Honey for the Ice Bear – Dynamic eBPF in P4

Manuel Simon, Henning Stubbe, Sebastian Gallenmüller, Georg Carle
Chair of Network Architectures and Services, Technical University of Munich
Garching near Munich, Germany
{simonm|stubbe|gallenmu|carle}@net.in.tum.de

Abstract

Software updates typically require system reboots, leading to service downtimes. We aim to solve this problem for network components allowing updates while avoiding service degradation. In this paper, we explore the integration of eBPF into the P4 pipeline for efficient packet processing. This way, we combine the flexibility and dynamic adaptability of eBPF with the efficiency of P4. The integration enhances the power of applications and enables the network operator to provide customizable data paths as a service. Our solution allows updating the data path at runtime and without downtime. We implement the approach for the P4 target T4P4S, discuss different performance models, and share implementation insights. The evaluation focuses on the overhead in terms of throughput and the costs of code updates expressed in the latency of the related packets. We show that eBPF execution is possible with reasonable costs, promising dynamic network functions within P4.

CCS Concepts

• **Networks → Network performance analysis**; **Programmable networks**; *Middle boxes / network appliances.*

Keywords

P4, eBPF, SDN, Dynamic Network Function

ACM Reference Format:
Manuel Simon, Henning Stubbe, Sebastian Gallenmüller, Georg Carle. 2024. Honey for the Ice Bear – Dynamic eBPF in P4. In *Workshop on eBPF and Kernel Extensions (eBPF '24), August 4–8, 2024, Sydney, NSW, Australia.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3672197.3673436

1 Introduction

The ever-increasing amount of data traversing the Internet or application demand for low latency, shape the design of packet processing devices. Updates of components impact the latency and reliability of networks. In an ideal world, updates could be performed without impacting the service quality of the network. Software-defined network components equip us with the required tools to implement such seamless updates. In recent years, two powerful technologies for flexible packet processing emerged: *P4* and *eBPF*.

P4 [7] allows programming high-performance packet processing in hardware or software, both in middleboxes and (with the upcoming PNA [8]) at end hosts. Network operators can tailor the packet processing to their needs. Though powerful, P4 functionality is limited. For example, P4 supports addition, subtraction, and multiplication only in powers of two and often relies on external functionality provided by the specific target but not directly supported by the P4 language. Without standardization, the APIs and the implementation of these externs vary between targets, severely limiting the portability of P4 code using externs.

eBPF (extended Berkeley Packet Filters) [13] is a growing and powerful language with fewer constraints than P4, especially for calculations. Its target-independent byte code runs on a VM allowing the extension of network stacks with user-defined functionality.

The P4 language is kept intentionally simple to allow fast packet processing. On the other hand, eBPF offers a higher degree of flexibility with specific restrictions to ensure execution times, e.g., bounded loops. Tackling problems from different angles, eBPF and P4 offer similar function at different costs. What stands out particularly is the flexibility of eBPF and the performance achievable with P4. Thus, integrating eBPF into the P4 pipeline may unleash a more powerful and flexible way to program packet processing devices.
More powerful processing: P4 allows only basic arithmetical operations, minimizing complexity and maximizing performance. eBPF allows expressing advanced functionality without vendor-specific extensions, increasing its portability. Programs run on different targets implementing the eBPF runtime environment. Complex applications are realized using eBPF, such as DDoS mitigation [4], IDS/firewalls [5, 37], monitoring [2], or even more complex network functions [27], like a 5G Mobile Gateway [34].
Dynamic reprogramming: eBPF is designed to be just-in-time compiled, enabling switching of its functionality without stopping the execution, in contrast to P4. Integration into the P4 pipeline introduces eBPF's dynamic and adaptive changes to high-performance P4 data plane processing. This integration goes beyond plain migration of network functions to other devices in case of failures. For example, network operators can offer a programmable data path as a service to their customers. Customer-defined functionality is applied to their flows. Utilizing eBPF, these customized network functions allow more complex processing beyond simple rule or table updates in P4. For instance, customized flow monitoring can be installed, adapted, or removed at runtime. Furthermore, tenants may use network resources for customized in-network computation. An advantageous property of eBPF is its portability utilizing platform-indepedent byte code that is translated to high-performance machine code on the target platform.

The promising advantages are worth investigating dynamic network functions leveraged by eBPF in the P4 pipeline: This paper (1) discusses approaches integrating eBPF into the P4 pipeline, (2) provides insights into implementation considerations, and (3) presents an overview of performance implications. Our implementation extends T4P4S [41]. We demonstrate eBPF execution and its dynamic re-programmability at reasonable costs and throughput rates.

2 Background & Related Work

P4: P4 [7] uses a pipeline to process packets, cf. Figure 1 (in blue), starting with a *parser*. Afterward, packets traverse *match-action* pipelines. There, match-action tables, the heart of P4, determine the control flow of the packets. Header fields are matched against table entries, specifying actions to be executed and its parameters. In the end, packets are *deparsed*. P4 offers full programmability of all stages, non-P4 functionality can be used as *externs*.

