Introducing P4TC - A P4 implementation on Linux Kernel using Traffic Control

Jamal Hadi Salim
Mojatatu Networks
Ottawa, Canada
jhs@mojatatu.com

Deb Chatterjee
Intel Corporation
Santa Clara, US
deb.chatterjee@intel.com

Victor Nogueira
Pedro Tammela
Mojatatu Networks
Ottawa, Canada
victor@mojatatu.com
pctammela@mojatatu.com

Tomasz Osiński*
Intel Corporation
Warsaw University of Technology
Warsaw, Poland
tomasz.osinski@intel.com

Evangelos Haleplidis
Mojatatu Networks
University of Piraeus
Ottawa, Canada
ehalep@mojatatu.com

Balachandher Sambasivam
Usha Gupta
Komal Jain
Sosutha Sethuramapandian
Intel Corporation
Bengaluru, India
balachandher.sambasivam@intel.com
usha.gupta@intel.com
komal.jain@intel.com
sosutha.sethuramapandian@intel.com

ABSTRACT

The networking industry is at an inflection point with ever increasing network link capacities coupled with the presence of programmable hardware ASICs. These set of circumstances call out for a robust approach to hardware and software co-existence for network programmability.

P4TC is a P4 Linux kernel-native implementation on top of the Linux Traffic Control (TC) infrastructure that provides a vendor-neutral, kernel-independent and architecture-independent interface for Match-Action packet processing compatible with the P4 specification. P4TC facilitates both a hardware datapath and a functionally equivalent kernel eBPF-assisted software datapath making it ideal to deal with both high speed links and programmable hardware.

In this paper, we describe the goals and motivation of P4TC, the design and architecture as well as illustrate the different concepts of the P4TC infrastructure via an example of a simple L2 switch.

CCS CONCEPTS

• **Networks** → **Programmable networks**.

*Supported by the Foundation for Polish Science (FNP)

EuroP4 '23, December 8, 2023, Paris, France

ACM ISBN 979-8-4007-0446-8/23/12...$15.00
https://doi.org/10.1145/3630047.3630193

KEYWORDS

Programmable data plane; P4; TC; eBPF; Linux

ACM Reference Format:
Jamal Hadi Salim, Deb Chatterjee, Victor Nogueira, Pedro Tammela, Tomasz Osiński, Evangelos Haleplidis, Balachandher Sambasivam, Usha Gupta, Komal Jain, and Sosutha Sethuramapandian. 2023. Introducing P4TC - A P4 implementation on Linux Kernel using Traffic Control. In *Proceedings of the 6th European P4 Workshop (EuroP4 '23), December 8, 2023, Paris, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3630047.3630193

1 INTRODUCTION

A recent talk at the P4 workshop [14] outlined that, on average, network link capacity goes up 10x, approximately every 8 years, whereas CPU speeds, on average, go up linearly and double about every 1.5 years. The resulting lifecycle is a step function for the link capacity that increases periodically whereas the CPU performance is a linear function.

In essence there are short periods in the described lifecycle where the datapath programmability in software can handle existing port speeds when the linear CPU performance is greater or equal to the link capacity step function graph. However, if you factor in that a host or middlebox may have two or more ports, then the postulation from [14] implies that it is impossible for software datapaths to keep up with port speeds.

To address the software/hardware capacity mismatch, the Linux kernel's TC Match-Action architecture [9], which has existed since kernel version 2.2 (circa 2000), around 2015 began offering a hybrid software-hardware datapath architecture [16] [17] using both the TC u32 [23], TC flower [7] and TC matchall classifiers [13]. Presently, most of the current Linux offload deployments are centered around the TC flower classifier [7].

The TC flower approach is equivalent to a fixed datapath[1]. As a result, there is *kernel-dependence* on making any changes to flower. Any extensions to TC flower, while possible, occur via a long arduous upstream process that requires modification of the Linux kernel code base, TC kernel core and driver code, and its associated netlink control interfaces and utilities.

The arduous upstream process could result in several years of effort to add support for a simple network protocol, or even header, and much longer during the step-up process discussed in [14]. Not only is the process time-intensive, but it also requires above-average technical and social skills to deal with the community.

1.1 P4 and P4TC to the Rescue

Network programmability, control and datapath separation, and software and hardware disaggregation have seen a lot of progress in the last two decades [6]. P4 [2] opens new doors that allow datapaths to be prescribed in the P4 language and then, using a compiler, cast into a hardware pipeline. With the advent of P4 programmable NICs [10] [1], it is possible to introduce new datapath processing in hardware - overcoming the performance lag discussed in the previous section.

