Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

e^{2}BPF: an Evaluation of In-Kernel Data Processing with eBPF

Author: Giulia Frascaria (VU: 2608500, UvA: 11992069)

1st supervisor: Animesh Trivedi
2nd reader: Alexandru Iosup

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

February 10, 2021

"Don't Panic."

from The Hitchhiker's Guide to the Galaxy*, by Douglas Adams*

Abstract

The economy of Big Data is increasingly gaining momentum, and with the increasing needs for data processing datacenters are under pressure improve performance for a diverse set of applications. While technological improvements are raising the bar of performance, the modern storage technologies like Flash SSD and Optane outpace the network speed and introduce a bottleneck in data transfer known as data movement wall. For this reason, an increasing trend in the industry is to push computation closer to storage, by means of software and hardware accelerators. In this thesis we will explore eBPF, an in-kernel infrastructure to execute user-defined programs, as a technology to implement programmability for data processing which can serve as a mitigation to the data movement wall. We will present a prototype (available at `https://github.com/giuliafrascaria/ebpf-filter-reduce` to highlighting the potential and current limitations that prevent it from being an efficient solution, despite its widespread use in other areas like networking.

Acknowledgements

I am incredibly proud to present this thesis. Working on this project has kept me (arguably) sane during a pandemic that had me living the better part of 2020 within my 20-square-meters studio. Had I not had this project that I could dedicate my time to, I would have struggled to keep the focus and motivation.

I would like to thank my supervisor, Animesh Trivedi, for the incredible guidance that I was given during the whole process, as well as for the motivation to push through the most frustrating moments and hiccups.

Needless to say, I am forever grateful for the support of my parents and sister, that made it possible for me to be where I am today.

I definitely owe appreciation to all the friends that were patient enough to keep listening to me trying to explain my thesis project and consequent headaches, and probably lied to some degree about understanding what I was talking about.

Contents

List of Figures

List of Tables

1 Introduction

In the modern world and economy, the use of technology is now pervasive to every sector. Both private individuals as well as companies use computer systems to perform an increasing number of tasks. Services like social networks (e.g. Facebook, Instagram, Twitter), audio and video streaming platforms (e.g. Spotify, Youtube), online services for productivity and storage (e.g. Google Docs and Drive) are available to everyone. In addition to that, companies make heavy use of computer systems for business critical applications that are based on data processing or machine learning. All these modern services have a common denominator, they produce, consume and need to store significant amounts of data. This trend has been growing over the years(5) and goes under the name of **Big Data**.

With this ever-increasing amount of data comes the need for a suitable processing and storage infrastructure. This is a challenging situation, since ownership and maintenance of such infrastructure can have a prohibitive cost to most companies. As a solution towards cutting the cost of data processing and storage, the industry is turning to **cloud computing**(6). A few cloud providers like Amazon AWS, Microsoft Azure and Google Cloud offer pay-per-use infrastructure that everyone can rent and share with other tenants at a fraction of the cost, and use it in order to process and store their data.

While cloud computing is arguably one of the biggest disruptors in the modern economy, with a market capitalization that reached $233 billion in 2019(7), its success comes with the challenge of continuous innovation. The possibility to rent a shared infrastructure with a pay-per-use basis allows an ever-increasing number of services to be deployed online. In particular, companies in the fields of Big Data and Machine Leaning (ML) make extensive use of datacenter-scale infrastructure in order to run their business, performing business-critical workloads.

However, the reliance of Big Data and ML workloads on **massive amounts of data** poses many challenges for cloud providers. With more and more sectors and companies turning to cloud computing, datacenter providers are also faced with an increasing diversity of workloads that are specific to different industries. This creates a clash between the effort to provide general-purpose infrastructure that can fit all use cases, and that of delivering high performance, which can be better achieved through specialized tailor-made solutions.

The ever-increasing success of cloud computing pushed innovation in datacenter technologies. In particular, in order to deliver improved modularity and to support computation on big data, datacenters are now composed of compute and storage nodes with high speed networks connecting them (8). This implies the need to transfer significant amounts of data to and from storage nodes in order to perform computation (see figure 1.1). However, while data processing needs increase, we are witnessing at the same time the end of Moore's Law (9) with a slowdown of CPU technological improvements, as well as an introduction of new storage technologies whose throughput is outpacing that of networks (1). We can refer to this problem as a **data movement wall**, where the high performance of storage is shifting the bottleneck from the historically slow I/O to CPU processing and network transfers (10), (11) (12). As a result, solutions are being sought to solve or mitigate this for example by reducing the amount of data that is transferred within a single machine as well as across the network.

As a result, the industry is in need of fast processing for data while CPU performance improvements are stalling and transfer across networks introduces a bottleneck. Since data processing is needed to perform an increasingly diversified set of business-critical applications, one of the trends that show promising results to mitigate the issue of data movement is that of shipping computation to the storage nodes (13), and implementing specialization of computation through **programmability**. The issue is being tackled at multiple levels, both in hardware and software, especially by increasing diversification and customization with the introduction of specialized hardware, as well as allowing for user programmability all across the software stack.

