P4rt-OVS: Programming Protocol-Independent, Runtime Extensions for Open vSwitch with P4

Tomasz Osiński[‡†], Halina Tarasiuk[†], Paul Chaignon[◇*], Mateusz Kossakowski[‡†]
[‡]Orange Labs, [†]Warsaw University of Technology, Warsaw, Poland
[◇]Isovalent, Eindhoven, The Netherlands

***Abstract*—Virtualized data centers implement overlay networking to provide network isolation. The key component that makes the overlay networking possible is a hypervisor switch, such as Open vSwitch (OVS), that is running on each compute node and switches packets to and from virtual machines. Software switches frequently require upgrading and customization of network protocol's stack to introduce novel or domain-specific networking techniques. However, it is still difficult to extend OVS to support new network features as it requires mastery of network protocol design, programming expertise and familiarity with the complex codebase of OVS. Moreover, there is currently no solution that enables the deployment of network features in OVS without recompilation.**

In this paper, we present P4rt-OVS, an original extension of OVS that enables runtime programming of protocol-independent and stateful packet processing pipelines. It extends the forwarding model of OVS with Berkeley Packet Filter (BPF), bringing a new extensibility mechanism. Moreover, P4rt-OVS comes with a P4-to-uBPF compiler, which allows developers to write data plane programs in the high-level P4 language. Our design results in a hybrid approach that provides P4 programmability without sacrificing the well-known features of OVS. The performance evaluation shows that P4rt-OVS does not introduce significant processing overhead, yet enables runtime protocol extensions and stateful packet processing.

***Index Terms*—Programmable data plane, P4, Software switch, OVS, BPF**

I. Introduction

Software switches are a key component of modern virtualized data centers. The software switch, such as Open vSwitch (OVS) [1], plays the role of a hypervisor switch forwarding packets to and from Virtual Machines (VMs) or containers. Hypervisor switches implement a set of network protocols to enable, among others, multi-tenant network virtualization through overlay tunneling, ACL (Access Control List) and QoS (Quality of Service) [2]. Moreover, as network virtualization systems are getting mature, more advanced and complex middlebox functions (such as stateful firewalls or NATs with connection tracking) are being implemented inside software switches to offload VM-based network functions, while preserving their flexibility and performance [3] [4].

Although OVS provides a high degree of programmability through the use of the OpenFlow forwarding model [5], it is still difficult to extend its packet processing pipeline. Developing new network feature requires domain-specific knowledge of network protocol's design, low-level C skills and familiarity with the large and complex codebase of the software switch. Moreover, OVS adopts the stateless forwarding model of OpenFlow, which prevents the implementation of stateful use cases, including many security services.

The domain-specific language (DSL) P4 [6] was invented to address the limitations of OpenFlow and enable the customization of network devices' protocol stack. Not only does P4 allow the description of stateless Match-Action pipelines, but it also exposes persistent memories on the switch, thereby enabling stateful packet processing. In [7], PISCES has been proposed as a programmable, protocol-independent software switch to enhance the level of programmability provided by OVS. PISCES enables custom protocol specification in the P4 language with negligible performance overhead and without the need for direct modifications to the switch codebase. PISCES, however, has two main drawbacks. First, it requires re-compilation every time the P4 program is changed. Therefore, if PISCES were to be used as the hypervisor switch of a network virtualization system, it would cause an outage of the entire infrastructure at each update of the data plane program. In addition, such a design does not allow to inject custom, vendor-specific data plane applications at runtime. Moreover, PISCES does not provide mechanisms to implement stateful data plane programs, and that limits its applications.

In this paper, we present the design and implementation of P4rt-OVS, which allows programming protocol-independent, runtime extensions for a software switch with P4. P4rt-OVS[1] is an original extension of OVS, designed around the following design principles:

Enable runtime programmability. We design our solution to be programmable at runtime. Therefore, we leverage Berkeley Packet Filter (BPF) [8] to provide a runtime extensibility mechanism for OVS.

Provide performance for NFV. OVS is used as a virtual switch in a majority of Network Function Virtualization (NFV) systems. To meet the performance requirements of NFV, the OVS datapath has been ported to DPDK [9]. Therefore, to provide the high performance we have built P4rt-OVS on top of OVS-DPDK.

Support stateful operations. Many network functions require access to the state of connections to fulfill their goal. As the P4 language provides a way to save custom data

*This author performed this work while at Orange Labs.

[1]P4rt-OVS is open-source and available at https://github.com/Orange-OpenSource/p4rt-ovs

ISBN 978-3-903176-28-7

structures in the switch's memory we treat the support for stateful operations as an added value for OVS.

The paper is organized as follows. Section II presents the main features of OVS, P4 and BPF. Then, we motivate our work in section III. Section IV describes the P4rt-OVS design and implementation. We evaluate the performance of P4rt-OVS in section V. In section VI, we discuss the related work. Finally, in section VII, we present conclusions and future work.

II. Background

Our P4rt-OVS prototype enables the upgrade and customization of OVS's network protocol stack at runtime. This is achieved by integrating P4 (providing the high-level language) and BPF, which provides the runtime extensibility mechanism, with OVS.

Open vSwitch. Open vSwitch [1] is widely used in virtualized data centers as a hypervisor switch. It implements a complex protocol's stack to enable multi-tenancy in a virtualized data center. In OVS, two major components participate in packet processing. The *datapath* is the main component responsible for packet forwarding and is also referred to as the *fastpath*. In the case of OVS-DPDK, the datapath component is implemented in userspace. The second component is *ovs-vswitchd*, which is a userspace daemon, and is also called the *slowpath*. It tells the fastpath how to forward incoming packets based on flow rules in Match-Action tables. In OVS, flow caching has been implemented to prevent packet's forwarding to the slowpath for every packet in the flow. Finally, the *ovs-vswitchd* exposes also the OpenFlow interface to external SDN controllers. OVS provides a wide range of OpenFlow actions to modify packets in the fastpath. When it comes to packet's tunneling, OVS uses a concept of *packet's recirculation*. When a packet arrives at the switch, only the outer header is extracted and known to the datapath. Therefore, if there are nested packet's headers, OVS needs to recirculate the packet, i.e., send it back to the beginning of the datapath processing to extract inner headers and allow for further processing.