P4TC[2] is a P4 kernel-native implementation on top of the Linux Traffic Control (TC) which aims at introducing a vendor-neutral, kernel-independent and P4-architecture-independent mechanism used to facilitate Match-Action processing within the Linux kernel.

By using the TC infrastructure, we can take advantage of the existing kernel ecosystem of vendors and operator communities already skilled in developing, deploying, and debugging TC infrastructure both in software and hardware datapaths. The P4TC infrastructure embraces P4 and removes the limitation of the TC flower fixed datapaths and associated kernel-dependency by enabling dynamic datapaths and associated controls defined by custom user-defined P4 programs.

A P4 program is compiled with the P4C[19] compiler's tc backend to generate all the necessary artifacts for both data and control plane to manifest the P4 program's abstraction into the kernel. Once loaded into the kernel, the program is instantiated and activated to execute the intended packet processing as subjected by the control plane.

This paper makes the following contributions:

- Introduces the goals and motivations of P4TC.
- Documents the design decisions and architecture of the P4TC infrastructure.
- Illustrates the P4TC infrastructure with a simple example of an L2 switch.

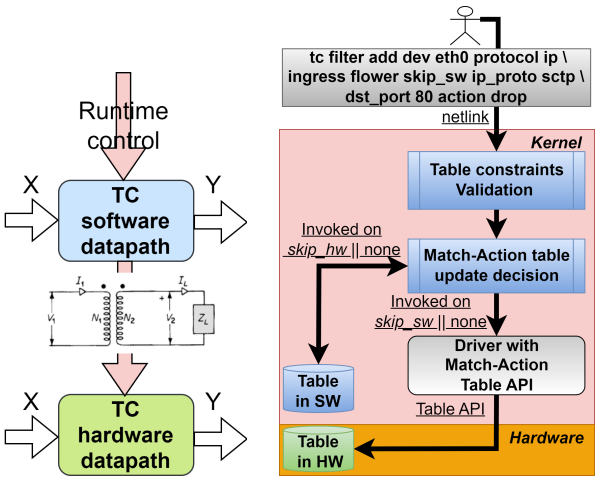
2 BACKGROUND

2.1 Traffic Control

Figure 1 shows the TC Match-Action functionality. Any hardware offload in the TC is mandated by the upstream process to provide a functionally equivalent software implementation as illustrated in Figure 1a.

[1]At the time of writing this paper, flower supported about 20 headers it can classify on

[2]At the time of writing this paper, the patchset with the P4TC implementation is under review by the Linux community.



(a) TC Equivalence (b) Flower Runtime

Figure 1: TC Match Action Functionality

In other words, for a given TC packet processing function, an input packet X should result in the same transformation with an output result of Y whether that packet is exercised by the kernel software function or hardware offloaded datapath. The hardware and software implementations do not have to be algorithmically equivalent as long as the same input produces the same expected output in both. Any impedance mismatch is handled in the north-south direction by the hardware vendor driver. This arrangement means that one could experiment with a given packet service in containers, VMs, etc., and when needing higher performance offload the packet service (on appropriate hardware) using a control plane policy toggle (with zero changes to the implementation).

Figure 1b illustrates the TC runtime control, using flower as an example. A match or action entry added by the control plane could target the software datapath (**skip_hw**) or the hardware datapath (**skip_sw**) or even specify that it should be installed to hardware if there are enough hardware resources available and, upon failure, add it to the software. Using these knobs, an operator could craft policies such that an ingress processing pipeline gets split with the initial processing done in hardware and subsequent in software, the reverse, or entirely in one or other environment.

2.2 eBPF and Kernel Functions

eBPF [5] provides an in-kernel virtual machine that can safely run eBPF programs within the kernel context. eBPF has many hooks in the Linux kernel where programs can be attached. For the sake of this paper we focus on the TC and XDP hooks.

eBPF programs, when unable to express functionality using the eBPF instruction set can use functionality within kernel proper using a concept of helpers (eBPF helpers are equivalent to P4 externs). A more recent access provided by the kernel to eBPF that is similar to helpers is BPF Kernel Functions (also known as kfuncs[11]). Kfuncs are functions in the Linux kernel that are exposed for use by eBPF programs.

