Linux Network Programming with P4

William Tu
VMware Inc.
tuc@vmware.com

Fabian Ruffy
University of British Columbia
fruffy@cs.ubc.ca

Mihai Budiu
VMware Research
mbudiu@vmware.com

Abstract

P4 is a programming language for implementing network dataplanes. eBPF is a safe virtual machine for executing sandboxed programs in the Linux kernel. This document describes how P4 programs can be translated into eBPF programs. We also discuss the limitations of both technologies for packet processing and show some performance measurements.

1 Introduction

Software Defined Networking (SDN) [12] decouples the network control-plane from the data-plane. Open Flow [16] is a typical incarnation of SDN. Even though SDN makes the control-plane programmable, it still assumes that the data-plane is fixed. The inability to program data-planes is a significant impediment to innovation: for example, the deployment of the VxLAN protocol [22] took 4 years between the initial proposal and its commercial availability in high-speed devices.

[7] proposed the P4 language: Programming Protocol-independent Packet Processors, which is designed to make the behavior of *data-planes* expressible as software. P4 has gained rapid adoption. The p4.org consortium [1] was created to steward the language evolution; p4.org currently includes more than 100 organizations in the areas of networking, cloud systems, network chip design, and academic institutions. The P4 specification is open and public [20]. Reference implementations for compilers, simulation and debugging tools are available with a permissive license at the GitHub P4 repository [2]. While initially P4 was designed for programming network switches, its scope has been broadened to cover a large variety of packet-processing systems (e.g., network cards, FPGAs, etc.).

2 Background concepts

This section describes the background of the P4 language, architecture, and eBPF. Some of the paragraphs are adapted from [9] and [3].

2.1 The P4 programming language

P4 is a relatively simple, statically-typed programming language, with a syntax based on C, designed to express transformations of network packets.

The core abstractions provided by the P4 language are listed in Figure 1. P4 lacks many common features found in other programming languages: for example, P4 has **no** support for pointers, dynamic memory allocation, floating-point numbers, or recursive functions; looping constructs are only allowed within parsers.

P4 emphasizes static resource allocation; unlike systems such as the Linux TC subsystem, in P4 all packet processing rules and all tables must be declared when the P4 program is created.

2.2 P4 Architectures

P4 allows programs to execute on arbitrary *targets*. Targets differ in their functionality, (e.g., a switch has to forward packets, a network card has to receive or transmit packets, and a firewall has to block or allow packets), and also in their custom capabilities (e.g., ASICs may provide associative TCAM memories or custom checksum computation hardware units, while an FPGA switch may allow users to implement custom queueing disciplines). P4 embraces this diversity of targets and provides some language mechanisms to express it.

Figure 2 is an abstract view of how a P4 program interacts with the data-plane of a packet-processing engine. The data-plane is a fixed-function device that provides several programmable "holes". The user writes a P4 program to specify the behavior of each hole. The target manufacturer describes the interfaces between the programmable blocks and the surrounding fixed-function blocks. These interfaces are target-specific. Note that the fixed-function part can be software, hardware, firmware, or a mixture.

A P4 architecture file is expected to contain declarations of types, constants, and a description of the control and parser blocks that the users need to implement. Section 3.1 contains an example P4 architecture description file.

Headers describe the format (the set of fields, their ordering and sizes) of each header within a network packet.

User-defined metadata are data structures associated with each packet.

Intrinsic metadata is information provided or consumed by the target, associated with each packet (e.g., the input port where a packet has been received, or the output port where a packet has to be forwarded).

Parsers describe the permitted header sequences within received packets, how to identify those header sequences, and the headers to extract from packets. Parsers are expressed as state-machines.

Actions are code fragments that describe how packet header fields and metadata are manipulated. Actions may include parameters supplied by the control-plane at run time (actions are closures created by the control-plane and executed by the data-plane).

Tables associate user-defined keys with actions. P4 tables generalize traditional switch tables; they can be used to implement routing tables, flow lookup tables, access-control lists, and other user-defined table types, including complex decisions depending on many fields. At runtime tables behave as match-action units [8], processing data in three steps:

- Construct lookup keys from packet fields or computed metadata,
- Perform lookup in a table populated by the control-plane, using the constructed key, and retrieving an action (including the associated parameters),
- Finally, execute the obtained action, which can modify the headers or metadata.