The BPF virtual machine. BPF is originally a pseudo-code virtual machine in the Linux kernel [8], designed to allow userspace processes to update the kernel's behavior at runtime. For example, BPF programs can be written to rewrite incoming packets or collect statistics every time a given kernel function is called. This runtime programming of the Linux kernel is made possible by an interpreter coupled with a set of JIT compilers: BPF programs are loaded as bytecode and either compiled to assembly code and executed or interpreted directly.

One particularity of BPF is its verifier, a static analyzer that runs at load time to ensure loaded programs are safe to be executed by the kernel, i.e., that they do not contain memory errors or various other faults. In the Linux kernel, an essential extension to BPF added persistent data structures [10], called *maps*, thereby allowing stateful processing in BPF programs. BPF maps are allocated in the kernel, outside the BPF VM, and accessed from the BPF VM through special functions, called *helpers*. These helpers are necessary whenever a BPF program needs to perform an action restricted by the VM. They implement safety checks to read memory from maps, retrieve the current time, update kernel data structures, etc.

Since P4rt-OVS processes packets in userspace with DPDK, it cannot use Linux's BPF VM. Fortunately, several userspace implementations of the BPF VM exist, with different supported features. P4rt-OVS relies on the userspace BPF VM implemented for the Oko software switch [11]. This particular implementation supports maps, can JIT compile programs to x86-64, and includes a limited static analyzer.

The P4 language. P4 [6] is the framework for programming protocol-independent packet processors. In particular, it provides a Domain-Specific Language (DSL) for expressing how a data plane of a programmable element (e.g. hardware or software switches, network interface cards) should process packets. In the P4 language, a programmer defines a set of supported network protocols and the behavior of a network device's data plane in a high-level and declarative manner. Then, the P4 program is translated by specialized compilers into a code representation consumable by the programmable device (called the *P4 target*). The most recent version of the language, $P4_{16}$ [12], allows for a wide range of P4 targets. Each target's manufacturer has to define an abstract forwarding model and target-specific capabilities (called *P4 externs*) in the form of *the architecture model*, thanks to which a programmer knows how to write P4 programs for a given platform. A typical forwarding model has at least one *programmable parser* (represented as a cyclic graph) to extract headers and at least one *programmable deparser* to fill a packet's fields before sending a packet to the wire. Moreover, the P4 architecture model can specify the number of *Control blocks*, which are composed of a set of Match-Action tables. The Control block is used to implement a packet processing pipeline (packet headers' modification, tunneling, stateful operations, etc.). At runtime, a control plane (via P4Runtime protocol) can add, modify or remove table entries, but it can also change the forwarding pipeline of a programmable device by installing a new P4 program.

III. Motivation

We believe that our solution can be used to implement many use cases, which are not currently supported by OVS. To motivate our work, we present examples of use cases, which are not provided by OVS, but can be described in the P4 language. Thus, they can show the added value of P4rt-OVS. The selected use cases can be divided into two categories: stateful data plane programs and custom, domain-specific protocol extensions.

A. Stateful data plane programs

Stateful firewall. Recent firewalls perform stateful analysis of packets to keep track of a transport connection. It enables firewalls to allow all packets from established connections through and to reject new connections that are not in the ACL list. The ACL is a stateless component and can be implemented

by static OpenFlow rules. However, tracking TCP connections requires stateful operations to store and update the state of a session. The P4 language, among other features, provides the stateful construct - the P4 register. By using P4 registers, users can describe the connection tracking component of a firewall. Thus, our solution enables implementing a stateful firewall on top of OVS.

Rate limiter. Where OVS provides support for per-port rate limiting, P4rt-OVS provides custom, arbitrary rate-limiting algorithms by leveraging P4 registers. Moreover, P4rt-OVS's rate limiters may be applied in a per-flow manner.

In-network DDoS mitigation. As P4 is being considered to implement DDoS detection and mitigation [13], it could also be implemented using P4rt-OVS. Offloading anti-DDoS applications from separate boxes (virtual machines) to the virtual switches inside the data center may also significantly improve the overall performance of the network function [3] [14].

B. Domain-specific protocol extensions

OVS is widely used in telecommunication use cases, which rely on specific protocol stacks. Although some extensions to implement telco-specific protocols have already been proposed, they were not integrated with OVS. Therefore, it leads to vendor-specific forks of OVS. In this section, we present examples of telecommunication protocols that could be implemented in the P4 language and injected into P4rt-OVS.

5G User Plane Function. OVS (with GPRS Tunneling Protocol extension) has been used in a virtual Serving and Packet Data Gateway (SPGW) of LTE as the packet forwarding engine. According to the 5G specification, the GTP protocol was chosen as the encapsulation protocol [15], but other technologies are also considered [16]. P4rt-OVS allows to describe any encapsulation technique in the P4 language and integrate it (even at runtime) with OVS. It results in a shorter time to market for new protocols.

BNG. The Broadband Network Gateway (BNG) provides an access gateway for the fixed-network subscribers. Except for control plane operations, the BNG handles data plane operations such as VLAN tagging, PPPoE and MPLS tunneling. With the use of P4rt-OVS, all of these data plane techniques can be implemented in P4. As a result, OVS can be extended and used to implement the data plane of BNGs.