3 INTRODUCING P4TC

3.1 Goals, Motivation and Design Decisions

The primary goal of P4TC is to grow the network programmability ecosystem. We picked P4 because it is currently considered the most open and deployed datapath language. P4 improves over the fixed datapath semantics, traditionally provided by datapath device vendors, by allowing custom specification of a datapath, which can then be cast into hardware. This aspect of P4 not only paves the path for innovation of new ideas but also fits well with the fact that almost all organizations conform to Conway's law [3] and model their datapaths based on their needs[3]. If you couple what P4 provides with high-speed xPUs shipped by multiple P4 vendors, you get a powerful concoction of necessary ingredients for advanced high-speed network programmability.

Large consumers of NICs currently provide their specifications using P4 [4] to vendors. P4TC completes that larger picture by enabling seamless co-existence of P4 hardware and software datapaths.

Linux is the most widely deployed networking infrastructure OS and, more importantly, the most accessible to academia, hobbyists, and the industry in general - and from that perspective, it makes P4TC ideal for building a ubiquitous network programmability ecosystem. Linux TC brings another necessary ingredient to our design: years of development and deployment experience on the concept of a hybrid software/hardware datapath co-existence coupled with a singular (netlink-based) control API.

Given the above choices, P4TC will provide a great opportunity to overcome existing constraints to wide deployment and commodification of network programmability. Often, a difficult constraint to overcome in network programmability in environments like Linux is the requirement for hard-to-acquire developer expertise. In the case of P4TC, this would imply requiring one to be knowledgeable in both eBPF and kernel code internals, and even when knowledgeable, productivity tends to be low, given the complexity of the involved infrastructure. P4TC addresses this by moving as much knowledge as possible into the compiler. The ultimate goal is for an author to write a P4 program, click a button and get all the necessary artifacts without being skilled in either the kernel or eBPF. This approach is intended to reduce or totally remove, whenever possible, the need for developers and instead shift the effort into subject matter expertise, as prescribed in P4. We hope to continue to embed necessary knowledge into compilers to automate a lot of the required coding.

3.2 P4TC Work Flow

Figure 2 illustrates the P4TC workflow, starting with authoring a P4 program, generating the necessary artifacts, installing, and finally running the program. First, the author writes the P4 program and then feeds it to P4C alongside an optional target constraint definition[4]. The developer chooses whether to target the P4TC software datapath, the hardware datapath or both.

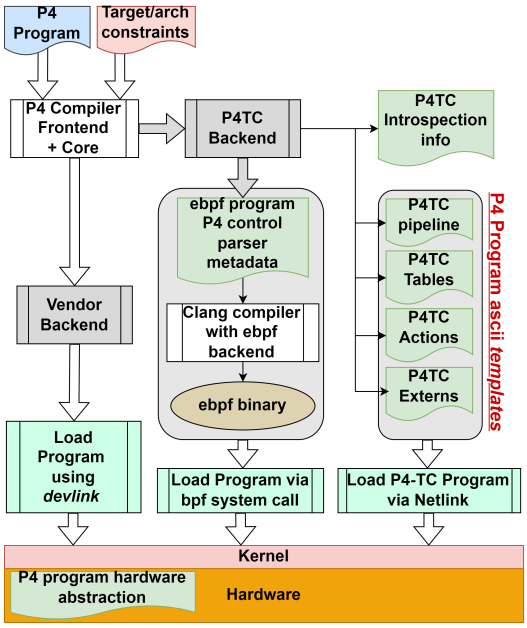


Figure 2: Creation And Installation

For the software datapath, the compiler generates firstly one or two ASCII files containing the eBPF parser and control code; compile options select whether the parser should be a separate eBPF program. Secondly, an ASCII file containing the template prescription/policy used to teach the kernel about the different P4 objects reflected in the P4 program; these include tables, actions and externs. Thirdly, an ASCII JSON file which describes all the necessary attributes of runtime objects.

To provide context, let's assume the following P4 program snippet represents a simple L2 switch whose FDB table is looked up using a destination MAC address as the key and is programmed from the control plane. The datapath forwards the packet if there's a hit on the table. Otherwise, the packet is dropped.

```
table FDB {
  key = {
    hdr.eth.dstAddr:
     exact @tc_type("macaddr") @name("dstAddr");
 }
  actions = {
   send_nh;
   drop;
  }
  default_action = drop;
}
```

The user would compile the p4 program as follows:

[3]Such datapaths are typically driven by how the organization structures its applications deployments. A good sample space is to compare the different cloud vendors approach to datapath implementation - for example both MS DASH and Google's new datapath; while both are based on P4, but they each implement different datapath pipelines.