Control blocks are imperative programs describing the data-dependent packet processing including the data-dependent sequence of table invocations.

Deparsing is the construction of outgoing packets from the computed headers.

Extern objects are library constructs that can be manipulated by a P4 program through well-defined APIs, but whose internal behavior is hardwired (e.g., checksum units) and hence not programmable using P4.

Architecture definition: a set of declarations that describes the programmable parts of a network processing device.

Figure 1: *Core abstractions of the P4 programming language.*

2.3 eBPF for network processing

eBPF is a acronym that stands for Extended Berkeley Packet Filters. In essence eBPF is a low-level programming language (similar to machine code); eBPF programs are traditionally executed by a virtual machine that resides in the Linux kernel. eBPF programs can be inserted and removed from a live kernel using dynamic code instrumentation. The main feature of eBPF programs is their static safety: prior to execution all eBPF programs have to be validated as being safe, and unsafe programs cannot be executed. A safe program provably cannot compromise the machine it is running on:

- it can only access a restricted set of memory regions (verified either statically or through inline bounds checks using software-fault isolation techniques [23]),
- it can run only for a limited amount of time; during execution it cannot block, sleep or take any locks,
- it cannot use any kernel resources with the exception of a limited set of kernel services which have been specifically whitelisted, including operations to manipulate tables (described below)

2.3.1 Kernel hooks

eBPF programs are inserted into the kernel using hooks; their execution is triggered when the flow of control reaches these hooks:

- function entry points can act as hooks; attaching an eBPF program to a function foo() will cause the eBPF program to execute every time some kernel thread executes foo().
- eBPF programs can also be attached using the Linux Traffic Control (TC) subsystem, in the network packet processing datapath. Such programs can be used as TC classifiers and actions.
- eBPF programs can also be attached to sockets or network interfaces. In this case they can be used for processing packets that flow through the socket/interface.

2.3.2 eBPF Maps

The eBPF runtime exposes a bi-directional kernel-userspace data communication channel, called maps.

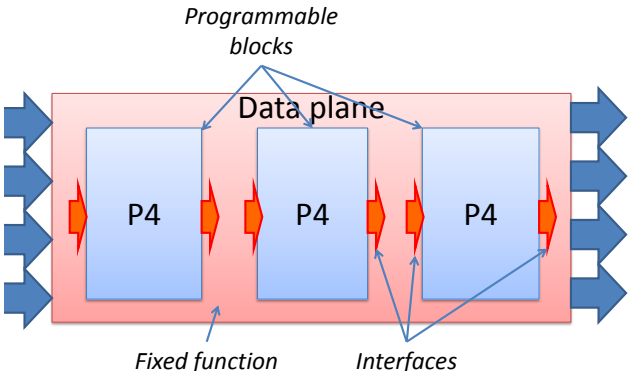

Figure 2: *Generic abstract packet-processing engine programmable in P4. The blocks labeled with P4 are programmable in P4; the surrounding block is fixed-function logic.*

eBPF maps are essentially key-value tables, where keys and values are arbitrary fixed-size bitstrings. The key width, value width and the maximum number of entries that can be stored in a map are declared at map creation time.

In user-space maps are are exposed as file descriptors. Both user- and kernel-space programs can manipulate maps by inserting, deleting, looking up, modifying, and enumerating entries.

In kernel space the keys and values are exposed as pointers to the raw underlying data stored in the map, whereas in user-space the pointers point to copies of the data.

2.3.3 Concurrency

The execution of an eBPF program is triggered by the corresponding kernel hook; multiple instances of the same kernel hook can be running simultaneously on different cores.

A map may be accessed simultaneously by multiple instances of the same eBPF program running as separate kernel threads on different cores. eBPF maps are native kernel objects, and access to the map contents is protected using the kernel RCU mechanism. This makes access to table entries safe under concurrent execution; for example, the memory associated to a value cannot be accidentally freed while an eBPF program holds a pointer to the respective value. However, accessing maps is prone to data races; since eBPF programs cannot use locks, some of these races often cannot be avoided.