IV. P4RT-OVS DESIGN AND IMPLEMENTATION

A. Overview

Figure 1 depicts the proposed extensions to the OVS architecture that enable programming the packet processing pipeline at runtime. It also shows the P4rt-OVS programming workflow. First of all, we have extended the userspace datapath of OVS with an additional BPF subsystem, which enables the injection of packet forwarding programs at runtime and their integration with the OVS forwarding pipeline. The BPF subsystem consumes bytecode that implements the packet processing model. In our framework, the P4-to-uBPF compiler is utilized to generate bytecode from the P4 program. Moreover, apart from OpenFlow, we have built the P4Runtime abstraction layer (P4RT-AL). It allows the integration of P4Runtime-compliant SDN controllers with P4rt-OVS. It results in a hybrid approach, which can be controlled by both OpenFlow and P4Runtime control protocols.

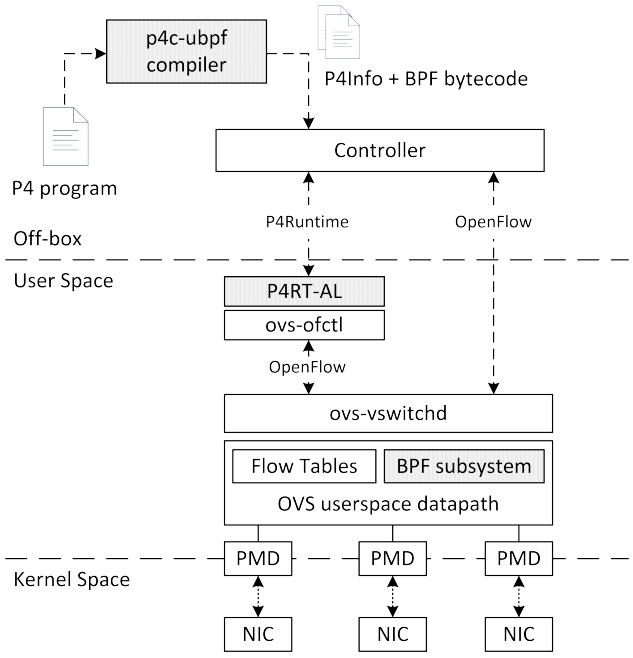

Fig. 1: The overall architecture of P4rt-OVS

The P4rt-OVS programming workflow assumes the P4rt-OVS has been compiled and run beforehand and is as follows. In the first step, the programmer designs and creates the P4 program implementing specific network features. According to the $P4_{16}$ specification, we also provide the P4 architecture model, which guides the programmer on how to write a data plane program for P4rt-OVS. Next, the user generates BPF bytecode, the data plane code, using the P4-to-uBPF compiler and, optionally, the P4Info metadata file to be used by the control plane as a contract describing the data plane implementation. Then, the BPF bytecode is injected in the OVS forwarding pipeline by either the SDN controller or via local CLI using the P4Runtime protocol. The data plane program appears in the switch as the BPF program with a new identifier. The last step for the user is to define the OpenFlow flow rule that will invoke the appropriate BPF program. The user can also configure BPF map entries for the BPF program before configuring a flow rule or when the BPF program is already in action. To modify a data plane program a user can create a new BPF program, inject it with a new identifier and modify flow rules to point to the new BPF program.

B. Modifications to OVS

We made four modifications to OVS in order to enable programming protocol-independent, runtime extensions using P4.

The BPF subsystem for OVS. The first design principle was to provide the runtime extensibility mechanism to OVS. We extended OVS with a new subsystem based on the userspace BPF (uBPF) implementation (see Section II). We retain the BPF infrastructure (abstract machine, BPF verifier and set of external functions) of Oko [11], but we also introduced several modifications needed to implement certain P4 capabilities.

Unlike the Oko [11] approach, we propose to pass the whole `dp_packet` structure, which represents a packet inside the userspace datapath. The `dp_packet` structure contains various information about a packet and not all of them are needed by the BPF program. However, such a design is necessary to implement arbitrary packet tunneling, which we will explain further in this subsection. The BPF program takes the `dp_packet` structure as an argument, which does not represent a packet data directly. Hence, for packet data to be processed by the BPF program, we have implemented a new uBPF helper, `ubpf_packet_data()`, which retrieves a packet's data from the `dp_packet` structure. Such a design requires a modification to the uBPF verifier to prohibit illegitimate accesses to the `dp_packet` structure.

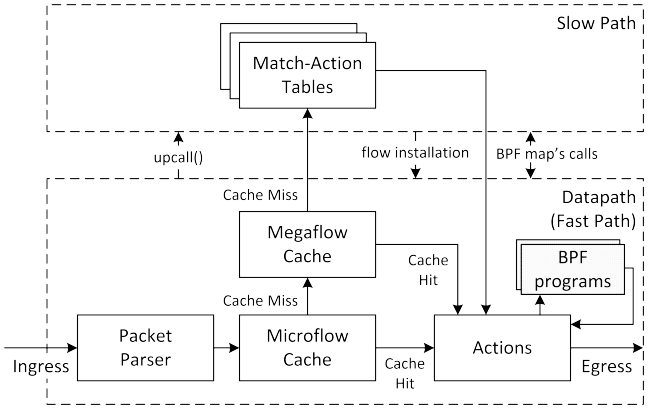

Fig. 2: The BPF subsystem and forwarding model of OVS

Programmable actions. The implementation of the Oko switch [11] assumes that the BPF programs are used as an enhanced filtering program to match packets. With such a design a user cannot modify packets, whether to write a packet field or to encapsulate them. In contrast, we design P4rt-OVS to allow for packet modification. Therefore, P4rt-OVS executes BPF programs as an OpenFlow action by implementing the new action type *OVS_ACTION_ATTR_EXECUTE_PROG*. Figure 2 presents this new design for the OVS forwarding model. In addition, contrary to Oko, because we integrate BPF programs as OpenFlow actions, the flow caching architecture stays unaffected. Therefore, BPF can be integrated without significant modifications to the OVS forwarding model and all the actions defined in the P4 program are executed in the fastpath.

