34, n. 5, p. 32–41, Sep. 2014. ISSN 1937-4143.

APPENDIX A — PVS FORWARDING ENGINE TUTORIAL

This Appendix presents instructions to deploy the example project *l2_router_firewall*, available in repository [1] of PvS Forwarding Engine. This project implements three vS instances: a Layer-2 switch, simple Router, and a Firewall. The following Sections show how to prepare the environment, execute the HLS flow, and download the example to the FPGA board.

A.1 Preparing the Environment

Clone the repository from github:

```
git clone https://github.com/p4vbox
git pull --tags
```

Add the following lines to the file /username/.bashrc:

```
export PVS = /username/projects/pvs-forwarding-engine/src/scripts/settings.sh
source $PVS
```

Update environment variables:

```
source /username/.bashrc
```

Run the following script to install dependencies:

```
sudo $PVS_SCRIPTS/tools/setup_SUME.sh
```

Build the library and install the SUME driver:

```
$PVS_MAKE_LIBRARY
$PVS_INSTALL_DRIVER
```

A.2 Executing the HLS Flow

Update the environt variable P4_PROJECT_NAME to l2_router_firewall in file $PVS_SCRIPTS/settings.sh:

```
export P4_PROJECT_NAMEl̄2_router_firewall
```

Update the environment variables:

```
source $PVS
```

[1]PvS forwarding engine example projects on GitHub repo <https://github.com/pvs-sigcomm/pvs-forwarding-engine/tree/master/src/contrib-projects/sume-sdnet-switch/projects>

In the project folder, generate test data:

```
cd $PVS_SCRIPTS
./pvs.py l2 router firewall -name l2_router_firewall -t
```

Verify and compile P4 (P4-to-HDL HLS flow, Chapter 5.1):

```
./pvs.py l2 router firewall -name l2_router_firewall -c
```

Verify and generate vS hdl and vS driver (vS and driver generation flows, Chapters 5.1 and 5.2):

```
./pvs.py l2 router firewall -name l2_router_firewall -s
```

Run simulation environment to test the generated platform:

```
./pvs.py l2 router firewall -name l2_router_firewall
```

A.3 Downloading the Example to the FPGA Board

Generate bitstream for the full reconfiguration methodology (Chapter 6.1):

```
./pvs.py l2 router firewall -name l2_router_firewall -imp
```

Program the FPGA board:

```
sudo $PVS_PROGSUME
```

APPENDIX B — PUBLICATIONS

Journals:

1. **P4VBox: Enabling P4-based Switch Virtualization.** M. Saquetti, G. Bueno, W. Cordeiro, J.R. Azambuja. IEEE Communications Letters. Nov. 2019.
doi: 10.1109/ LCOMM.2019.2953031.
2. **Evaluating the reliability of a GPU pipeline to SEU and the impacts of software-based and hardware-based fault tolerance techniques.** M. Gonçalves, M. Saquetti, J.R. Azambuja. Microelectronics Reliability. Sep. 2018.
doi: 10.1016/j.microrel.2018.07.007.
3. **A low-level software-based fault tolerance approach to detect SEUs in GPUs' register files.** M. Gonçalves, M. Saquetti, F. Kastensmidt, J.R. Azambuja. Microelectronics Reliability. Sep. 2017.
doi: 10.1016/j.microrel.2017.07.035.

Conferences:

1. **Hard Virtualization of P4-based switches with VirtP4.** M. Saquetti, G. Bueno, W. Cordeiro, J.R. Azambuja. In proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos 2019 Aug 19 (pp. 80-81).
doi: 10.1145/3342280.3342314.
2. **VirtP4: An Architecture for P4 Virtualization.** M. Saquetti, G. Bueno, W. Cordeiro, J.R. Azambuja. In 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) 2019 May 20 (pp. 75-78). IEEE.
doi: 10.1109/ IPDPSW.2019.00021.

Submitted, awaiting review:

1. **PvS: Programmable Virtual Switches.** M. Saquetti, G. Bueno, J.R. Azambuja, W. Cordeiro. ACM SIGCOMM 2020.

Awards:

1. **Third Place - Undergraduate Group - ACM SIGCOMM Student Research Competition**. Poster: "Hard Virtualization of P4-based switches with VirtP4" M. Saquetti, G. Bueno, W. Cordeiro, J.R. Azambuja. ACM SIGCOMM 2019 Conference Posters and Demos 2019 Aug 19