P4VBox: Enabling P4-based Switch Virtualization

Pablo Rodrigues, Mateus Saquetti, Guilherme Bueno, Weverton Cordeiro, Jose Rodrigo Azambuja
Institute of Informatics - PPGC - PGMICRO - Federal University of Rio Grande do Sul (UFRGS)
Av. Bento Gonçalves 9500, Porto Alegre - RS - Brazil
{pablo.rodrigues, mspctirone, gboliveira, weverton.cordeiro, jose.azambuja} @inf.ufrgs.br

***Abstract*—Virtualization has become the powerhouse for several networking concepts, from 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 power of forwarding plane virtualization remains untapped. In this letter, we present P4VBox, a reconfigurable architecture for data plane virtualization. P4VBox provides parallel execution and hot-swapping of virtual switch instances, without requiring switch source code (for either emulation or program composition). We experimented P4VBox on a NetFPGA-SUME board with three virtual switches: a layer-2 switch, a simple router, and a firewall. Area occupation measurements evidence the feasibility of running up to 13 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.**

***Index Terms*—Programmable Forwarding Planes, P4, Switch Virtualization, NetFPGA-SUME, Partial Reconfiguration.**

I. 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 [1] and P4 [2]. These languages unleashed innovation in the data plane [3], helping break further the "network ossification", and reshaped its research agenda to include aspects from switch program composition to verification and switch debugging. In this letter, we make the case for data plane virtualization.

The motivations for virtualization in the data plane are manifold, including network slicing and snapshotting, network function composition, and switch program profiling and debugging [4]. Yet, existing "hypervisors" for P4-based switches have only enabled switch *emulation* [4], [5] or composition of P4-based switch functions into a single program [6]. 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 intellectual property of switches.

We 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 like HyPer4 [4], HyperVDP [5], and P4Visor [6] partially fulfill some of these requirements, satisfying them simultaneously poses the following research challenges: (#1) decouple virtual switch instances from the virtualization environment, (#2) provide network flow isolation, (#3) ensure hardware resource isolation, (#4) support virtual networking within the hypervisor, and (#5) deliver an implementation with feasible performance and memory footprint.

HyPer4 and HyperVDP fall short in satisfying any of these research challenges, as they require custom compilation of switch source code, resulting in a *data plane model* (used for emulating the switch) that is tightly coupled with the hypervisor (challenge #1); use non-standard tagging scheme to provide flow isolation between switches, thus forcing other networking devices to support the same scheme, which may be unsuitable (#2); do not provide proper hardware resource isolation schemes like physical to virtual switch port mapping, CPU slicing, etc. (#3); *emulate* virtual networking with DSL primitives like P4 `recirculate` and `resubmit`, sacrificing throughput and latency (#4); and decouple match-action stages into a combination of several tables, leading to a ratio of declared tables per stage that makes it unfeasible to emulate more complex switches (#5). P4Visor, on the other hand, is able to deliver implementations of P4-based switches with feasible performance and memory footprint (#5), but without support to actual virtualization (challenges #1-4).

In this letter, we present P4VBox, an architecture that meets all the data plane virtualization requirements mentioned earlier, and an implementation methodology for hot-swapping virtual P4-based switches in the proposed architecture. We expand the state-of-the-art by providing a proof of concept deployment of the proposed architecture on a NetFPGA SUME board and discussing results achieved and lessons learned.