However, with the introduction of programmable actions, we face a new problem when dealing with multi-tables pipelines. If the BPF program modifies a packet, the OVS datapath should recirculate the packet, but the datapath does not have any information about how the packet is processed inside the BPF program. Therefore, the datapath cannot decide to recirculate the packet or not. The current implementation assumes the BPF program is always executed in the last table as the last action, so that recirculations are unnecessary. Nevertheless, there are other options to address this problem. The first option could be to leverage the BPF verifier to pass information to the OVS datapath, i.e., whether the BPF program performs packet modification. Another approach could be to force the BPF program's programmer to set the recirculation flag any time a packet is modified [3]. However, it requires programmers to understand the recirculation problem and its exact implications on packet processing. To conclude, the described problem should be the case for further improvements of P4rt-OVS.

Support for tunneling. An inseparable feature of the P4 language is support for arbitrary packet encapsulation. It requires two modifications to the BPF subsystem and the OVS datapath. First, already satisfied is the permission for packet modification in the BPF verifier. Second is the packet's length adjustment. This mechanism allows BPF programs to change the length of packets (usually adds zero bytes or removes bytes from the head of the packet) before sending it back to the OVS pipeline. The packet adjustment requires access to the `dp_packet` structure. Therefore, to support arbitrary tunneling, we added a new uBPF helper function, `ubpf_adjust_head()`, which has access to `dp_packet` and adjusts the packet's length according to the *offset* value passed as the argument to the helper.

Exposing the interface to the BPF subsystem. The next extension we make is an implementation of the set of functions for OVS that exposes the interface to the BPF subsystem. In fact, these functions implement additional OpenFlow messages that can be used to manage BPF programs and their maps. These messages are as follows.

- *LOAD_BPF_PROG* to install a new BPF program.
- *UNLOAD_BPF_PROG* to remove an existing BPF program with a given identifier.
- *SHOW_BPF_PROG* to list all BPF programs or show the information about a given BPF program.
- *UPDATE_BPF_MAP* to add or update an existing entry of the BPF map.
- *DUMP_BPF_MAP* to dump the content of the BPF map of a given BPF program.
- *DELETE_BPF_MAP* to remove an entry with a given key from the BPF map.

C. The P4 to uBPF compiler

In compliance with the $P4_{16}$ compiler's design [12] our P4-to-uBPF compiler implements a new backend, userspace BPF, for the compiler's frontend. The P4-to-uBPF compiler generates target-specific, restricted C code, that is compatible with uBPF and can be further compiled to the BPF bytecode using the *Clang* compiler. This one intermediate stage allows us to

leverage the existing compiler optimizations implemented by *Clang*.

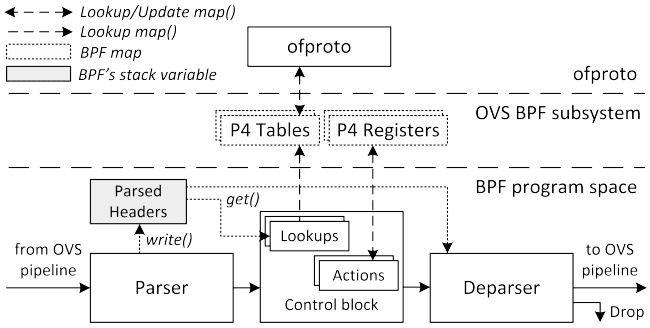

Fig. 3: The forwarding model of the BPF program generated from the P4 language

Along with the implementation of the P4-to-uBPF compiler, we have designed the architecture model for P4rt-OVS, which in particular, describes the forwarding model of the uBPF program. The forwarding model is tailored to the architecture of OVS and is depicted in Figure 3. In the architecture model, we elect to prevent P4 programs from forwarding packets themselves; it is therefore still the responsibility of OVS to determine an output port for a packet. As a result, the P4/BPF program can filter, inspect or modify a packet, but the only forwarding decision it can make is to decide whether to drop a packet or send it back to the OVS forwarding pipeline. This decision is due to the usage model of P4 in P4rt-OVS. Originally, P4 is designed to describe the whole functionality of a switch. In our case, P4 is used only to describe a specific part of a switch, a custom action. Thus, the OVS action can still be used to perform packet forwarding and the P4 language is just utilized to play its role: providing a high-level language to process packets' headers.

The forwarding model of the BPF program generated from the P4 language consists of three packet processing blocks: parser, control block and deparser.

Parser. The ingress block, the Parser, is responsible for reading a packet's headers and copying them to the `Headers_t` structure. The Parser reads each header field by field by loading bits and shifting or masking bits, if necessary. The output of a parsing stage is the `Headers_t` structure filled with a packet headers' data. Additionally, for each header in the `Headers_t` structure, a validity bit is associated. According to [17], if a header has been parsed correctly, the validity bit is set. The validity bit is further used to perform operations on headers (encapsulation or decapsulation) in Control Block and Deparser.

Control block. The control block is composed of a set of Match-Action tables implementing a packet processing. In our design, to implement a packet processing pipeline, a programmer can use read-only Match-Action tables (for stateless network functions) or registers with read-write permissions to implement stateful network applications. The P4rt-OVS provides two uBPF helpers, `ubpf_map_lookup()` and `ubpf_map_update()`, to read from the BPF map (table or register) and write to the BPF map (only registers) respectively. Since P4 tables and registers are implemented as BPF hashmaps, both `ubpf_map_lookup()` and `ubpf_map_update()` have an average-case complexity of O(1). In particular, in the Control block, the P4 program can encapsulate or decapsulate a packet by validating (`setValid()` operation) or invalidating (`setInvalid()` operation) the validity bit of a packet's header, respectively. The validity bit is further used in the Deparser to define the order of headers for an outgoing packet.