Different targets for P4 exist: *Hardware targets* provide the highest performance in terms of throughput and latency. The targets range from ASICs, such as the Intel Tofino [23], over SmartNICs, e.g., Agilio Netronome [3], to FPGAs, i.e., the P4→NetFPGA workflow [20] or the Intel P4 Suite for FPGA [22]. Hardware targets usually follow a pipelined approach, processing several packets at different stages at the same time. Thus, synchronization between several stages is more complex. *Software targets*, on the other side, provide the highest degree of flexibility. With lower performance, software targets run on commodity hardware and allow the easy integration of new functionality. bmv2 [33] is the P4 reference implementation for developing, but offers limited performance. T4P4S [41] and P4-DPDK [30] rely on the Data Plane Development Kit (DPDK) [35], a userspace framework for high-performance packet processing, providing better performance. P4TC [17] is an implenentation in the Linux Kernel using Traffic Control (`tc`). Software targets may either follow a pipelined, or, for a better performance, a run-to-completion model, processing packets iteratively [10].

eBPF Data Planes: The concept of programming the data plane with eBPF has been discussed previously. Jouet et al. [24] built BPFabric, a programmable software switch integrating eBPF. They demonstrated BPFabric's capability for complex packet processing. The uBPF project [1] ported the in-kernel eBPF VM to userspace. eBPF expressiveness is demonstrated in the P4 to eBPF/ uBPF/ XDP compiler back-ends [31, 32, 39], generating eBPF code out of P4. A concept that was further extended by Osiński et al. [28]. Apart from P4 targets, eBPF was also ported to other network devices. For example, Tu et al. [40] integrated eBPF into Open vSwitch with OVS-eBPF and -AFXDP.

Hardware Offloading: NIC offloading is an active field of research. XDP (eXpress Data Path) [19] is a high-performance data path offering a low-level interface for eBPF. The concept is similar to our approach, offloading specific functionality from the usual P4 processing. There exist specialized processors for both languages, especially in SmartNICs [3]. Kicinski et al. [25] showed a method to offload eBPF/XDP execution to SmartNICs, allowing an accelerated execution on Netronome NFP-based NICs. Salva-Garcia et al. [36] built a framework that allows offloading network functions using eBPF and XDP. They realized their implementation on Netronome SmartNICs.

Runtime Programmability: Changing functionality on data planes during runtime has been investigated as part of (capsule-based) active networking [38]. Das et al. [9] implemented an instruction set in P4 to express packet processing tasks that can be changed during runtime. Xing et al. [42] developed FlexCore enabling a partial reconfiguration of data planes at runtime. Feng et al. [15] extended P4 to enable in-situ programmability. Our approach also allows runtime updates but relies on the well-defined eBPF language.

All related works investigate a single language or technology to solve a problem. We take a different perspective by combining two technologies, P4 and eBPF, keeping their specific advantages. This way, packet processing tasks can be expressed in P4 and partially extended where eBPF is more appropriate to use or dynamical reprogrammability is required.

3 Approach

In this section, we describe fundamental approaches for eBPF execution environments in P4 and discuss advantages and disadvantages of possible use cases and their implementation. We distinguish between a *fixed* and a *flexible* placement of eBPF programs in the P4 pipeline. Both approaches can be combined with different modes regarding the dynamicity of the eBPF program: *static*, *pre-defined*, or *extensible*. Extensibility requires security mechanisms to prevent unauthorized eBPF program replacement. Approaches, modes, and their security implications will be discussed subsequently.

3.1 Placement in the P4 Pipeline

First, the question arises where in the P4 pipeline the eBPF functionality may be called. This question influences the capability of such a hybrid data plane and its requirements. The first approach (*fixed*), extends the P4 v1model to allow pre- and post-processing packets in eBPF, before, after, or in between the P4 pipeline. The second, dubbed *flexible*, allows the functionality to be called within the P4 pipeline and follows the usual *extern* approach of P4.

Fixed Components: The *fixed* approach extends P4's well-established v1model, depicted in Figure 1. The eBPF processor(s) can be placed at three different locations:

(1) eBPF as a *pre-processor*: That way, the pre-processor pre-filters the packets before they arrive at the P4 pipeline. The return value of the executed eBPF program determines if packets are dropped or forwarded to the P4 pipeline. This pre-processing helps relieve the P4 pipeline, potentially decreasing the number of packets traversing the pipeline. Such relief is of particular interest if: a) the execution of the P4 pipeline is expensive, i.e., the P4 program is complex, or b) the incoming traffic contains a considerable percentage of packets to be dropped anyway. Apart from dropping, arbitrary pre-processing of packets is possible; therefore, e.g., the use cases from Section 1 apply.

(2) eBPF as a *mid-processor*: Located between ingress and egress pipeline, e.g., in the traffic manager. There, calculations on header fields may be performed, which cannot be expressed in plain P4.

(3) eBPF as a *post-processor*: The post-processor can either perform final actions, such as calculations, cryptography/hashing, or decide to drop or forward a packet.