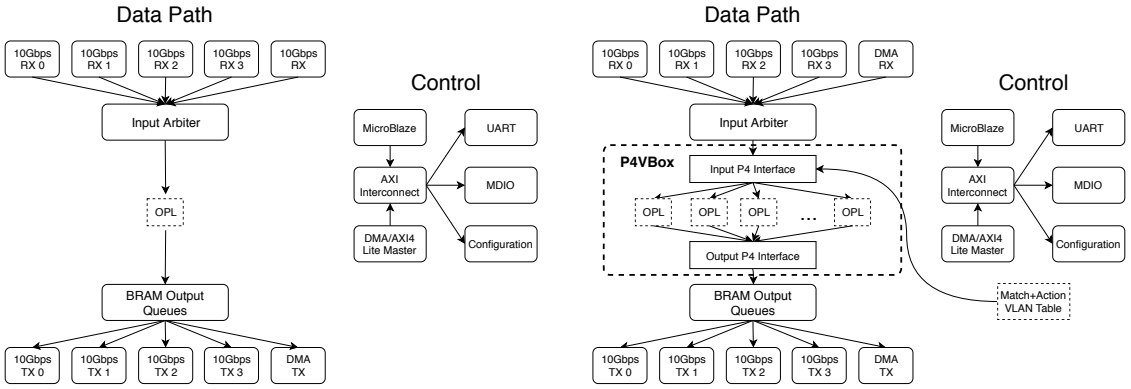
The novelty of this work lies in the following contributions:

- A conceptual and loosely coupled architecture for switch virtualization (thus addressing challenge #1) that provides network flow isolation through standard protocols (#2), hardware resource isolation (#3), virtual networking between switch instances hosted in the hypervisor (#4), and improved performance and memory occupation (#5);
- An implementation methodology for partially reconfiguring P4-based switches in FPGA boards.

In the remainder of this letter we present our conceptual design, followed by a discussion on deployment and evaluation. We then conclude with lessons learned and research directions.

II. Reconfigurable P4VBox Architecture

We argue that any switch hypervisor must allow one to deploy virtual switches from the same switch byte-codes one



(a) Canonical NetFPGA reference design [7]

(b) P4VBox design

Fig. 1. A comparison between P4VBox and the canonical NetFPGA design.

would deploy in a non-virtualized environment. P4VBox is an architecture we propose to fulfill that vision. Built on top of the canonical NetFPGA reference design [7], P4VBox provides support for deployment and parallel execution of multiple P4-based switches, tackling the discussed research challenges.

Figure 1 shows the architectures of the canonical NetFPGA reference design and of P4VBox. The canonical design is composed of the basic NetFPGA structure and a single Output Port Lookup (OPL) instance. Even though the canonical reference design alone enables one to implement a loosely coupled P4-based switch with feasible performance and memory footprint (challenges #1 and #5), it does not support switch instances running in parallel, and thus data flow and hardware resource isolation, and virtual networking (#2-4). P4VBox solves these challenges by expanding it to introduce an Input P4 Interface (IPI), an Output P4 Interface (OPI), and multiple OPL instances. In the following, we describe the proposed modules, how P4VBox is able to achieve challenges #2-4, as well as its reconfiguration capabilities.

A. Canonical NetFPGA Reference Design Architecture

We depict in Figure 1(a) an overview of canonical design. 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 virtual switch, 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.

For generating a switch implementation from a P4 description, we take advantage of a High-Level Synthesis (HLS) methodology and the Simple Sume Switch (SSS) model, available from the P4-NetFPGA project[1]. 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.

[1]https://github.com/NetFPGA/P4-NetFPGA-public

B. P4VBox Architecture

We show P4VBox in Figure 1(b). It improves the canonical design by replacing the single OPL module by a structure with multiple virtual OPL modules and an IPI and OPI. When instancing multiple OPLs, three issues arise: (i) multiple input flows must be distributed to deployed OPLs, (ii) processed flows must be delivered to shared output ports, and (iii) switch instances must be swapped in OPLs during runtime.

We address the first issue with the **Input P4 Interface** (IPI), a module that distributes data frames to multiple P4 switch instances deployed on OPLs. It receives frames from the IA and decides whether to forward them to a switch or to drop them silently. To this end, the IPI searches for a specific Switch identifier (*Sid*). In case the *Sid* of a deployed switch is found, it forwards the frame. Otherwise, the frame is dropped.