Deparser. Its function is to prepare a packet to be sent back to the OVS pipeline. In particular, it is responsible for modifying the packet's headers and performing an arbitrary packet encapsulation. The current design of the P4-to-uBPF compiler uses *post-pipeline editing*. It means that all modifications of the packet's headers are made in the Deparser. The intermediary Match-Action tables modify the header's metadata structure, which is further used to generate an outgoing packet. To perform an arbitrary encapsulation the Deparser makes use of the `ubpf_adjust_head()` helper, to adjust the length of a packet. Before adjusting a packet's head, the offset is calculated. If it is negative, bytes are removed from the head of a packet. Otherwise, zero bytes are added to the front of a packet. In comparison to previous versions of the language, $P4_{16}$ requires an explicit definition of the Deparser. Thus, the programmer has to define the order of headers for the outgoing packet in the P4 code. Then, the Deparser fills in the packet's payload with data from the `Headers_t` structure. As described above, the Deparser decides to append a particular header based on the validity bit associated with each packet's header.

As is the case with other P4 compilers, the P4-to-uBPF compiler also generates the P4Info metadata, which can be used by the P4Runtime-based control plane to interface with Match-Action tables.

D. The P4Runtime-based control plane

To effectively use P4rt-OVS via an external SDN controller a user needs to leverage both OpenFlow and P4Runtime protocols in conjunction. It is the result of the hybrid design we decided to follow for P4rt-OVS. We extend the OpenFlow protocol to support a new OpenFlow action (*prog*) in the FLOW_MOD message. The *prog* action invokes a given BPF program for packets matching the corresponding flow rule. Our implementation provides also all the OpenFlow messages listed in the last paragraph of subsection IV-B. We also extend the P4Runtime protocol. We introduce a new usage model for P4 devices: P4rt-OVS may be configured with multiple P4 programs, each of them describing a separate forwarding element. Therefore, in order to support multiple P4 programs, we customized the P4Runtime protocol by introducing a new field for the P4Runtime messages, *pipeline_id*. The *pipeline_id* field defines the P4 pipeline inside the P4 target. Thus, if the P4 target (identified by the *device_id*) supports multiple P4 pipelines running simultaneously, the P4Runtime controller

can use the *pipeline_id* field to refer to a given P4 pipeline. We have also implemented the P4Runtime abstraction layer (P4RT-AL) as a proof-of-concept Python application, which allows to control BPF programs using the P4 semantics.

V. Performance Evaluation

In this section, we compare the packet processing performance of P4rt-OVS, OVS [1] (the reference implementation) and PISCES [7] (the state-of-the-art implementation of P4-capable OVS). The goal of the evaluation was to check 1) whether P4rt-OVS introduces any performance overhead and 2) how P4rt-OVS performs in comparison to the other solutions. Moreover, we measure the overhead and efficiency of P4 extensions to OVS through a set of microbenchmarks.

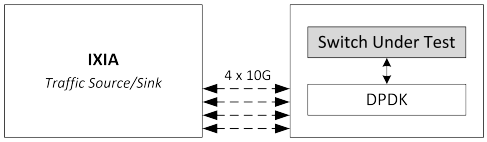

Fig. 4: The test topology

A. Evaluation environment

Figure 4 shows the test topology. The Switch Under Test (SUT) running on top of the DPDK framework is installed on the HP ProLiant DL380 Gen9 server equipped with 2x Intel(R) Xeon(R) CPU E5-2690 v3 running at 2.60GHz, with 128 GB RAM and two dual-port Intel 82599ES 10GE NICs. We use a IXIA hardware traffic generator connected directly to the ports of the server, so that SUT handles a total of 40 Gbps of traffic. For all experiments, we configured Linux (Ubuntu 16.04) to isolate DPDK cores from the Linux scheduler. The DPDK framework has been configured with 4 receiving queues per port running on 2 physical (4 logical) CPU cores. In all experiments, IXIA generates four traffic flows per port to test SUT with multiple flows. To measure the results, we use the methodology described in RFC2544 [18] (we assume 0.002 % packet loss). In end-to-end comparisons, each experiment lasts 60 seconds and we report the mean throughput in millions of packets per second (Mpps) and the 95% confidence interval over 10 runs. In the microbenchmarks, we measure CPU cycles per packet using the machine's time-stamp counter (TSC).

B. End-to-end performance

We next measured the end-to-end performance of example network functions to illustrate the cost of the P4 programmability in a near-realistic scenario.

Overhead evaluation. First, we have conducted the experiment to evaluate the overhead of introducing P4 extensions to OVS. For this purpose, we compared simple L2 forwarding performance of OVS and P4rt-OVS with the baseline P4 program performing no operations on packets. Thus, the experiment shows just the overhead introduced by our new OVS action, which invokes the BPF program to handle packets. We measured throughput rate in two scenarios: with microflow cache enabled and disabled. The results are shown in Figure 5. As expected the average throughput was slightly higher when microflow cache was enabled. However, the most important observation is that invoking the BPF program to process a packet introduces a negligible overhead. Nevertheless, a more complex packet processing pipeline of the P4 program could increase the overhead. Therefore, the impact is analyzed in the next paragraph.

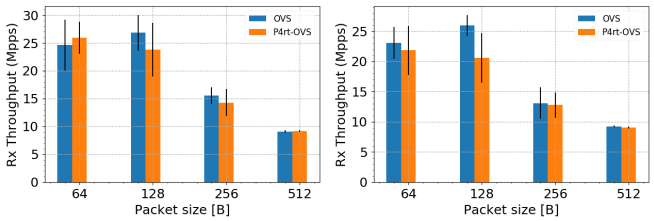

Fig. 5: L2 Forwarding performance in MPPs for input traffic of 40 Gbps: a) microflow enabled, b) microflow disabled

Packet processing evaluation. We selected a few scenarios, in which different packet operations (complex packet parsing, tunneling, packet modifications) are done to compare the performance in a near-realistic environment. We compared the performance of three solutions: P4rt-OVS, OVS and PISCES to check how P4rt-OVS performs compared to these state-of-the-art solutions. We measured the performance of the following network functions:

- **Static Destination NAT (DNAT)** that matches a destination IPv4 address and translates it based on static flow rules.
- **Static Network Address Port Translation (NAPT)** that matches source and destination IPv4 addresses and source and destination UDP ports and translates them based on static flow rules.
- **MPLS Label Edge Router (LER)** that matches an ingress port and destination IPv4 address and attaches the MPLS label to a packet.