All options have in common that an enabled processor will be executed for every packet. Processing a subset of packets is only possible if the P4 pipeline itself drops them. Moreover, the processing is performed on the whole packet. Therefore, the eBPF program can access and modify the entire packet. In case the program was delivered by a third party, this access can be considered harmful

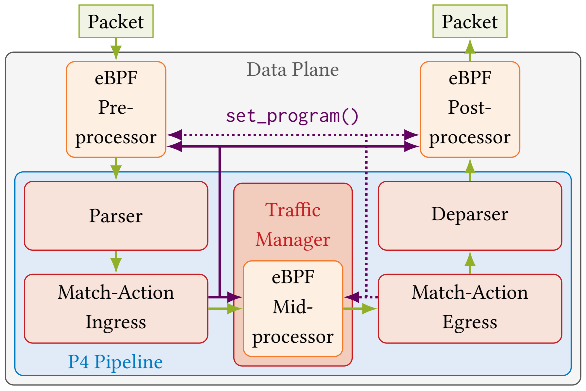

Figure 1: Fixed eBPF components

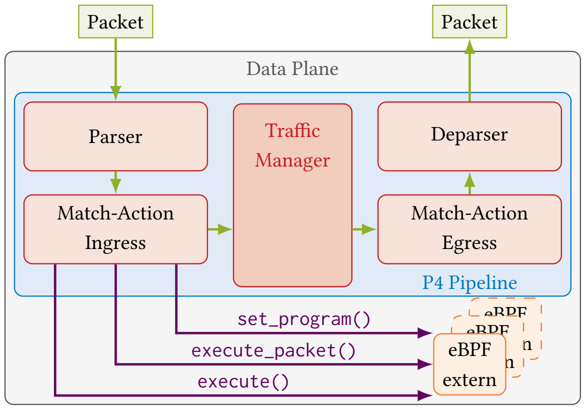

Figure 2: Flexible eBPF externs

for security reasons. Conversely, a fixed place of the eBPF functionality in the processing path eases the implementation, especially in hardware targets. For software targets, the processing can be done in batches; for hardware targets, especially SmartNICs, fixed eBPF/XDP processors already exist, e.g., in the Netronome Agilio CX [3].

Flexible Externs: The *flexible* approach is based on P4 *externs* shown in Figure 2. Externs can be used to add new non-P4 functionality to supported targets. This approach defines a new eBPF extern, which can be used several times and anywhere in the P4 program. Thus, different functionalities can be placed in the eBPF externs. There are two possible options: Depending on the API call, the extern may access and modify either the whole packet or read only specified header fields. Accessing entire packets offers the highest flexibility and, e.g., allows to compute hashes on the whole packet. With read-only access to specific header fields, no header field modification is possible; only the return value is accessible from the P4 pipeline. However, it decreases the amount of data leaked to the eBPF extern. In both cases, eBPF extern execution can be limited to a subset of the processed packets, e.g., by putting the call in an `if`-statement. Such conditional calls decrease the overhead of the eBPF execution for undesired packets significantly increasing flexibility. However, conditional execution may reduce the performance of software targets due to branch and cache misses. Plus, it may be harder to implement on hardware targets due to the pipelined approach. Furthermore, the eBPF processors on a hardware target may be limited in their amount of available resources.

3.2 eBPF Functionality

Both fixed and flexible approaches support three modes of eBPF functionality: *static*, *pre-defined*, and *extensible*.

Static: In static mode, fixed, non-changeable functionality is bound to the components during initialization, extending the P4 pipeline.

Pre-defined: In the pre-defined mode, a pre-implemented and fixed set of functionality is given during initialization, which can be activated and bound to the eBPF modules on demand during runtime. The functionality can be specified, e.g., by the path of the program. The network operator may provide different functionalities already implemented in eBPF, which the tenant can activate. Then, the tenant can pick functions out of the provided templates and start the execution for the upcoming packet from the flow belonging to them. As a result, the network operator can ensure that the programs meet given requirements, such as runtime boundaries or security.

Extensible: The extensible mode goes one step further. There, new programs can be sent to the P4 target's data plane. eBPF programs may be specified using C code or eBPF binaries. The inputs are processed on the P4 target and bound to the given eBPF module, as before. eBPF binary deployment offers benefits: Avoiding compilation from source to byte code reduces overhead and utilizes eBPF's platform independence; the same byte code binary can be used on any target supporting eBPF and is JIT compiled to machine code at runtime. Using the extensible way, updates of the data plane functionality is possible without interruption. Tenants have almost complete freedom to decide on the packet processing of their flows; a flow or stream could define itself, how it should be treated. Moreover, functionality can be moved from one device to another for redundancy or failure recovery.

3.3 Security

The possibility to remotely change code requires appropriate security mechanisms. We identified two main concerns: the authenticity and trustworthiness of code updates. For both, we suggest measures to secure dynamic functions:

Authenticity: Code updates must only be triggered by selected, authenticated, and authorized tenants. Otherwise, an adversary may compromise packet processing tasks. Authenticity is ensured by adequate cryptographic procedures. Additionally, tenants should only be allowed to change the processing of their own flows. In P4, this can be handled using appropriate tables and table entries.

Trustworthiness: Remotely installing new code is potentially dangerous. Therefore, the execution of the eBPF code must be strictly limited to the tenant's own packets, not impacting the processing of any other flow. Further, harm to the data plane processing itself must be prevented. Unrestricted code execution requires a high level of trust between network operators and tenants. Otherwise, the executed programs must be isolated or restricted in their power. Fittingly, isolation of eBPF is one of its design goals since it runs on a virtual machine. Originally, the Linux kernel has enforced the isolation running it in the kernel space. This is different when eBPF runs in user space or on the data plane. On the other side, the eBPF program then has fewer rights given by the operating system and it is still isolated from other processes. Another possible approach is to restrict the allowed instructions only allowing instructions that will not harm the system, such as arithmetical, logical, or conditional operations. For that, the source code or the compiled eBPF binary can be checked using static analysis.