[4]The constraint describes resource and functional hardware capabilities and limitations

```
p4c --target p4tc l2_switch.p4 -o \
l2_switch.tmpl -c l2_switch.c -i l2_switch.json
```

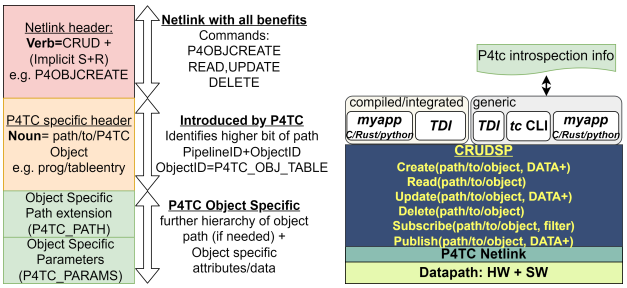
The compiler will generate several files. A template file named *l2_switch.tmpl*, an eBPF control code (*l2_switch.c*), an eBPF parser code (*l2_switch_parser.c*) and finally the JSON file (*l2_switch.json*). For the sample contents, see Appendix B. The program is loaded in the kernel by the dual steps of executing the template file as a script and loading the compiled eBPF program (*l2_switch.o*).

Once the program is loaded, it is ready for use but needs to be instantiated. It is achieved by creating a P4 filter - in this example, to a group of ports (see Appendix B for details on grouping ports using tc) of ports as such:

```
tc filter add block 123 ingress protocol all prio 10 \
 p4 pname l2_switch action bpf obj l2_switch_parser.o\
 section prog/tc-parser action bpf obj l2_switch.o \
 section p4prog/tc-ingress
```

3.3 P4TC Control Plane Interfacing

The P4TC control interface is designed to achieve high throughput rates by batching while also lowering latency for single requests. The kernel code is written to avoid unnecessary locking based on 1) deployment experiences and 2) published work on improving tc flower control interfacing [8]. P4TC uses standard netlink messaging for its control interface. The control API and messaging interface is designed to be oblivious of the P4 program. Figure 3 illustrates the netlink message layout and the API.



(a) Netlink Messaging **(b) Application Interface**

Figure 3: P4TC Messaging And API

Both the wire format and the API are based on an extended CRUD mechanism we defined as CRUDXPS (Create, Update, Delete, eXecute, Publish, Subscribe) semantic.

A control plane can *create*, for example, a table entry, *update* it, or *delete* it. A control plane can initiate *execution* of an eBPF program in the kernel to achieve a certain goal - for example, delete all entries that are below a certain *last used* timestamp value. A control plane can also *subscribe* to events from the kernel, such as table entry creation, deletion, or other events. A control plane can *publish* events through the kernel that will be distributed to subscribed listeners.

A data plane can *create* a P4 object, for example, a table entry, *update* it, or *delete* it. As mentioned, the data plane *executes* eBPF programs, which may invoke kfuncs into the P4TC domain. A data plane can also *subscribe* to events from the kernel, such as link events, and can *publish* events such as table entry creation, deletion, etc.

The API, as well as netlink messaging, can be described in the following grammar:

```
<VERB> < <NOUN> [OPTIONAL DATA] >+
```

Where **VERB** maps to one of *CRUDXPS* and the **NOUN** maps to a *path* pointing to the object (such as a table entry) and, depending on the verb used, **OPTIONAL DATA** may be present. It should be noted from the described grammar (see the "+") that a single request could batch many object instances. For example, the control plane can create/update/delete many table entries with a single control request - and the datapath can respond with many table entries to a single request targeted at a table.

A P4 program can express, via P4 annotations, access controls on what each plane can perform from the list of *CRUDXPS* verbs. For example, the control plane may be allowed to read but not update a specific table, whereas the datapath may be allowed to create, read, update, and delete an entry. See the description of the @tc_acl annotation further down in the document.

Figure 3a describes the netlink message. The common netlink header carries one of *CRUDXPS* commands. The common netlink header is followed by a header that identifies the target pipeline and object, such as a table entry. The object-specific header is then followed by an optional header carried in (Type-Length-Value, TLV) *P4TC_PATH* attribute, which provides a more refined path to identify a target object's details.