2.4 XDP: eXpress Data Path

An XDP program is special case of an eBPF program for processing network packets, which attaches to the lowest levels of the networking stack. XDP was designed initially for quickly computing whether a packet should be dropped, before too many kernel resources have been allocated, to prevent denial-of-service attacks. An XDP program can inspect a network packet right after the DMA engine copies the packet from the network card.

XDP program can access eBPF maps. The kernel expects an XDP program to return the decision taken about the processed packet, one of the following values:

XDP_DROP the packet should be immediately dropped,

XDP_TX bounce the received packet back on the same port it arrived on,

XDP_PASS continue to process the packet using, the normal kernel network stack,

XDP_REDIRECT forward the packet to another port.

2.5 Comparison of P4 and eBPF

A very thorough evaluation of eBPF for writing networking programs can be found in [17]. P4 and eBPF share many features. Table 1 shows a comparison of the high-level features of both languages.

P4 was designed as a language for programming switching devices, working mostly at levels L2 and L3 of the networking stack. While P4 is being used for programming network end-points, e.g., smart NICs, some end-point functionality cannot be naturally expressed in P4 (e.g., TSO, encryption, deep packet inspection).

Many of these P4 limitations are shared with eBPF. In general, while P4 and eBPF are good for performing relatively simple packet filtering/rewriting, neither language is good enough to implement a full end-point networking stack. Table 2 compares the limitations of P4 and eBPF.

3 Compiling P4 to eBPF

In this section we describe two open-source compilers that translate P4 programs in stylized C programs, that can in turn be compiled into eBPF programs using the LLVM eBPF back-end.

3.1 Packet filters with eBPF

The eBPF back-end is part of the P4 reference compiler implementation [3]. This back-end targets a relatively simple packet filter architecture. The following is the architectural model of an eBPF packet filter expressed in P4. This architecture comprises a parser and a control block; the control block must produce a Boolean value which indicates whether the packet is accepted or not.

Feature	P4	eBPF
Targets	ASIC, software, FPGA, NIC	Linux kernel
Licensing	Apache	GPL
Tools	Compilers, simulators	LLVM back-end, verifier
Level	High	Low
Safe	Yes	Yes
Safety	Type system	Verifier
Resources	Statically allocated	Statically allocated
Policies	Match-action tables	Key-value eBPF maps
Extern helpers	Target-specific	Hook-specific
Execution model	Event-driven	Event-driven
Control-plane API	Synthesized by compiler	eBPF maps
Concurrency	No shared R/W state	Maps are thread-safe

Table 1: Feature comparison between P4 and eBPF.

Limitation	P4	eBPF
Loops	Parsers	Tail call
Nested headers	Bounded depth	Bounded depth
Multicast/broadcast	External	Helpers
Packet segmentation	No	No
Packet reassembly	No	No
Timers/timeouts/aging	External	No
Queues	No	No
Scheduling	No	No
Data structures	No	No
Payload processing	No	No
State	Registers/counters	Maps
Iterating over packet payload	No	No
Wildcard matches	Yes	No
Table/map writes	Control-plane only	Data-plane and control-plane
Iteration over table/map values	Control-plane only	Control-plane only
Synchronization (data/data, data/-control)	No	No
Resources	Statically allocated	Limited stack and buffer
Control-plane support	Complex, including remote	Simple
Safety	Safe	Verifier limited to small programs
Compiler	Target-dependent	LLVM back-end

Table 2: Limitations comparison between P4 and eBPF.

```
#include <core.p4>

extern CounterArray {
    CounterArray(bit<32> max_index, bool sparse);
    void increment(in bit<32> index);
}

extern array_table {
    array_table(bit<32> size);
}

extern hash_table {
    hash_table(bit<32> size);
}

parser parse<H>(packet_in packet, out H headers);
control filter<H>(in H headers, out bool accept);

package ebpfFilter<H>(parse<H> prs,
                      filter<H> filt);
```