4 Implementation

Following the introduction of possible approaches, this section discusses the implementation thereof. We implement all the approaches for eBPF in the P4 software target T4P4S. As an open-source software target, T4P4S is extensible. T4P4S transpiles P4 code into C code linked to the state-of-the-art packet processing framework DPDK, allowing high performance. Therefore, T4P4S is an excellent choice for the creation of prototypes. Our implementation is based on commit `2308915` [29]. T4P4S uses the batch-processing of DPDK and follows the run-to-completion model to maximize performance [10]. A batch of packets is received from the NIC at once. Then, each packet traverses the generated P4 pipeline iteratively. Afterward, the whole batch is sent out. This model minimizes expensive memory accesses and optimizes throughput.

For the execution of the eBPF code, we rely on the built-in eBPF support of DPDK (which is also the foundation of T4P4S) using the `rte_bpf` library [11]. It allows JIT compilation for x86_64 architectures and offers tx/rx-device callbacks to the DPDK program [12], which our implementation uses for the *pre-* and *post-processor*. Hence, every packet received or sent is put into the eBPF execution callback in a batched way. Conveniently, DPDK [12] supports dropping packets if the return value of the eBPF program is zero. For the *mid-processor*, we use a non-batched eBPF call, which is executed on each packet between the generated code for ingress and egress processing. The limitation of this implementation is that packet modification only works if the valid headers remain unchanged. This limitation is caused by T4P4S which reorders packet headers at the end of the P4 pipeline, at the deparser. Up to this point, the headers may be located at other than the expected offsets.

Implementing the eBPF *externs* works similarly to the mid-processor relying on the DPDK eBPF execution environment. If the read-only call is performed, the specified header fields are copied into one continuous memory area, which is handed over to the eBPF program as a pointer. In case the eBPF processing is performed on the whole packet, the same requirements for header reordering stand.

As mentioned, both approaches work in different modes. In the *static* mode, the specified program, i.e., the path to its compiled version, is bound to the component during the initialization of T4P4S; in *pre-defined* the program can be bound and changed during runtime specifying a new, but pre-defined path in the P4 program. In the *extensible* mode, new programs can be added and executed during runtime either by source C-files or by compiled eBPF binaries. These binaries are written to a temporary file, which is bound to the component on the fly. However, in case of the callback-driven *fixed* components, the change will only take effect starting with the next batch of packets. Source files must be compiled before binding. The compilation uses *clang*: `clang -O3 -target bpf -c dummy.c`

Before installation, we check the authenticity of code updates using BLAKE3 [6]-based message authentication codes (MAC) with a 256-bit pre-shared symmetric key. The validation is not part of the P4 pipeline but occurs directly inside the eBPF modules. If the authenticity cannot be validated, the update is rejected.

Our source code and example programs are publicly available at GitHub [26]. Despite minor fixes, it introduces a latency-optimized version used for our evaluation and the support of the presented eBPF components.

5 Evaluation

In this section, we describe the setup and methodology of our experiments. Furthermore, we show and discuss their results. First, the influence of eBPF processing on the forwarding performance is investigated. Second, the occurring latencies around dynamic function changes are analyzed in detail.

5.1 Setup

Our evaluation uses a three-host setup. All three nodes are equipped with an Intel Xeon CPU D-1518 (4×2.2 GHz), 32 GB RAM, and dual-port Intel X552 NICs (2×10 Gbit/s). The Device under Test (DuT) runs on Debian Bullseye (kernel 5.10), executing the eBPF-capable version of T4P4S on a single isolated CPU core. Each received packet is processed by the DuT and forwarded back to the load generator (LoadGen). On the LoadGen, we use MoonGen [14] to create traffic for the DuT. The traffic consists of 84 B sized packets (88 B with CRC), each featuring a unique identifier. Both links are monitored via passive optical splitters on the timestamper host. The timestamper timestamps each incoming packet with a precision of 12.5 ns [21]. Out of the timestamped data, the latency can be calculated by matching unique identifiers. Our experiments are orchestrated using pos [16] to ensure reproducibility.

5.2 Overhead of eBPF components

At first, we measure the induced overhead of the eBPF programs, i.e., the *static* mode. For reference, we use a *baseline* P4 program, forwarding all packets without any eBPF execution enabled. We compare the baseline with the execution of eBPF programs at the different possible fixed locations before (*pre*) or after (*post*) the pipeline, or at the traffic manager (*mid*) with the execution as *extern*. The latter is done for both, the whole *packet* or only two 32-bit wide fields. To highlight the impact of different functionality, we use three different programs: (1) *dummy* program just returning a non-zero value to quantify the overhead of the eBPF execution itself, (2) *filter* program emulating a pre-filtering of incoming packets checking for one blocked source IPv4 address and one blocked UDP port. Note that real filtering can only be done in the pre or post position; and (3) a *change* program, which changes the source IPv4 address emulating a write access to the packet. Note that packet modification is not possible in the normal extern.