Figure 6 shows the performance results for SNAT, MPLS LER and NAPT respectively. Based on these results, we can conclude that the performance of P4rt-OVS is comparable to both OVS and PISCES for packet sizes of 64, 128 and 256 bytes. The mean throughput with 64B packets for MPLS LER in P4rt-OVS is lower than for OVS or PISCES; it may be caused by the overhead introduced by the `ubpf_adjust_head()` helper, which adds zero bytes to the head of a packet before sending it to the wire. For larger packets (512 bytes) the performance of P4rt-OVS is comparable to OVS and higher than the performance of PISCES. The results show that the overhead of more realistic P4 extensions to OVS is negligible in terms of the obtained throughput rate.

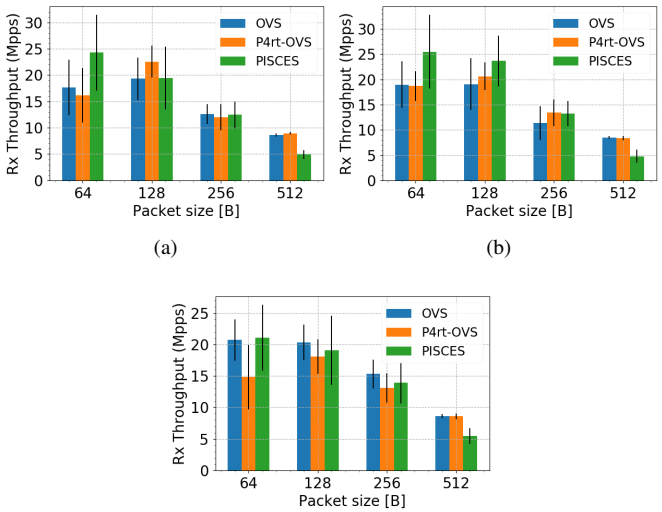

Fig. 6: The performance in Mpps of a) SNAT, b) NAPT, c) MPLS LER

C. Microbenchmarks

As a next step, we evaluated each component of the BPF program (Parser, Deparser, Match-Action pipeline) separately. Moreover, we measured how much overhead the P4 programmability introduces in comparison to writing the BPF program in the C language.

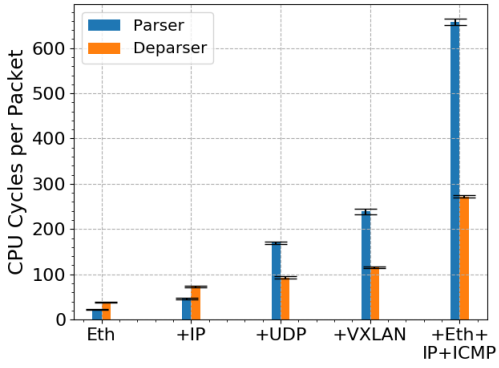

Fig. 7: The performance of Parser and Deparser as more protocols are handled.

Parser and Deparser performance. Figure 7 shows how CPU cycles per packet increase for both Parser and Deparser as the P4 program handles additional protocols. To parse only the Ethernet header, Parser consumes about 20 CPU cycles per packet, while Deparser consumes twice as many cycles, about 40 per packet. The deparsing process is more costly if there are a few protocols handled (up to 2). However, as the P4 program handles more protocols (layer 4 and above) the parser stage becomes more costly. The cost of Parser for a protocol stack composed of seven protocol's headers is about 2.4 times greater than the cost of Deparser. It means that the performance results may be degraded for the P4 programs handling more complex protocol stacks.

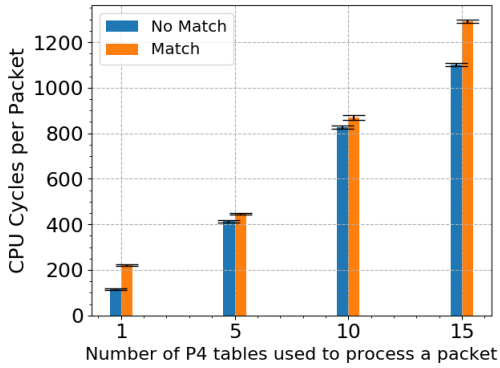

Fig. 8: Performance of the control block as more Match-Action tables are used to process a packet.

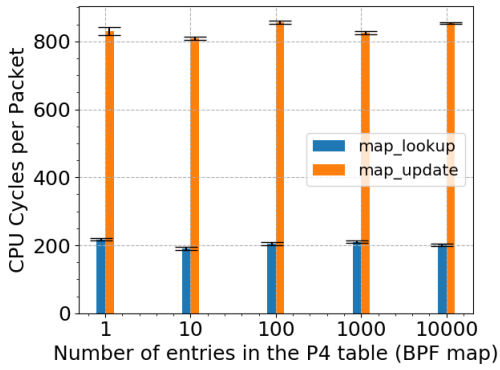

Fig. 9: Performance of table operations as more table entries are added.