Figure 3b describes the general API overview (which maps nicely to the netlink messaging). Let us illustrate the grammar with some sample runtime control policies (see Figure 4) using the tc utility:

```
1 tc p4ctrl create l2_switch/table/FDB \
2 dstAddr 4a:96:91:5d:02:86 skip_hw \
3 action send_nh param port_id ens10
4 tc p4ctrl create l2_switch/table/FDB \
5 dstAddr 4a:96:91:6c:01:84 skip_sw \
6 action send_nh param port_id ens22
7 tc p4ctrl read l2_switch/table/FDB \
8 dstAddr 4a:96:91:6c:01:84
9 tc p4ctrl read l2_switch/table/FDB
```

The above runtime control policies demonstrate two table entries both on program/pipeline known as *l2_switch*. (*Lines 1-3*) instructs the P4TC kernel to add the entry to the software datapath (**skip_hw**). A packet matching the destination MAC address 4a:96:91:5d:02:86, arriving at any of the configured tc block of ports, is forwarded to the egress of port ens10. For these instructions, the grammar breakdown is as follows: the **VERB** is *create*, the **NOUN** is *l2_switch/table/FDB dstAddr 4a:96:91:6c:01:86 skip_hw* while the **DATA** is *action send_nh param port_id ens10*. The second table entry *lines 4-6* is added in the hardware datapath (**skip_sw**) to direct packets matching destination mac address 4a:96:91:6c:01:84 to port ens22. *Lines 7-8* shows a *read* of a single entry whereas *Line 9* shows a *read* of a whole table.

The operator can create one or more P4 programs and, with properly crafted control plane policies (using skip_sw/hw), can achieve the following:

- run entirely in the kernel-based software datapath (bare metal, containers, VMs),

- offload them entirely to hardware or
- run in a hybrid mode where some programs run in software and others in hardware,
- for each P4 program, put part of the pipeline in hardware and part in software exception datapath.

3.3.1 Writting Control Applications. Control plane applications can be crafted in multiple ways (see Figure 3b). The *generic* approach is to make the application consume the compiler-generated JSON introspection file for object identity resolution. As an example, the standard Linux *tc* utility inspects the JSON output in order to resolve human-friendly object names, paths, and values into machine-readable data. The resolved binary details are used in the netlink communication with the kernel. Just like the *tc* utility example being agnostic of any specific P4 program, one could write a trivial control application in their favorite language and reference the JSON file for that P4 program object resolution. Alternatively, one could write a program that embeds the JSON file contents to avoid constantly referring to the JSON data (resulting in improved latency) as indicated in the figure under "compiled/integrated" box.

3.4 P4TC Software Datapath

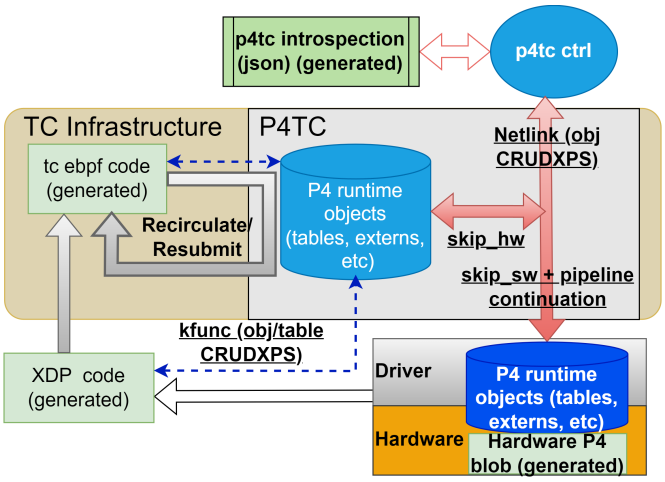


Figure 4: P4TC Control And Datapath

Figure 4 Illustrates the runtime control and datapath execution of the P4 program. All objects in green color are generated by the compiler and loaded before the program is instantiated, as described earlier. The blue-ish colored objects are assumed to be upstream, into either the kernel TC domain, driver, or user space *iproute::tc*. All P4 objects reside in the P4TC domain ("P4TC" grey box) and are controlled and configured via netlink.

An incoming packet may be processed entirely in hardware by the *Hardware P4 blob* - in which case the software side will not see the packet. A packet may be partially processed in hardware and then bubbled up the software stack alongside any metadata. Once in the software domain, depending on the P4 program and compilation, the packet and associated metadata may be processed by either or both of the XDP and TC layers. In either XDP or TC, the datapath flow may first go through a parser; the parsing results are stored in a location shared by both XDP and TC. The rest of the eBPF software datapath may access P4 objects in the P4TC domain by invoking appropriate kfuncs. Some sample of kfuncs are:

- *bpf_p4tc_tbl_read()* and *xdp_p4tc_tbl_read()*: Used to lookup a table entry. These calls take in the packet representation (skb, or xdp_md), pipeline ID, the table ID, a key and a key size. The kfuncs return either an entry or NULL. If the table entry is found and has been programmed by the control plane to have associated action, then appropriate action identification and associated attributes are returned.
- *bpf_p4tc_entry_create()* and *xdp_p4tc_entry_create()*: Used to create a table entry. These calls take either an skb or xdp_md, the pipeline ID, the table ID, a key and its size, and an action associated with the new entry. On success, a 0 is returned, and on failure, a negative error is returned.