Within this convergence of events, we focus and explore the topic of software-based programmability for data processing. In this thesis we address the issue of the data movement wall and present a novel software prototype to push user-defined code to storage nodes, as well as an evaluation of the underlying technology.

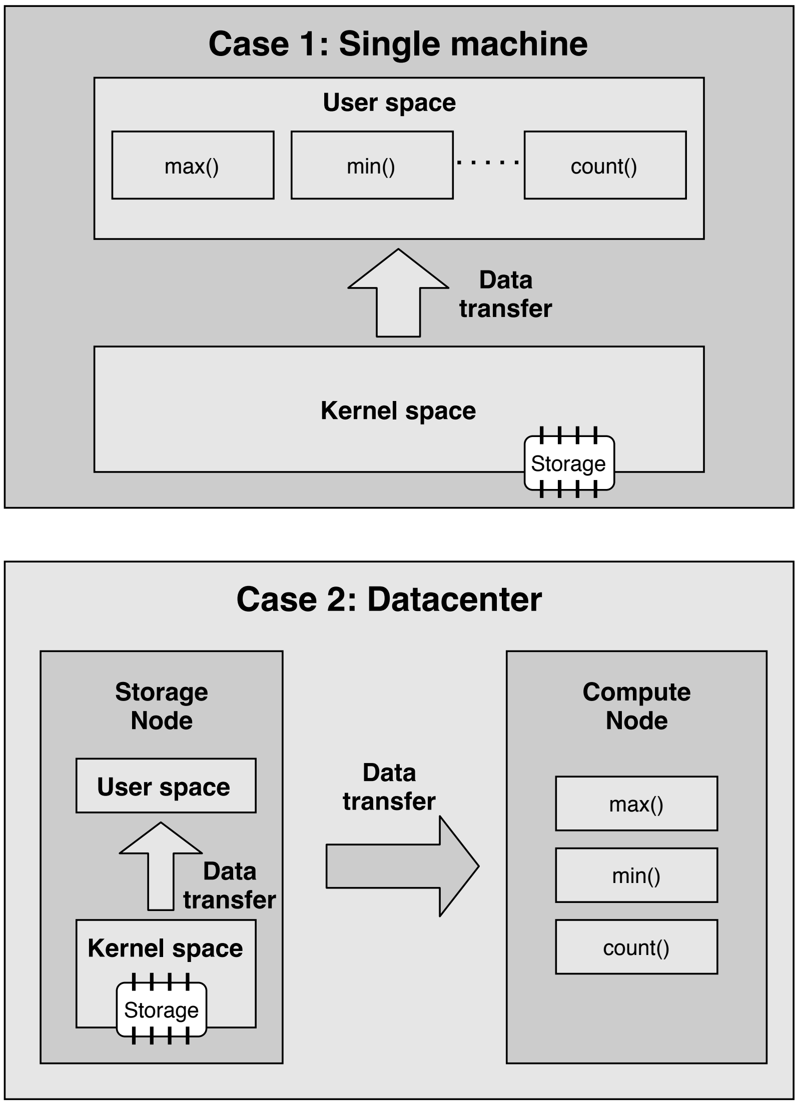


Figure 1.1: Traditional datacenter architecture and computation representation

1.1 A Case for Programmability

As the cost of data movement increases, solutions that can reduce the amount or size of data transfers are needed. Since data is transferred to and from storage nodes in order to use it for business-critical applications of every tenant, a possible solution is to ship computation on the storage nodes, an idea that is known in the research community as near-data processing (NDP) (13). This can be achieved by allowing tenants in datacenters to customize and program the datacenter infrastructure to optimize their workload. As we will see throughout the thesis, this is a challenging problem that can be tackled at different levels.