The following listing shows a P4 program written for this architecture that counts the number of IPv4 packets that are encountered.

```
#include <core.p4>
#include <ebpf_model.p4>

typedef bit<48> EthernetAddress;
typedef bit<32> IPv4Address;

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

// IPv4 header without options
header IPv4 {
    bit<4>       version;
    bit<4>       ihl;
    bit<8>       diffserv;
```

```
    bit<16>      totalLen;
    bit<16>      identification;
    bit<3>       flags;
    bit<13>      fragOffset;
    bit<8>       ttl;
    bit<8>       protocol;
    bit<16>      hdrChecksum;
    IPv4Address  srcAddr;
    IPv4Address  dstAddr;
}

struct Headers {
    Ethernet eth;
    IPv4     ipv4;
}

parser prs(packet_in p, out Headers headers) {
    state start {
        p.extract(headers.eth);
        transition select(headers.eth.etherType) {
            0x800 : ip;
            default : reject;
        }
    }

    state ip {
        p.extract(headers.ipv4);
        transition accept;
    }
}

control pipe(in Headers headers, out bool pass) {
    CounterArray(32w10, true) ctr;

    apply {
        if (headers.ipv4.isValid()) {
            ctr.increment(headers.ipv4.dstAddr);
            pass = true;
        } else
            pass = false;
    }
}

ebpfFilter(prs(), pipe()) main;
```

Compilation to C is fairly straightforward; the generated C program is always memory-safe, using bounds-checks for all packet accesses. For the entire P4 program a single C function is generated which returns a Boolean value. Table 3 shows how each P4 construct is converted to a C construct. Currently programs with parser loops are rejected, but a parser loop unrolling pass (under development) will allow such programs to be compiled.

3.2 Packet forwarding with XDP

A second P4 to C compiler is available as an open-source project: [4]. This compiler extends the eBPF compiler from Section 3.1. This compiler can target either a packet filter, or a packet switch. The following listing shows the XDP architectural model targeted by this compiler. You can see that a P4 XDP program can (1) return to the kernel one of the 4 outcomes described in Section 2.4, and (2) it can also modify the packet itself, by inserting, modifying or deleting headers.

P4 construct	C Translation
`header`	`struct` with an additional `valid` bit
`struct`	`struct`
parser state	block statement
state transition	`goto` statement
`extract` call	load/shift/mask data from packet
table	2 eBPF maps — one for actions, one for the default action
table key type	`struct` type
`action`	tagged `union` with action parameters
`action` parameters	`struct`
`action` body	block statement
table `apply`	lookup in eBFP map + `switch` statement with all actions
counters	eBPF map

Table 3: Compiling P4 constructs to C.

```
#include <ebpf_model.p4>
enum xdp_action {
  XDP_ABORTED,
  XDP_DROP,
  XDP_PASS,
  XDP_TX
}
struct xdp_input { bit<32> input_port }

struct xdp_output {
  xdp_action output_action;
  bit<32> output_port;
}

parser xdp_parse<H>(packet_in packet,
                    out H headers);
control xdp_switch<H>(inout H hdrs,
                      in xdp_input i,
                      out xdp_output o);
control xdp_deparse<H>(in H headers,
                       packet_out packet);

package xdp<H>(xdp_parse<H> p,
               xdp_switch<H> s,
               xdp_deparse<H> d);
```

4 Testing eBPF programs

We have built an elaborated testing infrastructure to test the P4 to C compilers. The infrastructure can perform both functional correctness testing at the user level, and complete end-to-end testing by running in the kernel.

4.1 User-Space Testing

User-space testing validates the correctness of the code generated by compiler and can be performed even on systems that lack eBPF support in the kernel. The user space testing framework does not depend

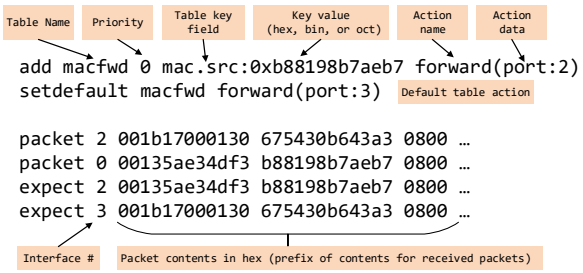

Figure 3: Annotated example of a basic STF file.

on the LLVM [15] or any particular kernel version. It also does not require usage of `iproute2` [14] tooling such as `tc` or `ip`. It is also easier to debug failing tests in user-space, by using tools such as GDB [21], Valgrind [19], or Wireshark [10].