To the reader familiar with P4, essentially the eBPF program maps closely to the P4 *control block* processing and any *apply()* within the control block that involves any object that has control plane parameters is diverted into the P4TC domain. With this split approach, we gain several things: 1) the control plane remains in the P4TC domain where we already have years of deployment experience and tooling using netlink for both software and hardware offload, 2) ensure the set of all possible P4 programs are covered despite the lack of Turing-completeness in eBPF.

3.4.1 Network Namespace Awareness. P4TC is namespace-aware (despite this feature not being built into eBPF); this means P4TC programs can be loaded into namespaces but remain isolated even if they have similar naming conventions. The network namespace identity is derived from either the skb (in the tc domain) or *struct xdp_md* (in the XDP domain).

3.4.2 P4 Tables. P4TC tables are conceptually similar to hardware TCAMs and the implementation could be labeled as an "algorithmic" TCAM which handles different lookup types specified in P4, namely: *exact*, *LPM*, *ternary*, and *ranges*. Tables have P4-defined attributes: a key of a specific size, a maximum number of entries and masks allowed. The kernel side is oblivious to all that and sees only bit blobs to which it applies masks before a lookup is performed.

4 CHALLENGES

4.1 Challenges with P4

P4, as a language, is primarily targeted at defining the datapath. However, in our bid to automate as much as possible, we had to come to terms with the fact that we needed to have the compiler generate some useful hints to the control plane. Some sample space:

- Annotations to provide better human readability for CLIs like *iproute2::tc*. An example is the *@tc_type* annotation used in the FDB table illustrated earlier. The FDB example uses "macaddr" to allow the CLI user to parse and print the key MAC address in a human-friendly octet format. Other types in this class are IPv4/6 addresses, linux "dev" names (e.g., eth0), etc.
- *@tc_acl* used to describe access control permission for P4 objects. As an example, a *const* table will be annotated with

@tc_acl="RS:RXP" which means the control plane (permission "RS") can **R**ead the table and **S**ubscribe to this tables' events but it cannot create, update or delete table entries. On the other hand, the datapath (permission "RXP") can only Read the table, execute entry-associated actions and publish events related to this table's entries.

4.2 Challenges with the Kernel

The biggest kernel challenge was to work around the kernel's approach of assuming a statically defined parser, tables and actions. Changes were needed to the core kernel to allow for the dynamic creation of table instances and action kinds. We achieved this goal by creating the templating infrastructure discussed earlier.

Another challenge we faced in the kernel was a social one. Ironically, it is directly related to our P4TC motivation towards kernel-independence. The earlier implementation of P4TC did not use eBPF [15]. Instead, it used a simple scripting approach definition language for the template. There was a pushback, primarily from folks using eBPF, that we should use eBPF. Two main reasons were cited: 1) better performance for the software datapath, 2) better security for the parser, given the existing kernel parser has had serious security issues. Our analysis [20] showed that under most circumstances eBPF software path performed better than our scripting approach, but, on the other hand, we lost the operational simplicity provided by the scripting approach. We nevertheless moved on and re-implemented the software datapath and templating approach as a compromise to accommodate the pushback. This effort cost us over 9 months of time.

5 PRELIMINARY PERFORMANCE NUMBERS

On a machine with an Intel Cascade Lake CPU, we achieved 10 Mpps of 64B packets per core forwarding using the example L2 switch program provided. This number does not scale linearly with the number of cores in Linux due to other overhead in the kernel, such as locking. Nevertheless, we did achieve the 35 Mpps forwarding rate required for wire speed 25G 64B packets with 6 cores.

On a VM with 4 allocated CPU cores (AMD Ryzen 4800H), we conducted two types of tests on an exact match table. The *best case* refers to table entries that have no actions and the *worst case* is for table entries with actions (with actions and related attributes being dynamically created - meaning incuring memory allocation cost):

(1) Adding 1M table entries individually as fast as possible, we get on a best case 641K entries/second/core and worst case 463K entries/sec/core. The result for 4 cores is not exactly linear. We get the best case of 1.78M entries/sec and the worst case of 1.2M entries/sec.
(2) Batching 16 table entries per message, we achieve in the best case 659K/s and worst case 657K/s. For 4 cores, we get best case 1.78M/s and worst case 1.62M/s.

