Virtualization of Programmable Forwarding Planes with P4VBox

Pablo Rodrigues, Mateus Saquetti, Guilherme Bueno, Weverton Cordeiro, and Jose Rodrigo Azambuja

Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil
email: pablo.rodrigues@inf.ufrgs.br

Abstract— Networking virtualization has enabled faster service provisioning and serves as a main driver of innovation, from software-defined networking to network functions virtualization and local area networks. Recent investigations have been assessing the feasibility of virtualization in programmable data planes. Despite the progress achieved, much work remains to assess their effectiveness for programmable virtual switches. In prior work, we introduced P4VBox, an architecture for the virtualization of programmable switches written using the P4 language. P4VBox provides parallel execution of multiple P4-based switch instances with the hot-swapping ability through full and partial reconfiguration methods. In this work, we build upon P4VBox and provide novel insights, substantiated by an experimental evaluation on a real-world testbed, addressing the power of switch virtualization. We measure resource utilization and performance to observe the behavior of P4VBox when handling large flows. Results show that P4VBox can achieve around 25 times higher throughputs than related works that provide virtualization at the cost of around 1.1% more resources and 20% less throughput than the P4-NetFPGA canonical reference design.

Index Terms— Programmable Forwarding Planes; P4; Switch Virtualization; FPGA Reconfiguration.

I. INTRODUCTION

The advent of Programmable Data Planes (PDPs) provided a novel approach for flexible programming of network devices, enabling the provisioning of novel networking protocols and services through the use of Domain-Specific Languages (DSLs) like POF [1], P4 [2], and Lyra [3]. These DSLs helped deliver programmable switch packet forwarding feasibility by providing a higher-level and vendor-agnostic API. This API contrasts with the ones generally specified on fixed-function chips of traditional switches by manufacturers, making data planes more flexible and breaking the “network ossification.”

Although network virtualization has been largely explored in domains like Software-Defined Networking (SDN), Network Functions Virtualization (NFV), Cloud Computing, and even in Local Area Networks (VLANs and VXLANs), the potential of virtualization within PDP remains to be explored. Recent P4-based switch hypervisors have explored the feasibility of programmable data plane virtualization by emulating switches [4, 5, 6] and performing program composition to emulate several switch functions [7]. These hypervisors may, however, impose severe performance overhead to programmable switch virtualization, considering the multiple match-action tables that these hypervisors require. Additionally, even though other prior solutions offer support to P4-based switch parallel execution [8, 9], they lack in replacing virtual instances without affecting the data plane state.

In a prior work [10], we presented and discussed a reconfigurable architecture for the virtualization of multiples P4-based programmable switches and its materialization with P4VBox. P4VBox enables multiple P4-based switches running in parallel in a single hardware substrate, allowing the hot-swapping of virtual instances without interfering in the operation of the others. In summary, P4VBox provides the network operator with the ability to deploy and hot-swap virtual instances, delivers performance levels equal to virtual switches deployed directly on the hardware, and protects the switch bytecode intellectual property provided by the network operator. In that prior work, we provided evidence through simulation that provides parallel execution of multiple virtual instances in a single hardware substrate, allowing the hot-swapping of the virtual switches.

In this paper, we move beyond our previous work and build upon P4VBox to provide novel insights, substantiated by experimental evaluation on a real-world testbed, on the evaluation of the real power of switch virtualization in a NetFPGA-SUME board. We used our testbed to assess the feasibility of PDP virtualization with three typical use cases of programmable switches: a layer-2 switch, a router, and a firewall. We measured resource utilization and performance to observe the behavior of P4VBox when handling large flows. Our results provide solid evidence that P4VBox incurs a small overhead compared with the canonical NetFPGA reference design. Yet, it presents an increase in performance when considering the existing works. As another significant contribution to the state of the art, we also discuss in this paper a High-Level Synthesis (HLS) process to deploy a virtual switch into P4VBox, and the implementation methodology used for both full and partial reconfiguration.