Command	Description
packet port data	Insert a frame of bytes *data* into port *port*.
expect port data	Expect a frame of bytes *data* on port *port*.
add tbl priority match action	Insert a match-action entry with key *match* and action *action* into table *tbl*.
setdefault tbl action	Set the default action for table *tbl*.
check_counter tbl key==n	Check if the value on the entry *key* in counter table *tbl* matches *n*.
wait	Pause for a second.

Table 4: The STF command palette.

4.2 The Simple Test Framework

The P4 compiler includes a simple language (STF = Simple Testing Framework) to describe input/output packets and to populate P4 tables. Initially STF was used with software simulators to validate P4 programs, but we have adapted it for testing the eBPF and XDP back-ends. The STF framework is written in Python. Figure 3 shows a small program written in the STF language. Table 4 describes the list of currently supported STF operations in the eBPF testing backend.

The STF `packet` statement describes an input port and the contents of a inbound packet. The `expect` statement describes an output port and the contents of an outbound packet.

Tables can be populated using the `add` statement, which indicates a P4 table and an action to insert in the table, including values for the action parameters. Currently we assume that all `add` statements are executed prior to all the packet manipulation statements.

Our testing framework converts `packet` statements into Pcap files, one for each input port tested. `add` statements are converted into C programs that populate eBPF maps.

Although STF supports testing counters as well, our eBPF testing framework does not yet support this feature.

4.3 The Test Runtime

Executing a P4 eBPF test is done in five stages (Figure 4):

1. **compile-p4:** Compile the P4 file to C program. This validates the P4 compiler.
2. **parse-stf:** Convert the STF file to a C program and into input PCAP files.
3. **compile-data-plane:** Compile and load the C programs into an executable.
4. **run:** Wire up the executable to read from the input PCAP files; run the executable – first populate tables then execute the program over the input packets. Capture the produced output packets into output files.
5. **check-results:** Compare the output packets with the expected results.

These five stages look slightly differently when testing in user-space and in kernel-space.

In user space we use a hash-table library to emulate eBPF maps.

When testing in kernel-space we compile the eBPF/XDP programs to eBPF object files using LLVM. Before the eBPF/XDP program is loaded, the framework creates a bridge running in a network namespace. Namespace-based isolation allows us to run multiple tests in parallel. Virtual interfaces are attached to the bridge. The testing runtim injects packets into the associated ports using raw sockets. The output results are recorded by attaching Tcpdump [11] to each output virtual interface.

5 Experimental results

5.1 Testbed

All of our performance results use a hardware test bed that consists of two Intel Xeon E5 2440 v2 1.9GHz servers, each with 1 CPU socket and 8 physical cores with hyperthreading enabled. Each target server has an Intel 10GbE X540-AT2 dual port NIC, with the two ports of the Intel NIC on one server connected to the two ports on the identical NIC on the other server. We installed p4c-xdp on one server, the *target server*, and attached the XDP program to the port that receives the packets The other server, the *source server*, generates packets at the maximum 10 Gbps packet rate of 14.88 Mpps using the DPDK-based TRex [13] traffic generator. The source server sends minimum