6 RELATED WORK

The P4TC eBPF code generation is based on work done for the P4-eBPF [21] backend [18]. P4TC, by virtue of being able to run kfuncs, which were introduced in Section 3.2, overcomes the limitations in [21]. For example, the ability to do more computations and loops. P4TC currently focuses on the latest PNA architecture as opposed to PSA in [21]. There are several projects that generate code derived from a P4 program, detailed in [18], none of which provides the software-hardware symbiosis that P4TC offers.

7 FUTURE WORK

7.1 Improving Test Infrastructure

We have crafted over 300 control plane test cases using the tdc [12] infrastructure. While these tests have helped validate the control code functionality, we found the manual exercise to be tedious and likely missing some valid scenarios. For this reason, we plan to generate control plane test cases using the compiler. We hope to have a much wider coverage of testing as a result of this exercise. The datapath testing at the time of publication is manual - and to that end, we are looking at creating datapath test cases using *p4testgen* [22].

7.2 Extern Support

Traditionally, P4 externs have been tightly coupled with the hardware and provided by a vendor that implements both the datapath and control plane interfaces. We strive to provide an easy-to-use framework for anyone to provide user-defined, custom externs.

7.3 Hardware Support

We are working to add initial P4TC support to Intel IPUs. Other vendors are involved in P4TC offload discussions, with AMD/Pensando being more engaged and NVIDIA participating as well.

REFERENCES

[1] amdpensando 2023. *AMD Pensando™ Infrastructure Accelerators.* Retrieved October 23, 2023 from https://www.amd.com/en/accelerators/pensando
[2] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.
[3] conway [n. d.]. *Conway's Law.* Retrieved September 15, 2023 from https://en.wikipedia.org/wiki/Conway%27s_law
[4] DASH 2023. *Sonic Dash.* Retrieved Septemeber 13, 2023 from https://github.com/sonic-net/DASH
[5] ebpf 2023. *ebpf.* Retrieved September 15, 2023 from https://lwn.net/Articles/599755/
[6] Nick Feamster, Jennifer Rexford, and Ellen Zegura. 2014. The road to SDN: an intellectual history of programmable networks. *ACM SIGCOMM Computer Communication Review* 44, 2 (2014), 87–98.
[7] flower 2015. *flower - flow based traffic control filter.* Retrieved September 15, 2023 from https://man7.org/linux/man-pages/man8/tc-flower.8.html
[8] flowerctrl 2023. *P4TC.* Retrieved September 15, 2023 from https://netdevconf.info/0x15/session.html?Where-turbo-boosting-TC-flower-control-path-had-led-us-to
[9] Jamal Hadi Salim. 2015. Linux traffic control classifier-action subsystem architecture. *Proceedings of Netdev 0.1* (2015).
[10] intelipu 2023. *Intel® Infrastructure Processing Unit (Intel® IPU) ASIC E2000.* Retrieved October 23, 2023 from https://www.intel.com/content/www/us/en/products/details/network-io/ipu/e2000-asic.html
[11] kfunc 2023. *kfunc.* Retrieved September 15, 2023 from https://docs.kernel.org/bpf/kfuncs.html
[12] Linux TDC 2016. *tdc - Linux Traffic Control (tc) unit testing suite.* Retrieved October 23, 2023 from https://www.kernel.org/doc/readme/tools-testing-selftests-tc-testing-README
[13] matchall 2015. *matchall - traffic control filter that matches every packet.* Retrieved August 28, 2023 from https://man7.org/linux/man-pages/man8/tc-matchall.8.html
[14] Pere Monclus. 2023. *Programmability and Networking - Why do we bother?* Retrieved September 09, 2023 from https://opennetworking.org/wp-content/uploads/2021/05/2021-P4-WS-Pere-Monclus-Slides.pdf