Match-Action pipeline's performance. The packet processing pipeline is described in the Control block of the P4 program. The Control block may be composed of multiple Match-Action tables. As our compiler generates the code that modifies packets in the *post-pipeline editing* a cost of a write action (set or modify a field) is included in the cost of the Deparser. Therefore, the main factors that impact the performance of the packet processing pipeline are operations on Match-Action tables. Figure 8 shows how many CPU cycles are required for Match-Action table's operations (lookup, write action) as more tables are used to process a packet. As expected, the cost of Match-Action table's operations grows (almost linearly), when the number of Match-Action tables is increased. If there is a match in the table lookup, appropriate action is invoked. However, the overhead of invoking an action is negligible due to *post-pipeline editing*. Nevertheless, a programmer should try to minimize the number of P4 tables to optimize the performance of the P4 program injected to P4rt-OVS. We also measured how many CPU cycles per packet are

required to perform table lookup and update (Figure 9). As expected, the cost of these operations is constant regardless of the number of entries stored in the P4 table. It is a consequence of using hash maps to implement P4 tables. However, we can observe that `ubpf_map_update()` uses many more CPU cycles per packet than map lookup. This is due to the memory allocation required to add new entries to the hash map.

Overhead of the P4 programmability. The data plane programs for P4rt-OVS do not necessarily have to be developed in P4. The skillful programmer may also use the C language with standard userspace libraries to implement runtime extensions for P4rt-OVS. P4 provides an expressive, declarative, high-level language, but a protocol-independence and programmability come with the cost of a more complex program structure and costly parsing and deparsing stages. In this experiment, we compare how many CPU cycles are consumed by the BPF program generated from P4 in comparison to the C-based program.

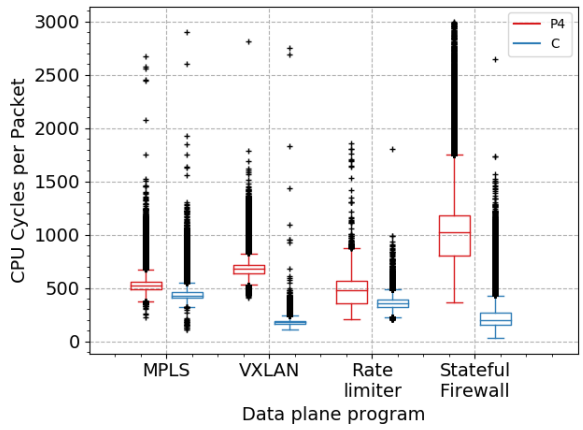

Fig. 10: The performance of data plane programs written in P4 and C.

Figure 10 depicts the performance results for several programs written in either P4 or C. It shows that P4 introduces the overhead in comparison to C programs, especially for VXLAN and stateful firewall. Based on these results, we can conclude that the performance of a data plane program performing MPLS tunneling is comparable for both C and P4. For this kind of program, a protocol stack is quite simple and, therefore, the cost of parsing and deparsing is low. However, we can observe significant overhead when comparing results for the VXLAN tunneling example. As observed in Figure 7 the cost of parsing and deparsing in P4 increases as more protocols are handled. This does not apply to the C program, as there is no need to perform a costly deparsing process. We also measured the performance for two stateful programs, namely the rate limiter and the stateful firewall. The performance of C and P4 implementations of the former are comparable because there is no need to implement parser and deparser for simple rate limiting in P4. However, the performance of a stateful firewall written in C is higher than the corresponding P4 program. A stateful firewall tracks the state of the TCP connection, so the P4 program must parse headers up to layer 4. Again, due to the high cost of parsing and deparsing stage, the stateful firewall written in P4 performs worse. Note also that the performance of a particular BPF program depends strongly on how the function is implemented and the results can vary considerably from a program to a program. For instance, we have implemented stateful firewall in C such that the packet processing is finished just after saving the state of connection. On contrary, the program generated from P4 always reaches Deparser (as the P4 language does not provide explicit keyword to stop execution such as `return` in C). This is also the reason why C-based stateful firewall performs better. In the case measurements considered for the stateful firewall there are many outliers. These are mainly values for `map_update()` operations as they are quite rare (an update of the state is done only when a session's state is changed); once the session is open `map_update()` is not invoked until the end of the session. To conclude, P4 introduces a notable performance overhead in comparison to C programs. It is mostly caused by the costly parsing and deparsing process performed in the BPF program generated from the P4 language. Hence, there is room for performance optimizations of our P4-to-uBPF compiler by generating more efficient Parser and Deparser.

VI. Related work

P4-capable software switches. P4rt-OVS leverages P4 as a high-level language to describe the packet processing pipeline. Although similar approaches exist [19] [20], P4 has so far been the focus of the industry. However, the design principles of P4rt-OVS may also be used with other data plane programming technologies. There were already some attempts [21] [7] [22] to implement P4-capable software switches. Nevertheless, P4rt-OVS is the first solution, which allows to implement protocol-independent and stateful programs for OVS and deploy them at runtime. P4rt-OVS provides the P4-to-uBPF compiler that extends a range of already developed P4 compilers for various targets [23] [7] [24] [25] and is the first P4 compiler for userspace BPF target. The P4-to-uBPF compiler can be used to implement P4 support for other software switches processing packets in userspace.

Other programmable runtime environments. There is a large body of previous work on programmable software switches [26] [27] [28]. From the perspective of this paper, it is worth to outline two extensions of OVS: SoftFlow [3] and Oko [11]. The former allows to execute network functions as OVS actions, but is not programmable at runtime. The latter integrates BPF with OVS and can be extended at runtime. However, it is limited only to programmable packet filters. There is also an effort on extending the Linux kernel's networking stack at runtime [8] [29]. eBPF was applied to OVS [30] [31] to provide runtime extensibility, but the results were not satisfying due to limitations of in-kernel BPF.

Performance of software switches. As software switches are widely used as hypervisor switches in virtualized data

centers, there are many research papers on performance comparison of different solutions [32] [33] [34] [35]. OVS-DPDK was proved to have a satisfying performance for NFV, hence we decided to base P4rt-OVS on it.

