Poster: The Case for Network Functions Decomposition

Farbod Shahinfar
Sharif University of Technology
fshahinfar@ce.sharif.edu

Sebastiano Miano
Queen Mary University of London
s.miano@qmul.ac.uk

Alireza Sanaee
Queen Mary University of London
a.sanaee@qmul.ac.uk

Giuseppe Siracusano
NEC Laboratories Europe
giuseppe.siracusano@neclab.eu

Roberto Bifulco
NEC Laboratories Europe
roberto.bifulco@neclab.eu

Gianni Antichi
Queen Mary University of London
g.antichi@qmul.ac.uk

ABSTRACT

This paper makes a case for writing unrestricted eBPF network functions which then get automatically decomposed between kernel and user-space.

CCS CONCEPTS

• **Networks → Programmable networks**.

KEYWORDS

Network Functions, Program Decomposition, eBPF

ACM Reference Format:
Farbod Shahinfar, Sebastiano Miano, Alireza Sanaee, Giuseppe Siracusano, Roberto Bifulco, and Gianni Antichi. 2021. Poster: The Case for Network Functions Decomposition. In *The 17th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '21), December 7–10, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3485983.3493349

1 INTRODUCTION

eBPF, a new technology for programming end-host systems, has recently attracted lot of attention from academia and industry [1, 3, 5]. This is because it allows running programs within the Linux kernel context without changing kernel source code or loading kernel modules. eBPF programs can be attached at various hook points, being the eXpress Data Path (XDP) the earliest in the packet receiving path. This is located in the NIC driver just after the interrupt processing and before any operations performed by the network stack, which are expensive [2]. Recognizing this opportunity, the research community has shown how it is possible to offload packet processing functionalities to XDP, thus removing the need for kernel processing and improving performance [3, 5].

Unfortunately, it is not possible to run *any packet processing function* in eBPF. This is because a program must be checked by the eBPF verifier before being loaded into the kernel so to ensure its behavior does not conflict with the kernel. Specifically, there are a number of restrictions that a programmer must take into account (see Table 1). Consequently, developers have so far resorted to specific workarounds to have their program accepted by the verifier. For example, BMC [3], a recently proposed XDP module for Key-Value store applications, can only accelerate small requests with key and value size less than 250 Bytes and 1000 Bytes, respectively. Also, BMC's functional logic has been manually separated into five smaller eBPF programs chained together to support a higher number of branches and instructions, since each of them are independently checked for safety.

Table 1: List of eBPF restrictions.

Criterion	Consequence
Complexity (# of Branches & Instructions)	Parsing complex protocols increases the number of states that the verifier has to check. When this number becomes too large, the program is rejected.
Loops	Bounded loops are possible in newer kernels (i.e., v5.3+) but their verification is subjected to the complexity of the entire program (see point above). For example, BMC [3] and bpf-iptables [5] limit the data they process (e.g., key-value size or number of supported rules) to pass the verifier checks.
Restricted functions & libraries	Only a limited number of kernel functions and C libraries can be used. Some network applications (e.g., IDS) may require complex operations on packets' content (e.g., regex matching) or custom data structures that are not available in eBPF.
Packet-driven processing	eBPF programs are only triggered on a per-packet basis. However, important data plane operations do not naturally fit into this programming model as network algorithms are naturally event-driven [4]. For example, this is the case when computing periodic tasks (i.e., measuring flow rates, resetting data structures).

In this poster, we advocate that a programmer *should not* worry about eBPF limitations and write the program logic without specific restrictions. We propose a compiler that takes as input *unrestricted eBPF code* and split it into two parts: one that can be accepted by the verifier and one that cannot. Our idea is to attach the former at the XDP hook point and run the latter at user space. The linking between the two is possible with AF_XDP, a new Linux socket type that allow XDP programs to redirect packets to a memory buffer in a user space application. Recent proposals have partially explored the idea of split-processing [7], in this poster, we propose that this shall be done *automatically* at program compile time. This approach brings two main benefits:

Expressiveness. As a first example, consider the case of a Memcached application where a loop over variable length data is required. If written naively, the program would be rejected by the verifier as the loop might result too large and exceed the maximum number of branches that the verifier can analyze. Here, instead of forcing the programmer to limit the key/value sizes to a specific value [3], the compiler shall automatically split the logic into two parts: one to be attached at the XDP layer with the maximum loop size supported, and the rest managed at user space to handle remaining cases. We call this *vertical function splitting*. As a second

CoNEXT '21, December 7–10, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-9098-9/21/12.
https://doi.org/10.1145/3485983.3493349

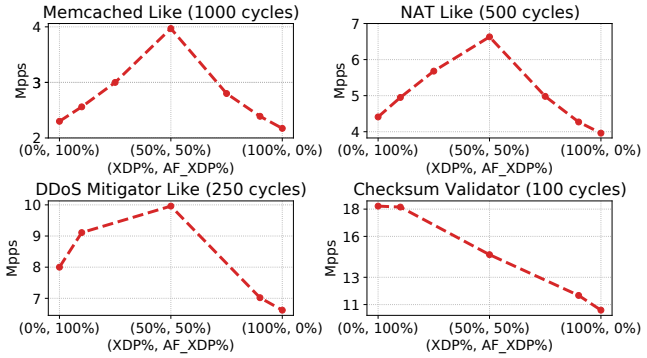

Figure 1: Throughput for different applications when the work is split between XDP and AF_XDP.

example, consider an Intrusion Detection System (IDS) that has to work on the entire packet. Here, the program might be rejected by the verifier for an excessive amount of branching. In this case, the compiler shall split the program logic into small pieces that alone could pass the verifier. Then, it shall chain them together automatically. We call this *horizontal function splitting*.