[15] Netdev session 2023. *Netdev 2022 Session: Your Network Datapath Will Be P4 Scripted.* Retrieved October 26, 2023 from https://netdevconf.info/0x16/sessions/talk/your-network-datapath-will-be-p4-scripted.html
[16] offload1 2015. *Linux Offload Discussions.* Retrieved September, 2023 from https://netdevconf.info/0.1/sessions.html
[17] offload2 2016. *Linux Offload begginings.* Retrieved September, 2023 from https://www.youtube.com/watch?v=M6l1rxZCqLM
[18] Tomasz Osiński, Jan Palimąka, Mateusz Kossakowski, Frédéric Dang Tran, El-Fadel Bonfoh, and Halina Tarasiuk. 2022. A novel programmable software datapath for software-defined networking. In *Proceedings of the 18th International Conference on emerging Networking EXperiments and Technologies.* 245–260.
[19] P4C 2023. *P4C.* Retrieved September 09, 2023 from https://github.com/p4lang/p4c/tree/main/backends/tc
[20] P4TC evaluation 2023. *P4 2023 Workshop: P4TC-Kernel Implementation Approaches and Performance Evaluation.* Retrieved October 23, 2023 from https://github.com/p4tc-dev/docs/blob/main/p4-conference-2023/2023P4WorkshopP4TC.pdf
[21] psaebpf 2023. *psaebpf.* Retrieved September 15, 2023 from https://github.com/p4lang/p4c/blob/main/backends/ebpf/psa/README.md
[22] Fabian Ruffy, Jed Liu, Prathima Kotikalapudi, Vojtech Havel, Hanneli Tavante, Rob Sherwood, Vladyslav Dubina, Volodymyr Peschanenko, Anirudh Sivaraman, and Nate Foster. 2023. P4Testgen: An Extensible Test Oracle For P4-16. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23).* Association for Computing Machinery, New York, NY, USA, 136–151. https://doi.org/10.1145/3603269.3604834
[23] u32 2001. *u32 - Universal 32bit classifier.* Retrieved September 15, 2023 from https://man7.org/linux/man-pages/man8/tc-u32.8.html

A GENERATED SAMPLE SUMMARY

The following illustrates the generated template file:

```
tc p4template create pipeline/l2_switch \
    numtables 1 pipeid 1
tc p4template create action/send_nh \
    actid 1 param port type dev id 1
tc p4template create action/drop actid 2
tc p4template create table/FDB tblid 1 \
    keysz 48 type exact tentries 262144 \
    nummasks 32 permissions 0x35F \
    table_acts act name send_nh \
    act name drop flags defaultonly
```

And the following illustrates parts of the generated eBPF code:

```
struct
    p4tc_table_entry_act_bpf_params__local
    params = {
.pipeid = 1,
.tblid = 1
};
struct ingress_FDB_key key = {};
key.keysz = 0;
key.field0 = hdr->ethernet.dstAddr;
struct p4tc_table_entry_act_bpf *act_bpf;
struct ingress_FDB_value *value = NULL;
act_bpf = bpf_p4tc_tbl_read(skb,
    &params, &key, sizeof(key));
value =
    (struct ingress_FDB_value *)act_bpf;
 if (value != NULL) {
    switch (value->action) {
    case
   INGRESS_NH_TABLE_ACT_INGRESS_SEND_NH:
    {
      compiler_meta__->egress_port =
      value->u.ingress_send_nh.port;
      compiler_meta__->drop = false;
    }
    break;
    case
    INGRESS_NH_TABLE_ACT_INGRESS_DROP:
    {
      compiler_meta__->drop = true;
    }
    break;
    default:
    return TC_ACT_SHOT;
   }
 } else {
    return TC_ACT_SHOT;
 }
\end{lstlisting}

And last the JSON file:

\begin{lstlisting}[basicstyle=\small]
{
```

```
"schema_version" : "1.0.0",
"pipeline_name" : "l2_switch",
"id" : 1,
"tables" : [
  {
    "name" : "ingress/FDB",
    "id" : 1,
    "tentries" : 262144,
    "nummask" : 8,
    "keysize" : 48,
    "keyfields" : [
      {
        "id" : 1,
        "name" : "dstAddr",
        "type" : "macaddr",
        "match_type" : "exact",
        "bitwidth" : 48
      }
    ],
    "actions" : [
      {
       "id" : 1,
       "name" : "ingress/send_nh",
       "annotations" : [],
       "params" : [
         {
           "id" : 1,
           "name" : "port_id",
           "type" : "dev",
           "bitwidth" : 32
         }
       ],
       "default_hit_action" : false,
       "default_miss_action" : false
      },
      {
       "id" : 2,
       "name" : "ingress/drop",
       "annotations" : [],
       "params" : [],
       "default_hit_action" : false,
       "default_miss_action" : true
      }
    ]
  }
]
}
```

B ASSOCATING PORTS TO A PROGRAM

Note ports are added to tc block 123 devices using the tc as in the following example:

```
tc qdisc add dev ens7 ingress block 123
tc qdisc add dev ens8 ingress block 123
```