Output Port Lookup (OPL) instances are the same as the one used by the canonical reference design. The only difference is that P4VBox can support as many OPLs as one can physically fit in the board, instead of only one. Each OPL can receive a switch instance with individual set of networking protocols, given that they implement the *Sid* tagging scheme. Switch metadata, variable scope, and control flow operate without sharing information or resources with other instances.

The **Output P4 Interface** (OPI) addresses the second issue, delivering packets from the virtual switch instances to the corresponding output ports. It works like a demultiplexer, receiving frames from each deployed virtual instance and forwarding them to the target physical port.

The **Control** module provides support to the architecture and implements a Command-Line Interface (CLI) to interact with the network operator. It has been modified mainly to support feedback from the IPI and OPI, deployment of multiple P4 switch instances, and partial reconfiguration.

Combined, these components form the P4VBox architecture, which is able to satisfy all previously mentioned research challenges. In the following, we describe how we address challenges #2-4: network flow isolation, hardware resource isolation, and virtual networking within the hypervisor.

Network flow isolation (#2). The *Sid* can follow any regular expression, as long as it is informed to the IPI through the Control module. We adopt VLAN (802.1Q) concatenated with the frame destination address as *Sid* and assume that incoming data frames have a 32-bit tag. With this approach, virtual switch instances belonging to a same VLAN cannot share physical I/O ports in the physical switch.

Hardware resource isolation (#3). The IPI and OPI are responsible for physical to logical I/O port mapping. They enable, for example, each virtual switch to have an arbitrary number of virtual ports. To perform port mapping, we use the `src_port` and `dst_port` fields of `struct sume_metadata_t`. The I/O port mapping is stored in the match+action VLAN table in the Control module.

Virtual networking within the hypervisor (#4). We provide virtual networking by copying to the DMA port those frames that are destined to another virtual switch within the hypervisor. To this end, we also use the VLAN match+action table in the Control module, matching the frame destination address.

C. Partial Reconfiguration for Switch Hot-Swapping

Partial reconfiguration solves the third issue discussed in the previous subsection and represents the cornerstone of P4VBox, enabling the dynamic swap of P4-based switches within predefined OPLs. It requires the implementation of a single full configuration for the architecture, with a hardcoded number of OPLs, and partial configurations, one for each switch instance. Note that it is possible to change the number of hardcoded OPLs, though it requires full board reconfiguration.

The use of partial configurations also requires improvements on the canonical NetFPGA reference design, especially in the Control module and supporting hardware structures. The Control modifications include internal and external support. The former is required to maintain consistency between IPI, deployed switch instances, and OPI and obtain status reports from deployed modules. The latter, through the CLI, is required to interface with the network operator, which is able to load new P4-based switches into P4VBox, check which are deployed, and reconfigure them. Supporting hardware, such as memories and configuration ports, is used to store partial configurations and configure modules during runtime.

Through partial reconfiguration, P4VBox can perform hot-swapping, allowing the network operator to deploy new P4 switch instances during execution, without halting the system, and to reduce deployment times of P4 modules around two orders of magnitude, when compared to full reconfiguration.

III. Implementation Methodology

Our proof of concept implementation of P4VBox is based on a fork of the P4-NetFPGA code, with a total diff of around 3,000 lines of code[2]. P4VBox components were developed in VHDL and Verilog hardware description languages.

To generate P4-based switch instances and instance them in P4VBox's OPLs, we run P4 sources through a HLS process. Figure 2 presents our proposed HLS flow. Initially, the HLS flow receives a `.p4` source file described in $P4_{16}$ and an `arch.cfg` file from the network operator. The latter contains information that signals to the Xillinx SDNet compiler which segments of the code can be optimized during hardware implementation. This information includes the largest packet size that the parser and deparser have to support, the maximum number of clock cycles an external function needs to complete, and the size of the address space to allocate to an external function. Since the P4 language is target independent, the compiler always assumes the worst-case scenario regarding the previous information. Then, the Xilinx SDNet compiler makes the conversion of the $P4_{16}$ code to the intermediate PX description language. Two files are generated in the process: a description of the virtual switch in PX (`.eHDL`) and a file containing information regarding the interface with the control plane (`.info`). The `.eHDL` and `arch.cfg` files are given as input to `Deploy.tcl`, a script that deploys the virtual switch into P4VBox. Finally, the `.info` file is used as input to `Define Switch Tables.sh` script, which creates internal routing tables for the virtual switch.