Performance. We performed a number of tests to better understand what are the performance implications when logic is split between XDP and user space. We connected two servers (2x10-core Intel Xeon Silver 4210R @2.40GHz) back-to-back, and we used one server to generate high-throughput 64B traffic, while in the other we evaluated the split program. We set up the server to consume a constant amount of CPU cycles per packet, representative for different applications. Figure 1 shows the achievable throughput of the server when its processing is split between XDP and user space (AF_XDP). As a baseline, we tested the cases where the processing is either entirely in XDP or at user space with AF_XDP. Generally, splitting the logic leads to better performance. This is because AF_XDP requires XDP to run; offloading work to the latter can improve the performance until XDP does not become the bottleneck, which is the case of the right part of the figures. For small tasks (~100 cycles), the cost of running the program at XDP level is higher, and AF_XDP only represents the best alternative.

2 OVERVIEW OF THE SOLUTION

In Figure 2, we show an overview of our solution. Unrestricted eBPF code is taken as input by our compiler which first performs a set of analysis passes (Static Analysis box in the Figure). This helps for building an understanding of the input code that will be used to split its logic in multiple mini-programs (Program Synthesis box). Here, we can use similar techniques that already proved to be successful for P4 targets [6, 8]. As discussed above, the splitting can happen either horizontally (by chaining together multiple XDP programs) or vertically (by disaggregating the function between kernel and user space). In some cases, only one option is possible: an unbounded loop cannot be represented with a set of bounded loops within multiple mini XDP programs. In others, both splitting mechanisms might be adopted. For example, in the presence of a big input program, dividing its functionality in smaller XDP modules might allow each of them to separately pass the eBPF verifier. In the next points we discuss opportunities and challenges in relation to network function decomposition.

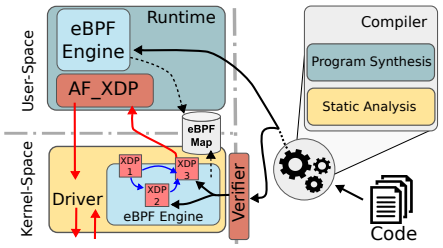

Figure 2: The compiler splits unrestricted eBPF code in a chain of XDP modules and a user space program.

Overheads. Decomposing a packet processing program in multiple mini-functions does not come for free. Naively, if the first function is in charge of parsing packets, the others in the chain should not redo this operation. This implies mini programs shall exchange also some metadata. As a consequence, function splitting increases data movement and this might impact the overall performance. In addition, the basic operation of jumping between programs (i.e., *tail call* in eBPF dialect) is expensive per se. A challenge is to find the sweet spot between number of mini programs that pass the verifier and overall packet-processing performance.

Metrics for splitting. Splitting the original code in what can be done at the XDP layer and what cannot is not the only option. There might be multiple metrics that can dictate this. For example, having early access to the packet with XDP might help with latency sensitive workloads. In contrast, using Single Instruction Multiple Data (SIMD) operations at user space can help for throughput intensive workloads. A challenge will be to define different type of splitting based on input metrics, which can be achieved by designing a performance prediction mechanism that would give the right hints on when the splitting is helpful and when it is not.

The need for network stack. The input program might require kernel intervention. The compiler shall be able to recognize this and avoid any type of vertical splitting as it might trigger multiple round-trips between kernel and user space. Only horizontal splitting shall be allowed and if not possible, the compiler shall reject the program. A challenge is designing new eBPF verbs that allow the user to provide hints to the compiler on the program's behavior. The compiler will then enhance the original program with additional instructions that will help distinguishing the packets requiring kernel intervention.

Acknowledgments. We thank the anonymous reviewers. This work is sponsored in part by the UK EPSRC project EP/T007206/1 and Facebook Networking System Award 2020.

REFERENCES

[1] 2021. Cilium. https://cilium.io/.
[2] Cai et al. 2021. Understanding host network stack overheads. In *SIGCOMM*. ACM.
[3] Ghigoff et al. 2021. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing.. In *NSDI*. USENIX.
[4] Ibanez et al. 2019. Event-Driven Packet Processing. In *HotNets*. ACM.
[5] Miano et al. 2019. Securing Linux with a Faster and Scalable Iptables. *ACM SIGCOMM Computer Communication Review* (2019).
[6] Soni et al. 2020. Composing Dataplane Programs with μP4. In *SIGCOMM*. ACM.
[7] Tu et al. 2021. revisiting the open vSwitch dataplane ten years later. In *SIGCOMM*. ACM.
[8] Zhang et al. 2020. Gallium: Automated Software Middlebox Offloading to Programmable Switches. In *SIGCOMM*. ACM.