The remainder of this paper is organized as follows. We discuss related work in Section II. In Section III, we provide an overview of the P4 language, which is used to program the virtual switches that run on top of P4VBox. We discuss P4VBox architecture for programmable switch virtualization in Section IV and propose a methodology for virtual switch deployment in Section V. We present the results of our evaluation in a real-world testbed with three use cases (12 switch, router, and firewall) in Section VI. We then close the paper in Section VIII with concluding remarks.

II. RELATED WORK

In this section, we discuss the related works to elucidate our proposed approach.

A. Switch program emulation

HyPer4 [4] and HyperVDP [5, 6] utilize a data plane model tightly coupled to the hypervisor that requires custom compilation of switch source code, therefore lacking

mechanisms to decouple virtual switch instances from the virtualization environment. These proposals also intend to provide flow isolation between switches with a non-standard tagging design, thus forcing other networking devices to support the same design, which may be unsuitable for enforcing flow isolation between virtual switch instances. Hardware resource isolation schemes are also not properly provided in these proposals, like physical to virtual switch port mapping, CPU slicing, etc. These systems emulate virtual networking by applying DSL primitives like P4 *recirculate* and *resubmit*, causing harm to the throughput and latency, thus lacking support for virtual networking within the hypervisor. Finally, these solutions decouple match-action stages into a combination of several match-action tables, making it improper for a more complex switch to be deployed in the solutions with a feasible performance and memory footprint.

B. Switch program composition and virtualization

P4Visor [7] allows the composition of multiples P4 programs into a single P4 program. The composed programs can be configured to emulate multiples switch functions running in a switch substrate. Unlike HyPer4 [4] and HyperVDP [5, 6], P4Visor supports P4-based switches with a feasible performance and memory footprint. However, it possesses flaws in providing solutions to virtualization challenges, like decoupling virtual switch instances from the virtualization environment, providing network flow isolation, ensuring hardware resource isolation, and supporting virtual networking within the hypervisor. Table I summarizes the key aspects of each proposal.

III. P4 LANGUAGE OVERVIEW

Programming Protocol-independent Packet Processors (P4) [2] is a high-level declarative language used to configure programmable data planes and specify how switches must process and forward packets.

The main goals of the P4 language consist of reconfigurability, protocol independence, and target independence. The reconfigurability defines that a network operator can redefine the parsing and processing of packets with a controller. P4 language can offer protocol independence through the extraction of non-standard header fields of packets specifying the packet parser and a set of tables and actions responsible for handling incoming packets of the switch. The target independence focuses on the possibility of network operators to describe the functionality of a switch without requiring extra information about the physical substrate.

Generally, a traditional switch is built on top of a “bottom-up” approach, where the switch’s data plane functionality is predefined by manufacturers. In contrast, a P4-based switch does not have a fixed data plane, thus offering a “top-down” approach and providing a network operator with the possibility to program the data plane writing a P4 program with a set of functionalities and then load it in a programmable switch. Fig. 1 shows the difference between these approaches.

Essentially, a vendor must provide an architecture model that specifies the components which a network operator can define and program, and a P4 compiler for the packet processing target device.

Fig. 1: Traditional switches vs. programmable switches.

```

1  control MatchAction(inout Parsed_packet hdr,
2                        inout Metadata metadata)
3  {
4      action forward(bit<8> port) {
5          metadata.dst_port = port;
6      }
7
8      table mac {
9          actions = { forward; }
10         key = { hdr.ethernet.dst_addr: exact; }
11         size = 64;
12     }
13
14     apply {
15         mac.apply();
16     }
17 }

```

Fig. 2: Match-action processing block of a L2 switch.

A P4 program is composed of four basic elements: (i) headers, formatted data that specifies network protocols; (ii) parser, implemented as a finite state machine to extract and analyze headers and fields of an incoming packet; (iii) match-action processing block, which uses tables and actions to analyze fields within a packet to try to perform a match with the key metadata, resulting in a certain action; and (iv) deparser, a control block that reassembles headers of an outgoing packet.

In Fig. 2 we show an example of a match-action processing block of a simple L2 switch. Table II shows a MAC address table with populated entries. The L2 switch match-action processing block receives a parsed packet (*hdr*) and metadata struct (*metadata*). In case of the MAC address in the destination MAC address field in the ethernet header (*hdr.ethernet.dst_addr*) of the parsed packet is matched with an entry on the MAC address table (*mac*), the *forward* action is activated, resulting in the packet being forwarded to the MAC address destination port specified in *mac* table.