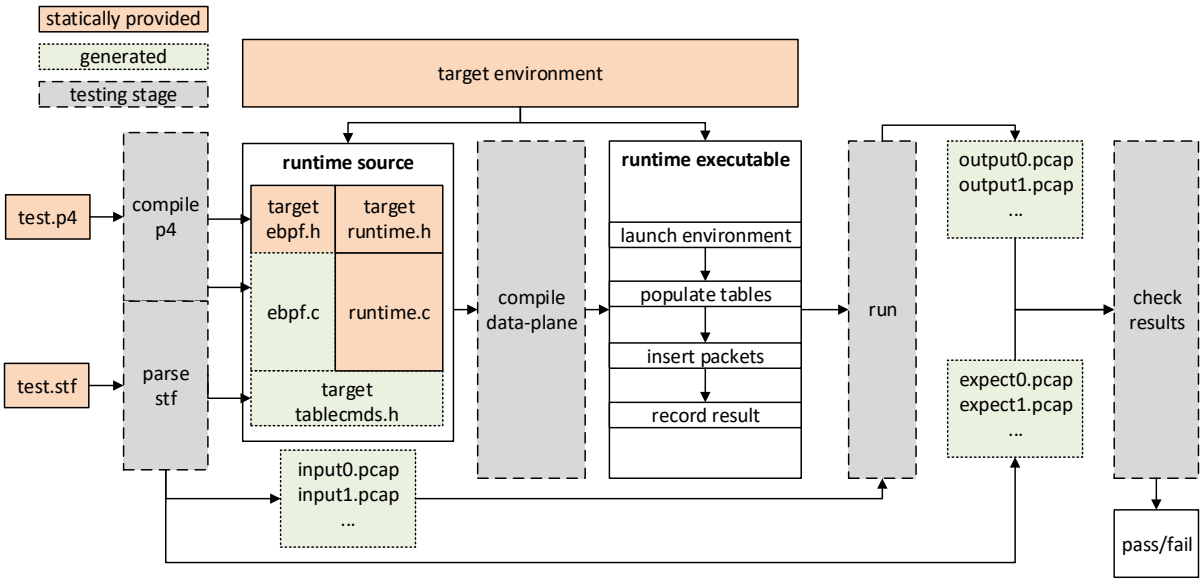

Figure 4: Testing workflow for a P4-eBPF program. Environment and target are provided by the user.

length 64-byte packets in a *single* UDP flow to one port of the target server, and receives the forwarded packets on the same port. At the target server, we use only one core to process all packets. Every packet received goes through the pipeline specified in P4.

We use the sample p4 programs in the tests directory and the following metrics to understand the performance impact of the P4-generated XDP program:

- Packet Processing Rate (Mpps): Once the XDP program finishes processing the packet, it returns one of the actions mentioned in section 2. When we want to count the number of packets that can be dropped per second, we modify each p4 program to always return XDP_DROP.
- CPU Utilization: Every packet processed by the XDP program is run under the per-core software IRQ daemon, named `ksoftirqd/`*`core`*. All packets are processed by only one core with one kthread, the ksoftirqd, and we measure the CPU utilization of the ksoftirqd on the core.
- Number of BPF instructions verified: For each program, we list the complexity as the number of BPF instructions the eBPF verifier scans.

The target server is running Linux kernel 4.19-rc5 and for all our tests, the BPF JIT (Just-In-Time) compiler is enabled and JIT hardening is disabled. All programs are compiled with clang 3.8 with llvm 5.0. For each test program, we use the following command from iproute2 to load it into kernel:

```
ip link set dev eth0 xdp obj xdp1.o verb
```

The Intel 10GbE X540 NIC is running the `ixgbe` driver with 16 RX queues set-up. Since the source server is sending single UDP flow, packets always arrive at a single ring ID. As a result, we collect the number of packets being dropped at this ring.

5.2 Results

To compute the baseline performance we wrote two small XDP programs by hand: `SimpleDrop`, drops all packets by returning `XDP_DROP` immediately. `SimpleTX` forwards the packet to the receiving port returning `XDP_TX`. Each of these programs consists of only two BPF instructions.

```
/* SimpleDrop */
0: (b7) r0 = 1 // XDP_DROP
1: (95) exit

/* SimpleTX */
0: (b7) r0 = 3 // XDP_TX
1: (95) exit
```

After, we attached the following P4 programs to the receiving device:

- xdp1.p4: Parse Ethernet/IPv4 header, deparse it, and drop.
- xdp3.p4: Parse Ethernet/IPv4 header, lookup a MAC address in a map, deparse it, and drop.
- xdp6.p4: Parse Ethernet/IPv4 header, lookup and get a new TTL value from eBPF map, set to IPv4 header, deparse it, and drop.
- xdp7.p4: Parse Ethernet/IPv4/UDP header, write a pre-defined source port and source IP, recalculate checksum, deparse, and drop.

