P4 in Open vSwitch with OFP4

Ben Pfaff, Debnil Sur, Leonid Ryzhyk, Mihai Budiu
{bpfaff, dsur, lryzhyk, mbudiu}@vmware.com
VMware

Abstract

Software implementations of P4 available today have significant limitations. Given that, we introduce OFP4, a prototype of an implementation of P4, including P4Runtime support, that uses Open vSwitch as its back-end. OFP4 translates P4 code plus runtime entities such as table entries into OpenFlow (OF) flows, which it installs in a running Open vSwitch instance using the OpenFlow protocol. This paper describes how this translation works and provides an overview of our proof-of-concept implementation.

1. Introduction

Despite 8 years of work on P4 [5], there are currently few high-performance software switches available. This paper introduces OFP4, a prototype implementation of P4 [2] that uses Open vSwitch [14], which is a widely used OpenFlow [9, 13] switch implementation, as its data plane. OFP4 is a daemon with a P4Runtime [11] front-end for the use of an SDN controller and an OpenFlow back-end to implement switching via Open vSwitch.

OFP4 was motivated by the limitations of the software implementations of P4 available today. The best known, BMv2 [4], is meant for accurate simulation, not high speed packet processing. We found ourselves unable to build and install the T_4P_4S [19] academic implementation on any system other than the exact distribution of Ubuntu used by the developers. The PISCES [17] academic project is unmaintained and lacks P4Runtime support. The P4 switch sample application that accompanies DPDK [3] holds promise, but it is DPDK-specific and lacks P4Runtime support. Finally, the ebpf-psa backend is currently under construction [10]. Other software implementations of P4, e.g. [6], have similar limitations.

The following section describes how OFP4 maps from P4 to OpenFlow. Section 3 describes our proof-of-concept implementation of OFP4. Section 4 describes the testing we've done so far. Section 5 states our conclusions and directions for future work.

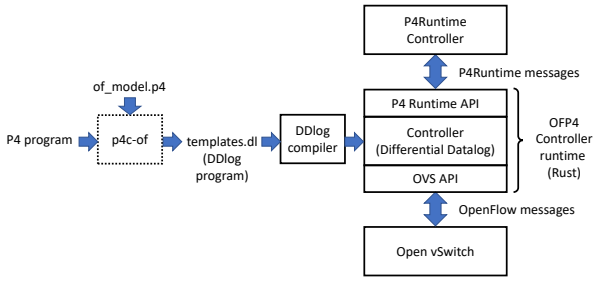

Figure 1: System architecture.

Source Code The source code for our OFP4 prototype is available under the MIT license at `https://github.com/vmware/nerpa` in the `ofp4` subdirectory.

2. Translating P4 to OpenFlow

Both P4 and OpenFlow enable SDN controllers to control packet processing using a programming model based on match-action tables. OpenFlow exposes a relatively rigid architecture in comparison to P4, which even allows modeling new packet formats through packet parsers. It may seem a strange choice to implement a flexible model (P4) on top of a more constrained one (OF). However, previous work [20] has shown that P4 can be used as a *specification language* even for fixed-function switching pipelines, which are not programmable at all! In building OFP4, we design a specific P4 architecture that models the capabilities of OpenFlow, in particular with Open vSwitch extensions. A sketch of our P4 architecture file is given in Section A in the Appendix. Figure 1 shows the architecture of our system. The box shown with dotted line is under construction.

Protocols. P4 supports programmable parser and deparser blocks, which enable developers to support both standard or experimental network protocols. In contrast, OpenFlow and Open vSwitch support a fixed (though broad) set of protocols. Thus, OFP4's P4 architecture lacks parser and deparser blocks, supporting only a fixed set of protocols, which are supplied as part of the `of_model.p4` architecture definition.

Pipelines. OpenFlow exposes a single pipeline, whereas hardware switches tend to have ingress and egress pipelines separated by buffering logic that replicates packets for multicast, mirroring, and other purposes. Unlike parser and deparsers, though, P4's separate pipelines can be implemented in OpenFlow with Open vSwitch extensions. We chose to implement an ingress and egress pipeline because this makes OFP4 architecture more similar to the widely known Portable Switch Architecture [12] (PSA) and the older `v1model` [8] architecture for P4.

Metadata. P4 supports user-defined named metadata and local variables, whereas OpenFlow and Open vSwitch provide only a number of fixed-size generic metadata "registers", mainly `reg0` through `reg15`. The P4 compiler can map the former to the latter, e.g. given the following P4 metadata structure definition:

```
struct metadata {
    bit<12> vlan;
    bool flood;
}
```

P4C-OF might map `vlan` to `reg4[0..11]`, that is, the low 12 bits of OVS metadata field `reg4`, and `flood` to `reg5[0]`.