VII. Conclusion

Even though OVS provides a high degree of programmability to the data center networking, it is still difficult to extend its packet processing pipeline to implement novel or domain-specific network protocols or stateful data plane programs. In this paper, we present our design and implementation of P4rt-OVS, an original extension of OVS that allows for programming protocol-independent and stateful runtime extensions for OVS. P4rt-OVS offers network engineers a flexible architecture to introduce new network features to OVS's forwarding pipeline dynamically and, therefore, to shorten the time to market for network protocols. The obtained performance evaluation results show that P4rt-OVS introduces a negligible overhead. Moreover, the microbenchmark provided tips on how to write efficient P4 programs for P4rt-OVS. Nevertheless, as microbenchmarks proved, there is still room for performance optimizations.

Our future work will focus on implementing further functional enhancements and performance optimizations. Moreover, we plan to implement new use cases (e.g. In-Band Network Telemetry) to evaluate further P4rt-OVS and its potential limitations.

References

[1] B. Pfaff *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, pp. 117–130.

[2] T. Koponen *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, pp. 203–216.

[3] E. J. Jackson *et al.*, "SoftFlow: A Middlebox Architecture for Open vSwitch," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, 2016, pp. 15–28.

[4] H. Mekky *et al.*, "Network function virtualization enablement within SDN data plane," in *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, May 2017, pp. 1–9.

[5] N. McKeown *et al.*, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[6] P. Bosshart *et al.*, "P4: Programming Protocol-independent Packet Processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[7] M. Shahbaz *et al.*, "PISCES: A Programmable, Protocol-Independent Software Switch," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 525–538.

[8] J. Corbet, "BPF: the universal in-kernel virtual machine," 2014. [Online]. Available: https://lwn.net/Articles/599755/

[9] Intel DPDK, "Data Plane Development Kit." [Online]. Available: www.dpdk.org

[10] J. Corbet, "Extending extended BPF ," 2014. [Online]. Available: https://lwn.net/Articles/603983/

[11] P. Chaignon *et al.*, "Oko: Extending Open vSwitch with Stateful Filters," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '18. New York, NY, USA: ACM, 2018, pp. 13:1–13:13.

[12] M. Budiu and C. Dodd, "The P416 Programming Language," *SIGOPS Oper. Syst. Rev.*, vol. 51, no. 1, pp. 5–14, Sep. 2017.

[13] Y. Afek *et al.*, "Network anti-spoofing with SDN data plane," in *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, May 2017, pp. 1–9.

[14] P. Chaignon *et al.*, "Offloading security services to the cloud infrastructure," in *Proceedings of the 2018 Workshop on Security in Softwarized Networks: Prospects and Challenges*, ser. SecSoN '18. New York, NY, USA: ACM, 2018, pp. 27–32.

[15] 3GPP, "5G System – Phase 1; CT WG4 Aspects," 3rd Generation Partnership Project (3GPP), Technical Report (TR) 29.891, 12 2017, version 15.0.0.

[16] K. Bogineni *et al.*, "Optimized Mobile User Plane Solutions for 5G," Working Draft, IETF, Internet-Draft, June 2018.

[17] P4.org, "P4_16 language specification," https://github.com/p4lang/p4-spec/tree/master/p4-16/spec, December 2019.

[18] S. Bradner and J. McQuaid, "Benchmarking Methodology for Network Interconnect Devices," RFC 2544, March 1999.

[19] S. Li *et al.*, "Protocol oblivious forwarding (pof): Software-defined networking with enhanced programmability," *IEEE Network*, vol. 31, no. 2, pp. 58–66, March 2017.

[20] S. Pontarelli *et al.*, "FlowBlaze: Stateful Packet Processing in Hardware," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 531–548.

[21] B. Pfaff. (December 2019) Proposal of P4 support for OVS. https://github.com/blp/ovs-reviews/tree/p4.

[22] S. Choi *et al.*, "The case for a flexible low-level backend for software data planes," in *Proceedings of the First Asia-Pacific Workshop on Networking*, ser. APNet'17. New York, NY, USA: ACM, 2017, pp. 71–77.

[23] H. Wang *et al.*, "P4FPGA: A Rapid Prototyping Framework for P4," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 122–135.

[24] S. Laki *et al.*, "High Speed Packet Forwarding Compiled from Protocol Independent Data Plane Specifications," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 629–630.

[25] F. Ruffy *et al.*, "P4C-XDP: Programming the Linux Kernel Forwarding Plane Using P4," in *Linux Plumbers Conference*, Vancouver, 2018.

[26] E. Kohler *et al.*, "The Click Modular Router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, p. 263–297, Aug. 2000.

[27] A. Panda *et al.*, "Netbricks: Taking the v out of nfv," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 203–216.

[28] J. Hwang *et al.*, "NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms," *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, pp. 34–47, March 2015.

[29] T. Høiland-Jørgensen *et al.*, "The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18. New York, NY, USA: ACM, 2018, pp. 54–66.

[30] C.-C. Tu *et al.*, "Building an Extensible Open vSwitch Datapath," *SIGOPS Oper. Syst. Rev.*, vol. 51, no. 1, pp. 72–77, Sep. 2017.

[31] W. Tu *et al.*, "Bringing the Power of eBPF to Open vSwitch," in *Linux Plumbers Conference*, Vancouver, 2018.

[32] T. Zhang *et al.*, "Comparing the Performance of State-of-the-Art Software Switches for NFV," in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, ser. CoNEXT '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 68–81.

[33] G. Lettieri *et al.*, "A Survey of Fast Packet I/O Technologies for Network Function Virtualization," in *High Performance Computing*. Springer, 2017, pp. 579–590.

[34] M. Paolino *et al.*, "SnabbSwitch user space virtual switch benchmark and performance optimization for NFV," in *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, Nov 2015, pp. 86–92.

[35] V. Fang *et al.*, "Evaluating Software Switches: Hard or Hopeless?" EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2018-136, Oct 2018.