We choose the P4 language as our DSL because P4 is the most used language in the literature. With a large adoption by academia, P4 has a large variety of available documentation and supporting tools.

Table I.: Existing work on virtualization of based-P4 forwarding elements.

Solution	Underlying hypervisor	Abstraction	Model	Custom compilation or merging?
Hyper4 [4]	Software switch	Virtual switch	Emulation	Yes
HyperVDP [5, 6]	Software switch	Virtual switch	Emulation	Yes
P4Visor [7]	Single switch program	Switch function	Emulation	Yes, switch program merging
VirtP4 [8]	Hardware switch	Virtual switch	Virtualization	No
P4VBox	Hardware switch	Virtual switch	Virtualization	No

Table II.: Example of a destination MAC address table.

Key	Action	Action Data
08:11:11:11:11:08	forward	1
08:22:22:22:22:08	forward	2

IV. RECONFIGURABLE P4VBOX ARCHITECTURE

P4VBox is a novel solution to programmable data plane virtualization [10]. It materializes a vision of a data plane hypervisor model that guarantees the ability to deploy and hot-swap virtual switch instances and provide performance levels equal to virtual switches deployed directly on the hardware. P4VBox is built on top of the canonical NetFPGA reference design architecture [11], which supports only one switch instance. Fig. 3 shows the architectures of the canonical NetFPGA reference design and P4VBox. This section describes both architectures and discusses key aspects to be observed when creating a virtualization model that offers the possibility to instantiate multiple switch instances.

A. Canonical NetFPGA Reference Design Architecture

The canonical NetFPGA reference design is based on the Simple Sume Switch (SSS) architecture model. The switch logic is composed of a parser, a match-action pipeline, and a deparser [11]. Fig. 3(a) shows the canonical NetFPGA reference design, which consists of four TX/RX and DMA TX/RX ports, Input Arbiter (IA), Output Port Lookup (OPL), and BRAM Output Queues(BOQ). The Input Arbiter admits packets arriving from input ports and forwards them towards the Output Port Lookup, where the main packet processing occurs, and the output port is selected. The OPL is a placeholder responsible for hosting the virtual switch instances. The BRAM Output Queues buffers packets while they wait to be sent to output ports. At last, the Management Interface allows the network operator to communicate with the switch.

B. P4VBox Architecture

Below, we briefly review our architecture for programmable switch virtualization with P4VBox [10]. P4VBox improves the canonical NetFPGA reference design by adding the Input P4 Interface (IPI), Output P4 Interface (OPI), and replacing a single OPL module with multiple OPLs, thus, allowing P4VBox to support multiples virtual switches instances. A conceptual view of P4VBox is showed in Fig. 3(b). To deal with the capability of instantiating multiple virtual switches in parallel in P4VBox, we come across the following issues: (i) how to steer multiples packet flows between the virtual switch instances deployed; (ii) how to deliver the processed packet flows to the correct

(a) Canonical NetFPGA reference design [11].

(b) P4VBox design.

Fig. 3: A comparison between P4VBox and the canonical NetFPGA design.

output ports, and (iii) how to provide hot-swap between switch instances.

The **IPI** solves the first issue. It receives packet flows by the Input Arbiter and distributes them to deployed switch instances or drops them. To guarantee the correct packet forwarding, when a frame packet arrives, IPI searches in the packet for a switch identifier that identifies the destination switch instance, which should be encapsulated in the packet information. In case the identifier is found, IPI forwards the frame to the correct switch instance. Otherwise, the packet is dropped. We adopted the VLAN protocol (802.1Q) concatenated with the packet destination address as the switch identifier. With this approach, virtual switch instances that are on the same VLAN cannot share physical TX/RX ports in the physical substrate. Following a regular expression, a network operator can provide a switch identifier informing it

to the IPI through the Management Interface.