To evaluate P4VBox, we implemented benchmarks from HyPer4 [4], HyperVDP [5], and P4Visor [6]: a layer-2 switch, a router, a firewall, and a combination of the layer-2 switch with the router. The first one is a simple L2 switch with learning ability that routes packets based on two match-action tables, for source and destination MAC tables; the second is a router that processes packets with the longest prefix match, locating destination address in an IPv4 routing table, updating source and destination MAC addresses, decrementing TTL, and setting the output port; and the third one is a L3/L4 firewall. We translated all switches to $P4_{16}$ before using the HLS flow. For the translation, we used the p4c compiler.

Table I shows the area occupation for the P4VBox architecture and the P4 modules. Note that the architecture components are always implemented on the board, independently of the deployed P4 switch instances. The static part requires around 12.2% of the board's LUTs, where the Control, 10Gbps Ports, and DMA together account for 84% of this value. The L2 switch, router, and firewall switch instances require 6.5%, 13.3%, and 10% of the board's LUTs, respectively. These results provide evidence that the NetFPGA-SUME could run in parallel up to thirteen deployed L2 switch instances, six routers, eight firewalls, or a combination of the previous.

IV. Results

We simulated P4VBox on a NetFPGA-SUME board and implemented the switches mentioned earlier. The implementations resulted in a clock frequency of 528.6 MHz, as the critical path resides in the fixed part of the architecture. We measured latency and bandwidth by injecting packet loads (2x 64-byte packet for the first and 64x 256-byte packets for the second) and evaluating the number of clock cycles. We compared these results with those of software (Native bmv2 and HyPer4) and hardware (P4Visor) switches. The software switches were run on an Intel Core i7-6500 with 16GB RAM.

[2]GitHub repo: https://github.com/p4vbox/

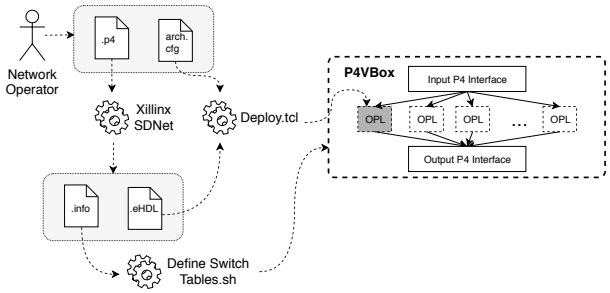


Fig. 2. Virtual switch deployment flow in P4VBox.

TABLE I
NETFPGA-SUME OCCUPATION

Module	LUT		FF	
	#	%	#	%
4x10Gbps Ports and DMA	26997	6.2%	34646	4.0%
Input Arbiter	2120	0.5%	4564	0.5%
Input P4 Interface	229	0.1%	1679	0.2%
Output P4 Interface	6181	1.4%	9779	1.1%
Control	17172	4.0%	25573	3.0%
L2 switch	28096	6.5%	40506	4.7%
Router	57704	13.3%	84149	9.7%
Firewall	45683	10.5%	74344	8.6%
TOTAL	184182	42.5%	275240	31.8%

TABLE II
LATENCY (μs)

P4 Module	Native bmv2	HyPer4 [4]	P4Visor [6]	P4VBox
L2 switch	460,500	1,570,000.0	–	0.54
Router	642,300	3,209,000.0	109.1	0.83
Firewall	553,600	2,133,000.0	–	0.82
L2 + Router	–	7,715,300.0	114.2	0.85