The maximum throughputs are depicted in Figure 3. First, we compare the maximum throughputs for the *dummy* programs. The fixed components perform better than the externs. The pre- and post-processing decreases the throughput by ≈ 6.8 % and 6.0 %,

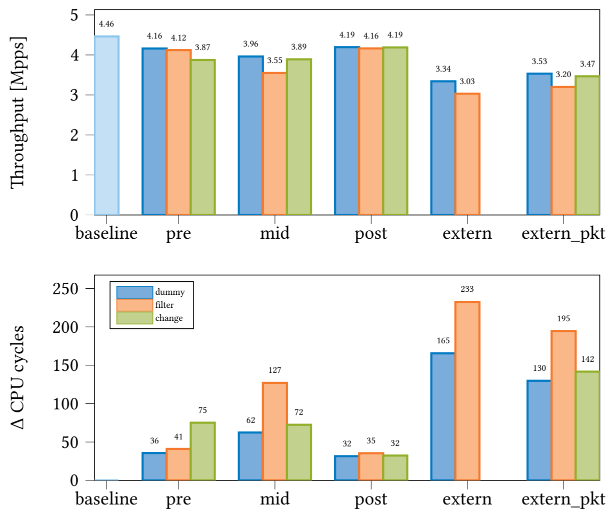

Figure 3: Overhead of eBPF functionality

respectively. Both processors are topped by the mid-processor's decrease of ≈ 11.23 %. The batched DPDK callback implementation increases processing efficiency compared to the non-batched one in the middle. Executing eBPF as externs decreases the forwarding performance by ≈ 25.1 % if only some header fields are used and by ≈ 20.9 % if the whole packet is passed to the eBPF function. The first case is more expensive as the two header fields must be copied first. In the second case, the packet is handed over by reference, which comes with the limitations mentioned in Section 4.

We can determine the average of used CPU cycles (C) per packet using CPU frequency (f_{cpu}) and packet rates (r). Results obtained with this model are depicted in Figure 3:

$$C = (f_{cpu}/r_{testcase}) - (f_{cpu}/r_{baseline})$$

We observe a similar performance behavior for the other programs. The overhead is smaller for the fixed positions than for the flexible externs. The more complex logic for the filter program also increases the modeled CPU cycles for the *pre* position. Packet modification is even more expensive. This changes for the mid and post position. There, filtering seems to be more costly. We assume caching effects to be the root cause. Before the P4 pipeline, the packets and their contents had not been touched. Then the usual rules are valid, that modifications are more expensive than lookups, even if the logic is simpler. After traversing (parts of) the P4 pipeline in T4P4S, the packet data has been fetched to the cache. As a result, the modification of already cached data becomes less expensive. Additionally, the more complex logic for filtering (two conditional checks instead of one straightforward modification) becomes more expensive in terms of CPU cycles.

5.3 Dynamic Program Loading

Dynamic changes of the loaded program introduce additional costs. To investigate these costs, we use a latency-optimized version of T4P4S with a batch size of one. This enables us to see the impact of loading a new program directly in the measured latency. To investigate a non-overloaded system, we use a moderate packet rate of 116 kpps (100 Mbit/s). First, the *dummy* program is loaded and applied to the first 10^6 packets. After that, the next packet contains the new program to be installed (change packet). In our case, we install another *dummy* program to only measure the influence of the change. Afterward, another $4 \cdot 10^6$ packets traverse the data plane with the new program. We investigate the impact of the three possible types of updates: loading a new program from *memory* (i.e., *pre-defined*), from the *source* code, or the *binary* given in the change packet (i.e., both *extensible*). For the two extensible approaches, we also measure the authentication overhead. Further, we investigate the impact for the *extern* and the *pre-processor* since mid- and post-processor work similarly.

Figure 4 depicts the latencies for the packets before, during, and after the changes for the *pre-processor* and the *extern* positions for one test run. Figure 5 shows the latencies of only the packet changing the program for ten test runs.

Memory: Update per memory means that the incoming change packet specifies an ID that is mapped to a program that should be loaded. The change has, as expected, a significantly higher latency (median: 168 µs) since there the new program is applied. A few following packets are also affected. The reason for the increased latency may be either the filled buffer, or the worse branch prediction/cache optimization directly after the new program is executed the first times. Afterward, the latency is the same as before since the semantics of the new program are the same. A similar picture is drawn for the extern with the difference that the latency of the change packet is smaller (median: 109 µs). In the extern, the new program has just to be loaded into the execution environment, while in the pre-processor, it has additionally to be bound as the callback for the queues.

Source: Here, the change packets contain the source code, which is compiled and bound to the execution environments. The latencies are depicted for the pre-processor and the extern in Figures 4-1b) and 2b). We used the `-O0` compiler flag to reduce the compilation time and its contribution towards latency. However, even then, the approach is subjected to high latency (44 ms, 36 ms), resulting in filled buffers and subsequent packet loss. Choosing a better but more complex optimization strategy (e.g., `-O3`) would lead to worse behavior at the moment of the program change.

Binary: To avoid the compilation delay to byte code, we investigate the dynamic integration of eBPF binaries. Figures 4-1c) and 2c) show the latencies for the two positions. In both, the latency of the change packet is increased but regresses swiftly. Again, the change introduces less latency to the extern (median: 127 µs) than to the pre-processor (median: 174 µs).

Authenticated updates: As discussed, dynamic code changes have to be secured. Authentication introduces a median latency of 180 µs (+6 µs) for the pre-processor and 130 µs (+3 µs) for the extern. Given the limited impact of authentication on latency, authenticated dynamic updates are feasible.