P4VBox provides slicing resources through **OPL** instances, which is an array of placeholders for hosting virtual switch instances. In contrast to the canonical NetFPGA reference design, P4VBox can support as many virtual switches as one can physically fit in the board, without degrading the performance of each virtual switch instance. Each OPL can host P4-based switch instances, assuming that they implement the identifier scheme. Moreover, P4VBox provides virtual networking by copying to the DMA port the frame packets that are destined to another virtual switch within the hypervisor. It enables the communication between the instances running on the board. By doing so, a switch instance can send a packet using a virtual transmission port to a virtual port intended for receiving packets and then send it to the destination switch instance running on P4VBox through a “recirculation” process. To this end, we also use the VLAN match-action table in the control module, matching the frame destination address.

The **OPI** solves the second issue. Once the packet is successfully delivered to its corresponding virtual switch instance and processed, it must be forwarded to an output port. With IPI, OPI ensures hardware resource isolation for virtual switches, being responsible for physical to logical input/output port mapping.

The **Management Interface** allows the network operator to interact with the switches and the architecture. It has been modified to support multiple switch virtualization.

V. IMPLEMENTATION METHODOLOGY

P4VBox modules were developed in Verilog and VHDL description languages. As the base, P4VBox is based on a fork of the P4-NetFPGA code, with a total diff of around 3,000 lines of code¹. Fig. 4 shows our proposed HLS flow. Initially, to design a P4-based switch and deploy in P4VBox’s OPLs, it is necessary a P4 file which describes the functionality of the switch (*.p4*) coded in *P4₁₆*. Then the Xilinx SDNet makes the conversion of the P4 code to PX intermediate language and encrypts it. Two files are generated in this process: a file to allow communication with the switch (*.cli*), and a file describing the virtual switch (*.eHDL*). Finally, the (*.eHDL*) file is given as input to *Deploy.tcl* script which will deploy the virtual switch instance into P4VBox.

We argue that since a vendor or developer can provide a switch file through an HLS flow as input to *Deploy.tcl*, P4-based switches can be deployed in P4VBox’s OPLs without requiring access to switch code, therefore protecting vendors and developers’ intellectual property.

VI. RECONFIGURATION CAPABILITIES

The reconfiguration of P4VBox architecture can be achieved through full and partial reconfiguration methodologies. Full reconfiguration loads a full bitstream that replaces all logic on an FPGA board. Partial reconfiguration allows loading a partial bitstream overwriting only a predefined and reconfigurable area in a design running on an FPGA board. Below we explain more details about both methodologies applied in P4VBox.

GitHub repo: <https://github.com/p4vbox/>

Fig. 4: Virtual switch deployment flow.

A. Full Reconfiguration

Full reconfiguration configures the whole FPGA through a full bitstream file, replacing the current context running on the board for a new logic contained in the loaded bitstream file. Fig. 5(a) shows the premise of the configuration process, loading a full bitstream that will occupy the entire FPGA area.

Synthesis tools take advantage of full reconfiguration methodology since the design can be optimized more effectively due to the availability of resources, thus achieving a higher number of virtual switch instances in P4VBox. Also, better frequency operations can be obtained, which can result in better throughput and latency. However, to replace modules, full reconfiguration suffers penalties like delayed time to perform, since reconfiguration times are dependent on configuration file sizes and that full configuration files are larger than partial ones. Also, full reconfiguration erases all contexts across the board, not allowing a network operator to perform the hot-swap of a virtual switch instance without compromising all the context.

Thus, for P4VBox to be a fully reconfigurable architecture for data plane virtualization, it must use other methodology to deal with the following challenge: how to ensure the hot-swapping of a virtual switch instance in a manner that reconfiguration does not affect the operation of others and the context running on the FPGA.

B. Partial Reconfiguration and Switch Hot-Swapping

Partial reconfiguration solves the third issue pointed in the previous section. It enables modifying a fraction of the FPGA board fully configured through a partial bitstream file while running without compromising other functions on the rest of the FPGA. Unlike full reconfiguration, which starts its process by loading a full configuration file corresponding to the final design, partial reconfiguration can initially program the board with a design containing blank areas that can later

receive different functionalities. To do so, the logic in the FPGA design is divided into static and reconfigurable logic. The static logic is the part of the design that remains operating during a life cycle and is unaffected by loading a partial bitstream file. The reconfigurable logic is the dynamic part of the design, which can be replaced by different functionalities loaded through partial bitstream files during the design's life cycle.