PSA and `v1model` also supply "standard" metadata with each packet. OpenFlow and Open vSwitch provide some of the same standard metadata, e.g. the packet's ingress port, for OFP4 to use directly. P4C-OF can allocate those without equivalents (e.g., `standard_metadata.egress_spec` in the ingress pipeline) to registers.

```
action Drop()
    { mark_to_drop(standard_metadata); exit; }
action SetVlan(VlanID vid) { meta.vlan = vid; }
action UseTaggedVlan() { meta.vlan = hdr.vlan.vid;}
table InputVlan {
    key = {
        standard_metadata.ingress_port: exact;
        hdr.vlan.isValid(): exact;
        hdr.vlan.vid: optional;
    }
    actions = { Drop; SetVlan; UseTaggedVlan; }
    default_action = Drop;
}
```

Figure 2: Excerpt of a P4 program for VLAN ingress processing.

Table Entries. A P4 table can be translated essentially one-to-one into an OpenFlow table, with each P4 table entry corresponding to one OpenFlow flow. Consider the P4 table definition for `InputVlan` in Figure 2. This table initializes `meta.vlan` with the packet's VLAN. At runtime, OFP4 translates each entry in this table into an OpenFlow flow. For example, an `InputVlan` entry to define port 5 as an access port for VLAN 10 might match `ingress_port = 5`, an invalid VLAN header, and wildcard `vid`, with action `SetVlan(10)`. If OFP4 maps table `InputVlan` to OpenFlow table 2, then the corresponding generated flow using Open vSwitch syntax would be

```
table=2 priority=100 in_port=5 vlan_tci=0
  actions=load(10->reg4[0..11]), resubmit(,3)
```

where the `load` action sets the metadata field allocated by the compiler for `meta.vlan` (assumed to be `reg4[0..11]`) and `resubmit` jumps to the next table in the pipeline.

Default Actions. A P4 table has a default action that executes if no entry in the table matches. OFP4 can implement the default action as an OpenFlow table entry with priority 0, ensuring that other entries have priority 1 or higher. For example, OFP4 can implement the `Drop` default action for `InputVlan` as the OpenFlow flow

```
table=2 priority=0
  actions=load(0->reg0), resubmit(,31)
```

where `reg0` is the field allocated to the egress port and OpenFlow table 31 separates the ingress and egress pipelines.

Expressions. P4 directly supports rich arithmetic expressions. OpenFlow does not have equivalent constructs, but since Open vSwitch flow tables and actions are Turing-complete, OpenFlow could implement computations, albeit inefficiently. Initially OFP4 will only support a small subset of P4 expressions. Perhaps Open vSwitch could be extended with actions performing arithmetic.

Multicast. Multicast and mirroring in PSA are implemented in buffering logic between the ingress and egress pipelines. OFP4 implements this same logic with an OpenFlow table between those used for the ingress and egress pipelines, currently table 31. For each multicast group, a flow in this table matches the group's number against `reg1[0..11]`, the metadata field allocated for the destination, with actions that clone the packet to a particular egress port and run it through the egress pipeline. For example, the flow for multicast group 12, containing ports 34 and 56, would be

```
table=31 reg1=12/0xfff actions=clone(
  load(34->reg3),resubmit(,32)),
  clone(load(56->reg3),resubmit(,32))
```

Control Flow. P4 control flow constructs translate to OpenFlow in simple ways, with `if` and `switch` becoming OpenFlow table lookups and `return` and `exit` becoming jumps to the OpenFlow table that processes the end of the current pipeline, similar to the implementation of P4 for the bmv2 backend.

Semantic differences. Changing a header in a P4 control block affects the packet only after deparsing, but many OpenFlow actions operate directly on the packet. This is most important when protocol headers must be added or removed. For example, in P4, the deparser decides whether an outgoing packet includes a VLAN header, whereas in OpenFlow an action adds or removes a VLAN header immediately. The OpenFlow "action set" of actions that are deferred until egress does not entirely bridge these differences, so P4C-OF must still compensate for them.

Digests. OFP4 could implement P4 *digest* messages from the dataplane to the control plane with OpenFlow "packet-in" messsages which, as extended by Open vSwitch, allow sending the packet with arbitrary additional data to the OpenFlow controller. OFP4 then translates the "packet-in" message into a P4Runtime digest.

3. Prototype

OFP4 is currently an experimental prototype, shown in Figure 1. We have not yet built the P4C-OF compiler, which would be a new backend for the open-source `p4c` compiler [7, 8]. However, we have manually translated a fixed P4 program into the expected output, which we use in our experiments below.

The P4C-OF compiler converts a P4 program for the `of_model.p4` architecture into a program written in the Differential Datalog language (DDlog) [15, 16]. The DDlog program implements a control-loop, waiting for P4Runtime messages and sending messages to an OvS instance.

The DDlog program is the core of the OFP4 daemon. The P4Runtime API layer accepts RPCs to populate and update entries in tables and memberships in multicast groups defined by that program. OFP4 translates insertions/deletions in P4 tables into OpenFlow flows, which it inserts/removes from OvS.