6 Discussion

We implemented and investigated the integration of eBPF into the software target T4P4S. When implementing it into hardware targets, different requirements arise. Fixed position components may be easier to integrate than flexible externs. Preprocessing in

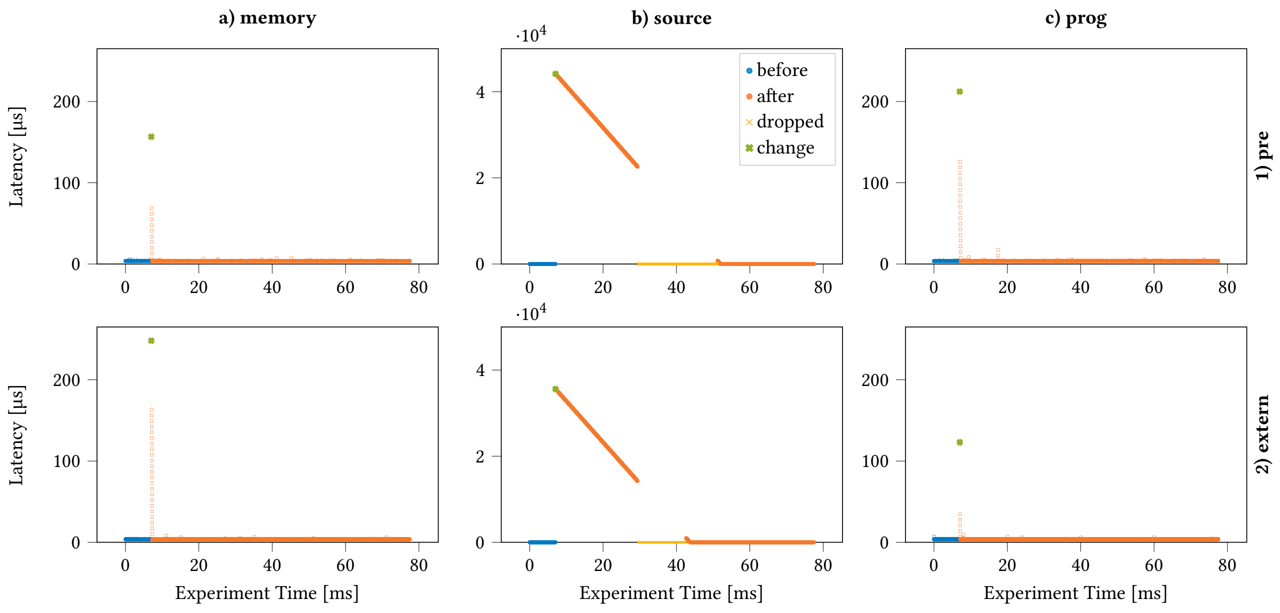

Figure 4: Latencies before, during, and after change packet

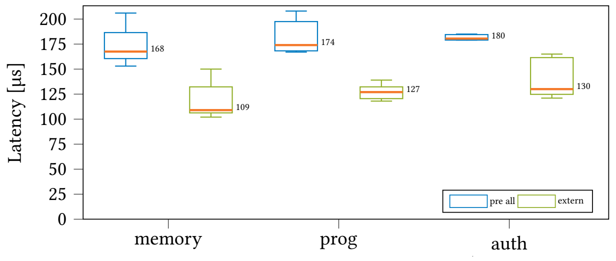

Figure 5: Latency of change packet (median as number)

eBPF is already performed on SmartNICs [3]. Implementing externs inside the P4 pipeline becomes more challenging due to the fixed clock rates between the pipeline stages. Conditional eBPF execution would likely not result in a performance gain for the packets that do not use the extern. Instead, the latencies are likely to be constant, independent of the executed control flow of each packet [18]. Another difficulty is to synchronize the execution times required for each stage. For that, the existing validators for eBPF guaranteeing maximum cycle counts can help. The same requirements hold for extensible updates, and maximum cycles have to be defined. Again, validators can help calculate maximum cycles. This requirement is more relaxed in software targets, i.e., run-to-completion targets.

eBPF helps ease runtime adaptability for hardware and software targets. P4 is a domain-specific language designed for packet processing exclusively; therefore, its execution can be optimized in hardware targets. The results show that providing new functionality by distributing its source code is not feasible. Conversely, a natively compiled binary can only be used on one specific platform. As eBPF can be distributed as platform-independent byte code and, as the results show, be installed in a timely manner, it is well-suited to enable such mechanisms. The same binary can be used for all targets, hardware, or software. The JIT compilation unleashes full performance, optimizing it for the underlying machine architecture.

7 Conclusion

We discussed, implemented, and evaluated different approaches to offload eBPF execution within P4. The overhead is smaller for fixed-position components than for flexible externs. Fixed-position components are likely easier to integrate into hardware targets. However, externs are more flexible in their usage. For dynamic changes, the fastest option is to activate pre-defined eBPF programs. However, the more powerful extensible updates, relying on eBPF binaries, are feasible. Dynamic updates allow an interrupt free service of the network. A dynamic network function can be implemented and secured, leveraging authenticated updates. On the other hand, sending dynamic updates using the source code proved impractical due to the significant compilation overhead, which eventually causes packet loss.

The results demonstrate that eBPF execution with dynamic and seamless updates is possible, enabling a variety of new applications. The source code of our implementaion is available on GitHub [26].

Acknowledgments

The authors thank Timon Tsiolis for his contributions and the fruitful discussions, and the reviewers for their valuable feedback. This work is partially funded by the European Union's Horizon 2020 research and innovation programme (grant agreement no. SLICES-PP 101079774 and GreenDIGIT 101131207). The German Federal Ministry of Education and Research (BMBF) supported our work under the projects 6G-life (16KISK002) and 6G-ANNA (16KISK107) as well as the German Research Foundation (DFG) as part of the HyperNIC (CA595/13-1) project.