To materialize this idea in P4VBox, OPLs are defined and statically allocated in the FPGA as a set of reconfigurable areas to host P4-based instances. These are marked as reconfigurable modules that one at a time can be deployed in the predefined reconfigurable areas. At the end of these steps, the full and partial bitstreams are generated. After a full bitstream file configures the FPGA, a partial bitstream file can be downloaded to modify reconfigurable areas in the FPGA board without compromising the integrity of the applications running on those parts of the device that are not being reconfigured. Fig. 5(b) shows the process of loading a full bitstream containing configurable areas marked as *OPL1* and *OPL2* on the FPGA board. After the full bitstream is loaded, partial bitstreams corresponding to reconfigurable modules can be deployed and replaced in the reconfigurable areas without compromising the other ones and the static part of P4VBox architecture. A point to highlight is that it is possible to change the number of hardcoded OPLs. However, it requires a network operator to implement the full reconfiguration process again, thus reconfiguring the entire board. Through partial reconfiguration, P4VBox can perform hot-swapping, allowing the network operator to deploy new P4-based virtual switch instances during execution without halting the system.

VII. EVALUATION

The purpose of our evaluation is to verify the performance, resource utilization and analyze the reconfiguration times of P4VBox. We deployed P4VBox in a NetFPGA-SUME Virtex-7 board with four 10Gbps ports, shown in Fig. 6, at 200 MHz operating frequency. To serve as a baseline for our experiment, we synthesized the P4-NetFPGA canonical reference design [11] and used data obtained from P4Visor [7]. In our experiments, we used three switch programs described in P4: a layer-2 switch, a simple router, and a firewall. These are open-source switches commonly used in the literature as case-study switches. The layer-2 switch forwards packets based on source and destination MAC tables. The simple router locates an incoming packet's destination address in an IPv4 routing table, updates source and destination MAC address, decrements TTL and sets the output port. Finally, the firewall filters incoming packets based on the packet source port. The switch programs were generated with the complete HLS implementation flow present in Section V.

A. Resource utilization

Our implementation resulted in a maximum operating frequency of 547.6 MHz for both P4VBox and P4-NetFPGA. This value is directly related to the fixed part of both architectures, where the critical path resides.

Table III shows the occupation area that was measured in terms of Look-Up Tables (LUTs) and the use of registers for

(a) Deployment through full reconfiguration methodology.

(b) Deployment through partial reconfiguration methodology.

Fig. 5: The premise of reconfiguration process for full and partial reconfiguration methodologies.

allocating P4VBox, P4-NetFPGA, and the switches. Both architectures, running a single P4-based switch instance, have similar resource usage. P4VBox spends 11.4% of available LUTs to provide virtualization of modules, whereas P4-NetFPGA, which does not provide virtualization, spends 10.3%. It shows that P4VBox requires around 1.1% resources when compared to the P4-NetFPGA to enable virtualization on PDP. This slight overhead can be justified by the addition of the needed logic to provide parallel switch instances execution.

B. Performance

We simulated P4VBox running at the theoretical maximum operating frequency of 547.6 MHz, rather than the clock frequency of the NetFPGA-SUME, which is 200 MHz. To measure throughput and latency we injected 512×256 -byte packets and 2×64 -byte packets, respectively, to simulate an incoming packet flow. For both packet flows, we measured on the Xilinx Vivado Simulator the number of clock cycles between the arrival of the first packet and the completion of the last. Table IV shows the achieved throughput,

Fig. 6: NetFPGA-SUME FPGA Development Board [12].

Table III.: NetFPGA SUME resource usage.

Module	LUTs		Registers	
	#	%	#	%
P4VBox	48471	(11.4%)	74338	(8.6%)
P4-NetFPGA [11]	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%)

where P4VBox achieved around 100 Gbps which translates to 37 Gbps at the deployed 200 MHz operating frequency. When compared with P4Visor and P4-NetFPGA, P4VBox's throughput is around 25 times higher than P4Visor and is around 20% smaller than P4-NetFPGA. Table V shows latency, which was of around $0.74 \mu s$, and shown a decrease of orders of magnitude than P4Visor and an increase of 6.8% when compared to P4-NetFPGA.