TABLE III
BANDWIDTH ($Mbps$)

P4 Module	Native bmv2	HyPer4 [4]	P4Visor [6]	P4VBox
L2 switch	105.0	22.4	–	88,760.9
Router	69.5	12.0	4,604.6	81,282.9
Firewall	67.0	14.6	–	81,801.6
L2 + Router	–	3.79	4,350.9	88,760.9

Tables II and III present latency and bandwidth, respectively. Observe that, while bmv2 and HyPer4 are software-based, they represent the only feasible basis for a comparative analysis.

When considering latency, P4VBox is up to six orders of magnitude faster than software switches and around 130 times faster than P4Visor. When considering bandwidth, P4VBox provided up to two orders of magnitude more bandwidth than software switches and around 20 times more than P4Visor. The main reason for the better performance of P4VBox compared to software switches is that, as a dedicated FPGA-based platform, it runs much faster than Native bmv2 and HyPer4 on a general-purpose CPU. When compared to P4Visor, the parallel architecture of P4VBox provides faster processing.

Table IV shows the number of match+action tables required. Observe that P4VBox 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, making it unfeasible to run them in NetFPGA.

TABLE IV
TABLE USAGE IN RUNTIME FOR DIFFERENT PROGRAMS

P4 Module	Native bmv2	HyPer4 [4]	HyperVDP [5]	P4VBox
L2 switch	2	13	5	2
Router	4	28	16	4
Firewall	3	22	8	3
L2 + Router	–	–	–	6 (2 + 4)

We measured reconfiguration times according to [8] and a 66 Mbps JTAG configuration port. Achieved reconfiguration times may vary according to bitstream size and configuration port bandwidth. Therefore, a smaller module or a configuration port with more bandwidth could improve reconfiguration times. Considering 2.8 Mb bitstreams, all switches required 0.72 seconds, against 60 seconds for a full board configuration. Even though we were able to lower deployment time by 98.8%, one could further reduce it by switching to configuration ports with more bandwidth, such as ICAP and SelectMAP, possibly lowering deployment times to around 0.015 seconds.

V. FINAL CONSIDERATIONS

We presented P4VBox, a proof-of-concept virtualization solution for P4 switches based on the Simple Sume Switch model for a NetFPGA SUME target environment. Our experiments 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 results were able to advance the state-of-the-art in terms of bandwidth and latency. For future work, we intend to (i) port our conceptual architecture to a Barefoot Tofino and Tofino 2, and (ii) address security-related aspects like match+action table isolation between switch instances.

REFERENCES

[1] H. Song, "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*, ser. HotSDN '13. New York, NY, USA: ACM, 2013, pp. 127–132.

[2] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[3] W. L. da Costa Cordeiro, J. A. Marques, and L. P. Gaspary, "Data plane programmability beyond openflow: Opportunities and challenges for network and service operations and management," *Journal of Network and Systems Management*, vol. 25, no. 4, pp. 784–818, Oct 2017.

[4] D. Hancock and J. Van Der Merwe, "Hyper4: Using p4 to virtualize the programmable data plane," in *CoNEXT'16*. ACM, 2016, pp. 35–49.

[5] C. Zhang, J. Bi, Y. Zhou, and J. Wu, "Hypervdp: High-performance virtualization of the programmable data plane," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 556–569, March 2019.

[6] P. Zheng, T. Benson, and C. Hu, "P4visor: Lightweight virtualization and composition primitives for building and testing modular programs," in *CoNEXT '18*. New York, NY, USA: ACM, 2018, pp. 98–111.

[7] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman, "The p4-netfpga workflow for line-rate packet processing," in *ACM/SIGDA Int'l Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: ACM, 2019, pp. 1–9.

[8] Digilent Inc., "Netfpga-sume reference manual," Accessed: July 2019. [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/netfpga-sume/reference-manual