References

[1] Lane, Rich and others . 2024. GitHub uBPF - Userspace eBPF VM. https://github.com/iovisor/ubpf Last accessed: 2024-05-24.

[2] Marcelo Abranches, Oliver Michel, Eric Keller, and Stefan Schmid. 2021. Efficient Network Monitoring Applications in the Kernel with eBPF and XDP. In *2021 IEEE Conference on Network Function Virtualization and Software Defined Networks, NFV-SDN 2021, Heraklion, Greece, November 9-11, 2021*. IEEE, 28–34. https://doi.org/10.1109/NFV-SDN53031.2021.9665095

[3] Agilio CX SmartNICs. 2024. Netronome. https://www.netronome.com/products/agilio-smartnics/ Last accessed: 2024-05-24.

[4] Gilberto Bertin. 2017. XDP in practice: integrating XDP into our DDoS mitigation pipeline. In *Technical Conference on Linux Networking, Netdev*, Vol. 2. The NetDev Society, 1–5.

[5] Matteo Bertrone, Sebastiano Miano, Fulvio Risso, and Massimo Tumolo. 2018. Accelerating Linux Security with eBPF iptables. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 108–110. https://doi.org/10.1145/3234200.3234228

[6] BLAKE3-team. 2024. GitHub: BLAKE3-team/BLAKE3. https://github.com/BLAKE3-team/BLAKE3/tree/master/c Last accessed: 2024-05-24.

[7] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: programming protocol-independent packet processors. *Comput. Commun. Rev.* 44, 3 (2014), 87–95. https://doi.org/10.1145/2656877.2656890

[8] The P4 Language Consortium. 2021. P4 Portable NIC Architecture (PNA), version 0.5. https://p4.org/p4-spec/docs/PNA.html Last accessed: 2024-05-24.

[9] Rajdeep Das and Alex C Snoeren. 2023. Memory Management in ActiveRMT: Towards Runtime-Programmable Switches. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 1043–1059. https://doi.org/10.1145/3603269.3604864

[10] Mihai Dobrescu, Norbert Egi, Katerina J. Argyraki, Byung-Gon Chun, Kevin R. Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. 2009. RouteBricks: exploiting parallelism to scale software routers. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, Jeanna Neefe Matthews and Thomas E. Anderson (Eds.). ACM, 15–28. https://doi.org/10.1145/1629575.1629578

[11] DPDK. 2021. DPDK documentation—rte_bpf library. https://doc.dpdk.org/api-21.08/rte__bpf_8h.html Last accessed: 2024-05-24.

[12] DPDK. 2021. DPDK documentation—rte_bpf_ethdev library. https://doc.dpdk.org/api-21.08/rte__bpf__ethdev_8h.html Last accessed: 2024-05-24.

[13] eBPF community. 2024. eBPF - Introduction, Tutorials & Community Resources. https://ebpf.io/ Last accessed: 2024-05-24.

[14] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. 2015. MoonGen: A Scriptable High-Speed Packet Generator. In *Proceedings of the 2015 ACM Internet Measurement Conference, IMC 2015, Tokyo, Japan, October 28-30, 2015*, Kenjiro Cho, Kensuke Fukuda, Vivek S. Pai, and Neil Spring (Eds.). ACM, 275–287. https://doi.org/10.1145/2815675.2815692

[15] Yong Feng, Zhikang Chen, Haoyu Song, Wenquan Xu, Jiahao Li, Zijian Zhang, Tong Yun, Ying Wan, and Bin Liu. 2022. Enabling In-situ Programmability in Network Data Plane: From Architecture to Language. In *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*, Amar Phanishayee and Vyas Sekar (Eds.). USENIX Association, 635–649. https://www.usenix.org/conference/nsdi22/presentation/feng

[16] Sebastian Gallenmüller, Dominik Scholz, Henning Stubbe, and Georg Carle. 2021. The pos framework: a methodology and toolchain for reproducible network experiments. In *CoNEXT '21: The 17th International Conference on emerging Networking EXperiments and Technologies, Virtual Event, Munich, Germany, December 7 - 10, 2021*, Georg Carle and Jörg Ott (Eds.). ACM, 259–266. https://doi.org/10.1145/3485983.3494841

[17] Jamal Hadi Salim, Deb Chatterjee, Victor Nogueira, Pedro Tammela, Tomasz Osinski, 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 on European P4 Workshop (EuroP4 '23)*. Association for Computing Machinery, New York, NY, USA, 25–32. https://doi.org/10.1145/3630047.3630193

[18] Eric Hauser, Manuel Simon, Henning Stubbe, Sebastian Gallenmüller, and Georg Carle. 2022. Slicing Networks with P4 Hardware and Software Targets. In *5G-MeMU '22: Proceedings of the ACM SIGCOMM Workshop on 5G and Beyond Network Measurements, Modeling, and Use Cases, Amsterdam, The Netherlands, August 22, 2022*, Özgü Alay and Ying Wang (Eds.). ACM, 36–42. https://doi.org/10.1145/3538394.3546043

[19] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '18)*. Association for Computing Machinery, New York, NY, USA, 54–66. https://doi.org/10.1145/3281411.3281443

[20] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. 2019. The P4->NetFPGA Workflow for Line-Rate Packet Processing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 1–9. https://doi.org/10.1145/3289602.3293924

[21] Intel. 2023. Intel Ethernet Controller X550 Datasheet rev 2.7. https://www.intel.com/content/www/us/en/content-details/333369/intel-ethernet-controller-x550-datasheet.html Last accessed: 2024-05-24.