In comparison with P4Visor, our approach shows a higher throughput performance because its parallel architecture provides faster processing. When compared to P4-NetFPGA, ours has a deeper pipeline, as it also contains modules to deal with programmable switches virtualization (i.e., IPI and OPI), which P4-NetFPGA architecture does not provide. As these modules use buffer queues implemented in BRAM modules to store packets, destinate them to the right switch, and buffers the outgoing packets to the output, the access time to the data stored in the queues and the logic to implement virtualization are a key factor that impacts the throughput achieved. Also, this addition results in a longer path for packet processing and forwarding, consequently affecting latency. Finally, our approach requires packets to have a VLAN tag scheme, which also impacts effective throughput. Despite small overheads, the obtained results show that P4VBox delivers enough latency and throughput to up to 10×10 Gbps ports at the maximum operating frequency, however still performs slightly worse than P4-NetFPGA, showing the performance cost of implementing parallel virtual switch instances.

To saturate a single port with multiple connections, we deployed sixteen virtual machines, each one with a 1Gbps Network Interface Card (NIC) generating packet flows to an OpenFlow Switch (with 48×1 Gbps RJ45 and 4×10 Gbps SFP+ ports). Each SFP+ port is mirrored to 10 RJ45 ports using Openflow rules, and each RJ45 port is connected to a

Table IV.: Throughput (Gbps)

Switch instance	P4Visor [7]	P4-NetFPGA [11]	P4VBox
L2 switch	—	121.8	101.1
Router	4.6	119.6	100.2
Firewall	—	114.8	99.8
L2 + Router	4.4	—	118.3

Table V.: Latency (μs)

Switch instance	P4Visor [7]	P4-NetFPGA [11]	P4VBox
L2 switch	—	0.52	0.56
Router	109.1	0.79	0.84
Firewall	—	0.76	0.81
L2 + Router	—	—	—

VM NIC. Fig. 7 shows the setup mounted to evaluate performance experiments. In the first experiment, we placed the Openflow switch SFP+ ports in a loopback; in the second experiment, the Openflow switch SFP+ ports are connected to the NetFPGA-SUME, running P4-NetFPGA; in the third experiment, we replace P4-NetFPGA running on NetFPGA-SUME for P4VBox. P4VBox forwards the VM flows to a given virtual switch instance, which then processes those flows and forwards them back to the OpenFlow switch. Finally, the flows are redistributed back to the VMs. We use iperf3 to measure and generated packet flows. Fig. 8 shows throughput for the loopback, P4VBox, and P4-NetFPGA, increasing each new iperf3 flow with less than a 2% difference up until saturation point, where they slightly differ due to cubic converging to a fair bandwidth shared between flows. All iperf3 flows stabilize at 9.5 Gbps.

C. Reconfiguration Analysis

Fig. 9 shows the maximum resource allocation for full and partial reconfiguration methodologies on NetFPGA-SUME. Fig. 9(a) shows a snapshot of a P4VBox full reconfiguration example containing eight layer-2 switch instances; Fig. 9(b) shows the same layer-2 switch instances into 4 reconfigurable areas, each one containing a single layer-2 switch instance.

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 re-

Fig. 7: Setup for performance evaluation.

Fig. 8: Port saturation throughput measurement.

(a) 8x layer-2 switch instances using full reconfiguration.

(b) 4x layer-2 switch instances using partial reconfiguration.

Fig. 9: P4VBox design methodologies implementation.

configuration was limited by the sizing and placement of the reconfigurable areas. Reconfiguration time is directly related to configuration files' size and the configuration port's bandwidth [13]. Our generated bitstreams, which have been compressed by the synthesis tool, contain sizes around 14.6 MB (full) and 4.3 MB (partial). Our measurements have been performed through a USB-JTAG configuration port operating at 66 Mbps on the host and resulted in a reconfiguration time of around 8 and 4 seconds for full and partial configuration files, respectively. We took into account the metric proposed by Xilinx [13] to estimate reconfiguration times and implemented a script that monitors the reconfiguration time of Vivado's process reconfiguration to confirm that. As measurements have been performed on the host-side, there are hidden overheads related to the system that could be optimized to decrease reconfiguration times down to 1.8 and 0.5s. The experiments demonstrated that the design's performance using full reconfiguration showed no decrease, while the partial reconfiguration showed decreases of 65% and

71% for three and four reconfigurable areas, respectively. It is important to notice that, better reconfiguration times can be achieved when newer technologies are considered, like using onboard flash memory and using the SelectMap interface (3.2 Gbps) to achieve reconfiguration times around 36 and 11 milliseconds.

