Abstract

P4 is a domain-specific language for implementing network data-planes. The P4 abstraction allows programmers to write network protocols in a generalized fashion, without needing to know the configuration specifics of the targeted data-plane.

The extended Berkeley Packet Filter (eBPF) is a safe virtual machine for executing sand-boxed programs in the Linux kernel. eBPF, and its extension the eXpress Data Path (XDP), effectively serve as programmable data-planes of the kernel.

P4C-XDP is a project combining the performance of XDP with the generality and usability of P4. In this document, we describe how P4 can be translated into eBPF/XDP. We review the fundamental limitations of both technologies, analyze the performance of several generated XDP programs, and discuss problems we have faced while working on this new technology.

1 Introduction

The introduction of Software Defined Networking (SDN) [12] has decoupled the network control-plane from the data-plane. The Open Flow Protocol [17] 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 [23] took 4 years between the initial proposal and its commercial availability in high-speed devices.

To address this state of affairs, [7] introduced 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 [21]. 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 briefly describes the P4 programming language and eBPF. Some of the text is 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.

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
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.

Figure 2: Generic abstract packet-processing engine programmable in P4.

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. Sections 3.1 and 3.2 contain examples P4 architecture description files.

2.3 eBPF for network processing

eBPF is an 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 re-
regions (verified either statically or through inline bounds checks using software-fault isolation techniques [24]),

- 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

Epid 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-to-user-space data communication channel, called maps. 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 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 a specific class of eBPF packet processing program, which attaches to the lowest levels of the networking stack [13]. XDP was initially designed to prevent denial-of-service attacks by quickly deciding whether a packet should be dropped, before too many kernel resources have been allocated. An XDP program can inspect a network packet right after the DMA engine copies the packet from the network card and can also access eBPF maps.

The kernel expects an XDP program to return the decision taken about the processed packet. This may be 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 [18]. P4 and eBPF share many features. Table 2 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 1 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 into stylized C, which in turn can be compiled into eBPF programs using the LLVM eBPF back-end.
<table>
<thead>
<tr>
<th>Limitation</th>
<th>P4</th>
<th>eBPF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loops</td>
<td>Parsers</td>
<td>Tail call</td>
</tr>
<tr>
<td>Nested headers</td>
<td>Bounded depth</td>
<td>Bounded depth</td>
</tr>
<tr>
<td>Multicast/broadcast</td>
<td>External</td>
<td>Helpers</td>
</tr>
<tr>
<td>Packet segmentation</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Packet reassembly</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Timers/timeouts/aging</td>
<td>External</td>
<td>No</td>
</tr>
<tr>
<td>Queues</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Scheduling</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Data structures</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Payload processing</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>State</td>
<td>Registers/counters</td>
<td>Maps</td>
</tr>
<tr>
<td>Iterating over packet payload</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Wildcard matches</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Table/map writes</td>
<td>Control-plane only</td>
<td>Data-plane and control-plane</td>
</tr>
<tr>
<td>Iteration over table/map values</td>
<td>Control-plane only</td>
<td>Control-plane only</td>
</tr>
<tr>
<td>Synchronization (data/data, data/control)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Resources</td>
<td>Statically allocated</td>
<td>Limited stack and buffer</td>
</tr>
<tr>
<td>Control-plane support</td>
<td>Complex, including remote</td>
<td>Simple</td>
</tr>
<tr>
<td>Safety</td>
<td>Safe</td>
<td>Verifier limited to small programs</td>
</tr>
<tr>
<td>Compiler</td>
<td>Target-dependent</td>
<td>LLVM back-end</td>
</tr>
</tbody>
</table>

Table 1: Comparison of the limitations of P4 and eBPF.

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

Table 2: Feature comparison between P4 and eBPF.

```c
#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);
```

Figure 3: Packet filter P4 architectural model for an eBPF target.

### 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. Figure 3 shows 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.

Figure 4 shows a P4 program written for this architecture. It counts the number of IPv4 packets that are encountered.