[22] Intel. 2024. Intel P4 Suite for FPGAs. https://www.intel.com/content/www/us/en/software/programmable/p4-suite-fpga/overview.html Last accessed: 2024-05-24.

[23] Intel. 2024. Intel® Tofino™ Series Programmable Ethernet Switch ASIC. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html Last accessed: 2024-05-24.

[24] Simon Jouet and Dimitrios P. Pezaros. 2017. BPFabric: Data Plane Programmability for Software Defined Networks. In *2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 38–48. https://doi.org/10.1109/ANCS.2017.14

[25] Jakub Kicinski and Nicolaas Viljoen. 2016. eBPF Hardware Offload to SmartNICs: cls bpf and XDP. *Proceedings of netdev* 1 (2016).

[26] manuel simon. 2024. GitHub manuel-simon/t4p4s - Retargetable compiler for the P4 language (fork) - branch: ebpf. https://github.com/manuel-simon/t4p4s/tree/ebpf Last accessed: 2024-06-18.

[27] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. 2018. Creating Complex Network Services with eBPF: Experience and Lessons Learned. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. 1–8. https://doi.org/10.1109/HPSR.2018.8850758

[28] Tomasz Osinski, Jan Palimaka, 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, CoNEXT 2022, Roma, Italy, December 6-9, 2022*, Giuseppe Bianchi and Alessandro Mei (Eds.). ACM, 245–260. https://doi.org/10.1145/3555050.3569117

[29] P4ELTE. 2024. GitHub P4ELTE/t4p4s - Retargetable compiler for the P4 language - commit: a3a54e3. https://github.com/P4ELTE/t4p4s/commit/a3a54e37521dcc61365d09dd705c3709a533e07a Last accessed: 2024-06-18.

[30] p4lang. 2024. GitHub p4c/backends/dpdk - DPDK backend. https://github.com/p4lang/p4c/blob/main/backends/dpdk/README.md Last accessed: 2024-05-24.

[31] p4lang. 2024. GitHub p4c/backends/ebpf - eBPF Backend. https://github.com/p4lang/p4c/blob/main/backends/ebpf/README.md Last accessed: 2024-05-24.

[32] p4lang. 2024. GitHub p4c/backends/ubpf - uBPF Backend. https://github.com/p4lang/p4c/blob/main/backends/ubpf/README.md Last accessed: 2024-05-24.

[33] p4lang. 2024. GitHub p4lang/behavioral-model - BEHAVIORAL MODEL (bmv2). https://github.com/p4lang/behavioral-model/blob/main/README.md Last accessed: 2024-05-24.

[34] Federico Parola, Sebastiano Miano, and Fulvio Risso. 2021. A Proof-of-Concept 5G Mobile Gateway with eBPF. In *Proceedings of the SIGCOMM '20 Poster and Demo Sessions* (Virtual event) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 68–69. https://doi.org/10.1145/3405837.3411395

[35] DPDK Project. 2024. DPDK. https://www.dpdk.org/ Last accessed: 2024-05-24.

[36] Pablo Salva-Garcia, Ruben Ricart-Sanchez, Enrique Chirivella-Perez, Qi Wang, and Jose M. Alcaraz-Calero. 2022. XDP-Based SmartNIC Hardware Performance Acceleration for Next-Generation Networks. *J. Netw. Syst. Manag.* 30, 4 (2022), 75. https://doi.org/10.1007/s10922-022-09687-z

[37] Dominik Scholz, Daniel Raumer, Paul Emmerich, Alexander Kurtz, Krzysztof Lesiak, and Georg Carle. 2018. Performance Implications of Packet Filtering with Linux eBPF. In *30th International Teletraffic Congress, ITC 2018, Vienna, Austria, September 3-7, 2018 - Volume 1*. IEEE, 209–217. https://doi.org/10.1109/ITC30.2018.00039

[38] David L. Tennenhouse and David J. Wetherall. 1996. Towards an Active Network Architecture. *SIGCOMM Comput. Commun. Rev.* 26, 2 (apr 1996), 5–17. https://doi.org/10.1145/231699.231701

[39] William Tu, Fabian Ruffy, and Mihai Budiu. 2018. P4C-XDP: Programming the Linux Kernel Forwarding Plane using P4. In *Linux Plumbers Conference*.

[40] William Tu, Joe Stringer, Yifeng Sun, and Yi-Hung Wei. 2018. Bringing the Power of eBPF to Open vSwitch. In *Linux Plumbers Conference*. 11.

[41] Péter Vörös, Dániel Horpácsi, Róbert Kitlei, Dániel Leskó, Máté Tejfel, and Sándor Laki. 2018. T4P4S: A Target-independent Compiler for Protocol-independent Packet Processors. In *IEEE 19th International Conference on High Performance Switching and Routing, HPSR 2018, Bucharest, Romania, June 18-20, 2018*. IEEE, 1–8.

[42] Jiarong Xing, Kuo-Feng Hsu, Matty Kadosh, Alan Lo, Yonatan Piasetzky, Arvind Krishnamurthy, and Ang Chen. 2022. Runtime Programmable Switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 651–665. https://www.usenix.org/conference/nsdi22/presentation/xing