Both methodologies have advantages and disadvantages. To highlight these, an analysis of the scenario must be done to choose which methodology should be used. The full reconfiguration enables better resource allocation, but it takes longer to download the bitstream on the FPGA board, thus full reconfiguration is recommended for those scenarios where all switch instances fit on the target board or changes in the running context are not made with a higher frequency since the context switching can impact the other switch instances. Partial reconfiguration is suitable for those scenarios where few changes are made with more frequency, like replace a switch instance with another one, due to the higher flexibility offered by the hot-swapping and the low reconfiguration times, but this methodology has a cost since the size of reconfigurable partitions can limit the number of switch instances running on the board.

VIII. FINAL CONSIDERATIONS

The ability to achieve data plane programmability has offered network operators several opportunities to define their network behavior. Performing virtualization in the forwarding plane follows that pattern, opening precedents to use multiple virtual data plane devices on top of a physical substrate. On the other hand, several challenges arise from delivering a feasible proposal, like guaranteeing that switch code intellectual property is protected or decoupling the virtual switch instances from the hypervisor.

Therefore, we present P4VBox, a reconfigurable architecture for P4-based switch virtualization that attends to the research challenges, allowing hot-swapping of the virtual instances while preserving the switch's intellectual property. P4VBox is based on the Simple Sume Switch model for a NetFPGA-SUME target environment. We experimented P4VBox with three P4 virtual switches: (i) layer-2 switch (L2), (ii) router, and (iii) firewall. Our evaluation results show that P4VBox has a good performance latency and throughput compared with the state of the art solutions like P4Visor and a lightly lower performance level compared with P4-NetFPGA, considering the performance cost of multiple virtual switch instances running in parallel. Also, the full and partial reconfiguration results confirm low reconfiguration times with virtual switches.

ACKNOWLEDGEMENTS

The authors thank CNPq and FAPERGS for the financial support to this work. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. This work also received funding from Grant #2020/05183-0, São Paulo Research Foundation (FAPESP). We also received support from Xilinx Inc.

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] J. Gao, E. Zhai, H. H. Liu, R. Miao, Y. Zhou, B. Tian, C. Sun, D. Cai, M. Zhang, and M. Yu, "Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 435–450.
- [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, A. B. Dogar, and J. Wu, "Hyperv: A high performance hypervisor for virtualization of the programmable data plane," in *Computer Communication and Networks (ICCCN), 2017 26th Int'l Conference on*. IEEE, 2017, pp. 1–9.
- [6] 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.
- [7] 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.
- [8] M. Saquetti, G. Bueno, W. Cordeiro, and J. R. Azambuja, "Virt4: An architecture for p4 virtualization," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2019, pp. 75–78.
- [9] M. Saquetti, G. Bueno, W. Cordeiro, and J. R. Azambuja, "Hard virtualization of p4-based switches with virt4," in *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, ser. SIGCOMM Posters and Demos '19. New York, NY, USA: ACM, 2019, pp. 80–81.
- [10] M. Saquetti, G. Bueno, W. Cordeiro, and J. R. Azambuja, "P4vbox: Enabling p4-based switch virtualization," *IEEE Communications Letters*, vol. 24, no. 1, pp. 146–149, 2019.
- [11] 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.
- [12] Digilent Inc., "Netfpga-sume reference manual," Accessed: July 2019. [Online]. Available: <https://reference.digilentinc.com/reference/programmable-logic/netfpga-sume/reference-manual>
- [13] Xilinx, *Vivado Design Suite User Guide: Partial Reconfiguration*, 2019. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug909-vivado-partial-reconfiguration.pdf