OFP4 is implemented in Rust. We used the Rust `grpcio` crate [1] to interface to gRPC-based P4Runtime. We wrote manually the templates translating P4Runtime operation into OpenFlow flows in DDlog. The salient DDlog feature that we use is that it is a language for describing *incremental* computations: it only recomputes parts of the output that are affected by each input change. To interface to Open vSwitch via OpenFlow, we wrote Rust wrappers for Open vSwitch's own libraries.

4. Evaluation

We have tested OFP4 with Nerpa [18], a programming framework for co-design of control planes and data planes. When OFP4 is used with the Nerpa controller and the P4 program for which OpenFlow templates are implemented, OFP4 correctly computes and installs the initial set of OpenFlow flows; it also reacts to P4Runtime messages generating the expected OpenFlow rules. OvS with OFP4 should perform comparably to hand-written OpenFlow. The implementation can be used with any OpenFlow switch that supports the required extensions (either a hardware or a software device).

5. Conclusion

Given the limited landscape of today's software P4 implementations, this paper explored the concept of mapping P4 into OpenFlow. We described some of the challenges in such a translation and their solutions and described a prototype. As future work, we hope to refine our prototype to a degree where it can be useful for production workloads.

References

[1] gRPC-rs. https://github.com/tikv/grpc-rs, April 2022.

[2] P4 open source programming language. https://p4.org/, April 2022.

[3] Sample applications user guide: Pipeline application. http://doc.dpdk.org/guides/sample_app_ug/pipeline.html, April 2022.

[4] Antonin Bas, Andy Fingerhut, Anirudh Sivaraman, et al. Behavioral model (BMv2). URL: https://github.com/p4lang/behavioral-model, April 2022.

[5] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. 44(3):87–95, July 2014.

[6] Mihai Budiu. Compiling P4 to eBPF. https://github.com/iovisor/bcc/tree/master/src/cc/frontends/p4, 2015.

[7] Mihai Budiu and Chris Dodd. The P4-16 programming language. *ACM SIGOPS Operating Systems Review*, 51(1):5–14, August 2017.

[8] Mihai Budiu, Chris Dodd, et al. p4c: Reference compiler for the P4 programming language. https://github.com/p4lang/p4c, April 2022.

[9] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM computer communication review*, 38(2):69–74, 2008.

[10] Tomasz Osiński, Mateusz Kossakowski, and Jan Palimąka. PSA implementation for the ebpf backend. https://github.com/p4lang/p4c/tree/master/backends/ebpf/psa/, April 2022.

[11] P4.org API Working Group. P4Runtime specification version 1.3.0. https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html, July 2021.

[12] P4.org Architecture Working Group. $P4_{16}$ portable switch architecture (PSA). https://p4.org/p4-spec/docs/PSA.html, April 2021.

[13] Justin Petitt, Jean Tourrilhes, et al. OpenFlow switch specification version 1.5.1 (protocol version 0x06). https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf, March 2015.

[14] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. The Design and Implementation of Open vSwitch. In *Network Systems Design and Implementation (NSDI)*. USENIX, 2015.

[15] Leonid Ryzhyk and Mihai Budiu. Differential datalog. In *Datalog 2.0*, Philadelphia, PA, June 4-5 2019.

[16] Leonid Ryzhyk, Mihai Budiu, Daniel Müller, Chase Wilson, et al. Differential Datalog. https://github.com/vmware/differential-datalog, April 2022.

[17] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. PISCES: A Programmable, Protocol-Independent Software Switch. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2016.

[18] Debnil Sur and Ben Pfaff. Nerpa: Network programming with relational and procedural abstractions. https://github.com/vmware/nerpa, April 2022.

[19] Péter Vörös, Dániel Horpácsi, Róbert Kitlei, Dániel Leskó, Máté Tejfel, and Sándor Laki. T4P4S: A target-independent compiler for protocol-independent packet processors. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–8, 2018.

[20] Konstantin Weitz, Stefan Heule, Waqar Mohsin, Lorenzo Vicisano, and Amin Vahdat. Leveraging P4 for fixed function switches. In *P4 Workshop*, Stanford, CA, May 1 2019.

A. OpenFlow architecture in P4

Figure 3 shows the skeleton of the P4 architecture file for describing an OpenFlow switch in P4.

```
// of_model.p4 architecture file

header Ethernet {
  bit<48> srcAddr;
  bit<48> dstAddr;
  bit<16> etherType;
}

...
// All OpenFlow headers are pre-declared

// All possible OF headers
struct Headers {
  Ethernet eth;
  IPv4     ipv4;
  ...
}

// Architecture does not have a parser

// Standard metadata
struct std_meta {
  bit<8> ingress_port;
  bit<8> egress_port;
  ...
}

// Headers are fixed.
// User-defined metadata has type M.
control Ingress<M>(inout Headers headers,
                   inout M meta,
                   inout std_meta std);
control Egress<M>(inout Headers headers,
                  inout M meta,
                  inout std_meta std);
// OpenFlow top-level package.
package OF<M>(Ingress<M> i, Egress<M> e);
```

Figure 3: P4 architecture file describing the OpenFlow pipeline.