P4 program	CPU Util.	Mpps	Insns./Stack
SimpleDrop	75%	14.4	2/0
SimpleTX	100%	7.2	2/0
xdp1.p4	100%	8.1	277/256
xdp3.p4	100%	7.1	326/256
xdp6.p4	100%	2.5	335/272
xdp7.p4	100%	5.7	5821/336
xdp11.p4	100%	4.7	335/216
xdp15.p4	100%	5.5	96/56

Table 5: Performance of XDP program generated by p4c-xdp compiler using single core.

- xdp11.p4: Parse Ethernet/IPv4 header, swap src/dst MAC address, deparse it, and send back to the same port (XDP_TX).
- xdp15.p4: Parse Ethernet header, insert a customized 8-byte header, deparse it, and send back to the same port (XDP_TX).

As shown in Table 5, xdp1.p4 allows us to measure the overhead introduced by parsing and deparsing: a drop from 14.4 Mpps to 8.1 Mpps. xdp3.p4 reduces the rate by another million PPS due to the eBPF map lookup (this operation always return NULL, no value from the map is accessed). xdp6.p4 has significant overhead because it accesses a map, finds a new TTL value, and writes to the IPv4 header. Interestingly, although xdp7.p4 does extra parsing to the UDP header and checksum recalculation, it has only a moderate overhead because of the lack of map accesses.

Finally, xdp11.p4 and xdp15.p4 show the transmit (XDP_TX) performance. Compared with xdp11, xdp15.p4 invokes the bpf_adjust_head helper function to reset the pointer for extra bytes. It does not incur much overhead because there is already a reserved space in front of every XDP packet frame.

5.3 Performance Analysis

To further understand the performance overhead of programs generated by p4c-xdp, we started broke down the CPU utilization. We used the Linux perf tool on the process ID of the ksoftirqd that shows 100%:

```
perf record -p <pid of ksoftirqd> sleep 10
```

The following output shows the profile of xdp1.p4:

```
83.19% [kernel.kallsyms] [k] ___bpf_prog_run
8.14%  [ixgbe]           [k] ixgbe_clean_rx_irq
4.82%  [kernel.kallsyms] [k] nmi
1.48%  [kernel.kallsyms] [k] bpf_xdp_adjust_head
1.07%  [kernel.kallsyms] [k] __rcu_read_unlock
0.40%  [ixgbe]           [k] ixgbe_alloc_rx_buffers
```

This confirms that most of the CPU cycles are spent on executing the XDP program, `___bpf_prog_run`, which caused us to investigate the eBPF C code of xdp1.p4.

P4 program	CPU Util.	Mpps	Insns./Stack
xdp1.p4	77%	14.8	26/0
xdp3.p4	100%	13	100/16
xdp6.p4	100%	12	98/40

Table 6: Performance of XDP program without deparser.

After commenting out the deparser C code, performance increases significantly (see Table 6). In the generated code, the p4c-xdp compiler always writes back the entire packet content, even when the P4 program does not modify any fields. In addition, the parser/deparser incur byte-order translation, e.g., htonl, ntohl. This could be avoided by always using network byte-order in P4 and XDP. We plan to implement optimizations to reduce this overhead.

6 Challenges and Future Work

In general, our development experience is mirrored by the lessons described in [17] and [5].

No multi-/broadcast support While XDP is able to redirect single frames it does not have the ability to clone and redirect packets similar to `bpf_clone_redirect`. This makes development of more sophisticated P4 forwarding programs problematic.

The stack size is too small More complex XDP programs are rejected by the verifier despite their safeness. This is a particular challenge when attempting to implement network function chaining or more advanced pipelined packet processing in a single XDP program.