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.
```c
#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<16> 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;
  }
}

// Instantiate main package
ebpffilter(prs(), pipe()) main;
```

Figure 4: A P4 program that counts the number of IPv4 packets encountered.

```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);

Figure 5: XDP packet switching architectural model.

### 3.2 Packet forwarding with XDP

A second P4 to C compiler, P4C-XDP, is available as an open-source project hosted at [4]. P4C-XDP is licensed under the GNU GPL and Apache License. This compiler extends the eBPF compiler from Section 3.1. It 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 four outcomes described in Section 2.4, and (2) it can also modify the packet itself, by inserting, modifying or deleting headers.
4 Testing eBPF programs

To test the P4 to C compilers we have adapted and extended the existing P4 testing infrastructure. The infrastructure can perform both functional correctness testing at the user-level, and complete end-to-end testing running programs in the kernel.

4.1 User-Space Testing

User-space testing validates the correctness of the code generated by compiler. It can even be performed on systems that lack eBPF support in the kernel. The user-space testing framework does not depend on the LLVM [16] or any particular kernel version. It also does not require usage of iproute2 [15] tooling such as tc or ip. It is also easier to debug failing tests in user-space, by using tools such as GDB [22], Valgrind [20], or Wireshark [10].

4.2 The Simple Test Framework

```plaintext
<table>
<thead>
<tr>
<th>Table Name</th>
<th>Priority</th>
<th>Table Key</th>
<th>Key value</th>
<th>Action name</th>
<th>Action data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>setdefault macfwd forward(port:2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>packet 2 001b17000130 67543bb643a3 0800 ... forward(port:2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>expect 2 00135e344df3 b881987ae7b 0800 ... forward(port:3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>packet 0 00135e344df3 b881987ae7b 0800 ... forward(port:3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6: Annotated example of an STF program for testing the P4 compiler.
```

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 6 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 (Packet CAPture) 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 7):
1. compile-p4: Compile the P4 file to C program.
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 runtime 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 testbed 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 10GhE 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 T Rex [14] traffic generator. The source server sends minimum length 64-byte packets in a single UDP flow to one port of the target server, and re-
Table 4: The STF command palette.

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

Figure 7: Testing workflow for a P4-eBPF program. Environment and target are provided by the user.

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:

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

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

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

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

<table>
<thead>
<tr>
<th>P4 program</th>
<th>CPU Util.</th>
<th>Mpps</th>
<th>Insns./Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>xdp1.p4</td>
<td>77%</td>
<td>14.8</td>
<td>26/0</td>
</tr>
<tr>
<td>xdp3.p4</td>
<td>100%</td>
<td>13</td>
<td>100/16</td>
</tr>
<tr>
<td>xdp6.p4</td>
<td>100%</td>
<td>12</td>
<td>98/40</td>
</tr>
</tbody>
</table>

Table 6: Performance of XDP program without deparser.

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 to break 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.

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 Lessons learned

In general, our development experience is mirrored by the lessons described in [18] and [5].

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

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

**Generic XDP and TCP**  Our testing framework uses virtual Linux interfaces and generic XDP [19] 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 data-gram.

**Using the libbpf user-space library**  Compilation of eBPF programs in user-space requires substantial effort. Many function calls and variables available in sample programs are not available as general C library. Any user trying to interact with the generated C code has to provide their own sources. Currently, P4C-XDP maintains copies of utilities from kernel code or various online sources. This is not a sustainable approach. We plan to integrate libbpf with our repository to control and manage the eBPF programs and maps.

**Pinned eBPF maps in network namespaces**  When using eBPF programs in namespaces, maps that were pinned via tc do not persist across ip netns exec calls. As consequence, any program has to be run in a single continuous shell command. Example:

```
bash -c "tc filter add ...; ls /sys/fs/bpf/tc/globals"
```

Once ip netns exec has finished, the reference to the eBPF map and all its associated state disappear. The eBPF program, however, remains attached to the virtual interface, leading to inconsistent packet processing behavior.

**References**


[16] Lattner, C., and Adve, V. Llvm: A compilation framework for lifelong program analysis