Generic XDP and TCP Our testing framework uses virtual Linux interfaces and generic XDP [18] to verify XDP programs. Unfortunately, we are unable to test TCP streams as the protocol is not supported by this driver [6]. Any program loaded by generic XDP operates after the creation of the `skb` and requires the original packet data. Since TCP clones every packet and passes the unmodifiable `skb` clone, generic XDP is bypassed and never receives the datagram.

Using libbpf userspace library Creating of compilation of eBPF programs in userspace requires substantial effort. Many function calls and variables available in sample programs are not available as C library and currently the p4c-xdp project copied these utilities from kernel code or assembled from various online sources. We plan to switch to use libbpf to control and manage the eBPF programs and maps.

Persistent eBPF maps in namespaces [need to verify] When using eBPF programs in namespaces, maps exported via tc do not persist across `ip netns exec` calls. The consequence is that any program has to be run in a single shell command, otherwise the eBPF map becomes unusable despite the continued existence of the namespace.

References

[1] P4 Consortium. `http://p4.org`.

[2] P4 github repository. `https://github.com/p4lang`. Retrieved May 2017.

[3] A p4 to ebpf compiler back-end. `https://github.com/p4lang/p4c/blob/master/backends/ebpf/README.md`. Retrieved September 2019.

[4] A p4 to xdp compiler back-end. https://github.com/vmware/p4c-xdp. Retrieved September 2019.

[5] Bertin, G. "xdp in practice: Integrating xdp into our ddos mitigation pipeline. In *Technical Conference on Linux Networking (Netdev 2.1)* (Montreal, Canada, Apr 2017).

[6] Bertrone, M. Generic XDP and veth. `https://www.spinics.net/lists/xdp-newbies/msg00440.html`, Jan 2018.

[7] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. Programming protocol-independent packet processors. In *ACM SIGCOMM Computer Communications Review (CCR)* (July 2014), vol. 44.

[8] Bosshart, P., Gibb, G., Kim, H.-S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., and Horowitz, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *SIGCOMM* (2013), ACM, pp. 99–110.

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

[10] Combs, G. Wireshark: Go deep. `https://www.wireshark.org/`.

[11] Group, T. Tcpdump - dump traffic on a network. `http://www.tcpdump.org/`.

[12] Haleplidis, E., Pentikousis, K., Denazis, S., Salim, J. H., Meyer, D., and Koufopavlou, O. Software-defined networking (SDN): Layers and architecture terminology. `https://tools.ietf.org/html/rfc7426`, January 2015. RFC 7426.

[13] Inc, C. Trex: Realistic traffic generator. `https://trex-tgn.cisco.com/`.

[14] Kuznetsov, A. Iproute2: Linux network management. `https://wiki.linuxfoundation.org/networking/iproute2`.

[15] Lattner, C., and Adve, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization* (Washington, DC, USA, 2004), CGO '04, IEEE Computer Society, pp. 75–.

[16] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev. 38*, 2 (Mar. 2008), 69–74.

[17] Miano, S., Bertrone, M., Risso, F., Tumolo, M., and Bernal, M. V. Creating complex network services with ebpf: Experience and lessons learned. In *IEEE High Performance Switching and Routing (HPSR18)* (Bucharest, Romania, June 2018).

[18] Miller, D. Generic XDP. `https://lwn.net/Articles/720072/`, April 2017.

[19] Nethercote, N., and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2007), PLDI '07, ACM, pp. 89–100.

[20] P4.org. P4-16 language specification. `https://github.com/p4lang/p4-spec/tree/master/p4-16/spec`, May 2017.

[21] Project, G. Gdb: The gnu project debugger. `https://www.gnu.org/software/gdb/`.

[22] Storvisor, M. M., Dutt, D., Duda, K., Agarwal, P., Kreeger, L., Sridhar, T.,

Bursell, M., and Wright, C. Virtual eXtensible Local Area Network (VXLAN): A framework for overlaying virtualized layer 2 networks over layer 3 networks. `https://tools.ietf.org/html/rfc7348`, August 2014.

[23] Wahbe, R., Lucco, S., Anderson, T., and Graham, S. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review 27*, 5 (December 1993), 203–216.