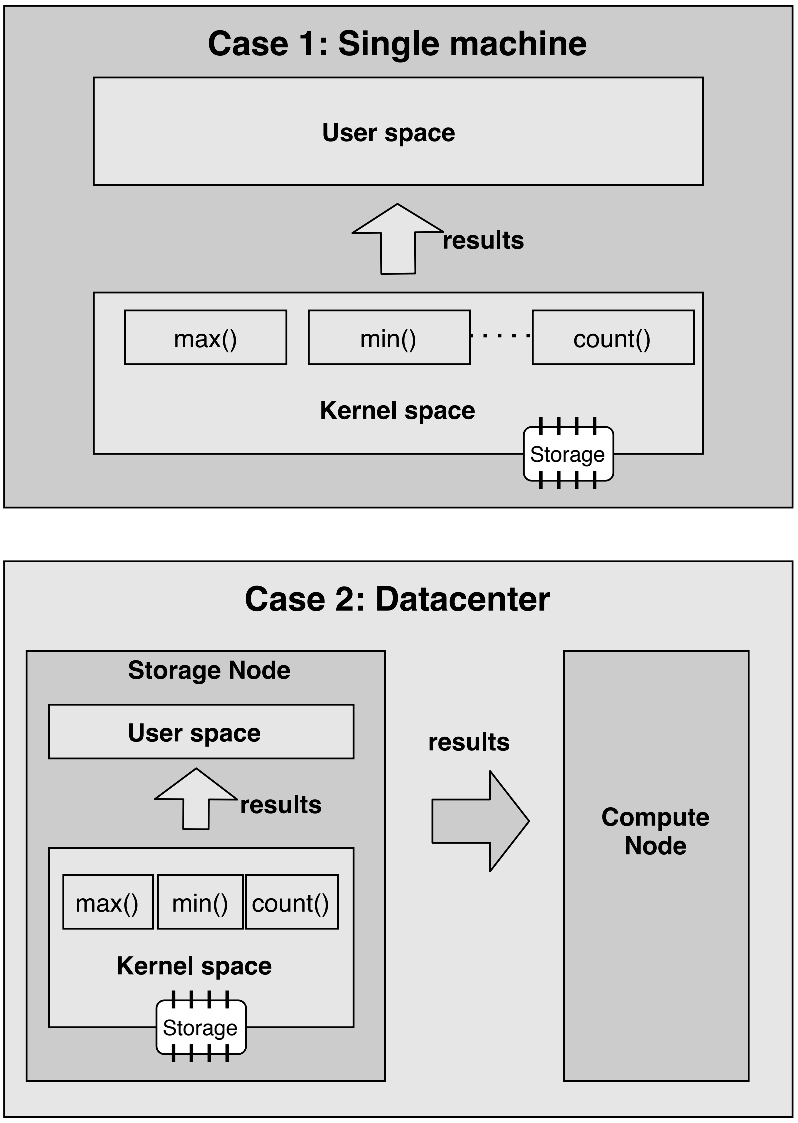


Figure 1.2: eBPF-based programmability in a datacenter architecture

As more industries started to make use of datacenters to run their applications, the variety of workloads that run in datacenters increased greatly. While it is true that with general-purpose computing infrastructures it is possible to run all kinds of workloads, the end of Moore's law and of CPU frequency scaling means that this will hardly be enough to accommodate the increased performance requirements of datacenter applications (14) (15). For this reason we are now witnessing a trend that goes in the direction opposite to general purpose computing, allowing for increased specialization. The goal is then to diversify the infrastructure and choose the optimal hardware and software for each of the workloads, rather than going for a one-size-fits-them-all approach.

The possibility to tailor hardware and software to optimize different workloads is known as **programmability**. Programmability can happen at all levels of the stack. For example, at the hardware level the datacenters can offer ASICs, FPGAs and GPUs to clients. Specialized and programmable hardware can greatly reduce the latency in computation when the fitting workload is executed.

However, while hardware acceleration has the potential to significantly improve performance, it does not come for free. Buying specialized hardware for datacenters is an upfront investment for cloud providers, and in order for it to be cost-effective it is important that the hardware can fit the workload requirements. Since workloads can change rapidly, this means that the hardware accelerators should be reconfigurable (16) like GPUs and FPGAs, thus implementing hardware programmability.

On the other side of the spectrum , it is possible to design optimizations of workloads using software-based programmability. In this way, users can implement their functionality in any programming language (e.g. C, Go, Rust) and have it executed on stoarge nodes. Compared to hardware programmability, this has the advantage of not requiring a change in the infrastructure, but comes with the cost of a reduced performance improvement due to the software overhead.

In between hardware and user applications however lies the operating system. In particular, datacenters usually run Linux (17). Within this software layer there are examples of programmability especially in the field of networking. These solutions use a software infrastructure called eBPF (extended Berkley Packet Filter) (3), that allows users to program small software extensions that run in a safe environment in the kernel level. With this infrastructure in place and already present in every Linux-based datacenter, it becomes relevant to investigate whether its potential for programmability can be leveraged in the storage stack to reduce the problem of data movement (see figure 1.2).

1.2 Problem Statement and Objective

As data movement becomes an increasingly relevant issue in datacenter performance, it is important to investigate all possible mitigations. As we highlighted in the previous section, programmability is a promising option that can be implemented both at hardware and software level. Within the software layer however it is possible to further distinguish between user applications and operating system. However, while user-level solutions can be implemented in any programming language, implementing programmability solutions in the operating system requires a solid infrastructure that can safeguard the computer system. While this reduces the options available for programmability, we see that eBPF appears to be a mature solution in fields like tracing, networking and security. As we will further analyse in the remainder of the thesis, eBPF (extended Berkley Packet Filter) allows users to push user-defined functionality to execute in kernel space in fields like networking and tracing. However the applicability to storage functions is still limited at best, although the eBPF infrastructure is under the spotlight due to its expressiveness, flexibility and potential for extensibility, as well as the ready availability within the Linux kernel (18).

Due to its versatility eBPF is already widely used in the networking stack to implement functionalities like packet inspection, modification and filtering, leading to applications like load balancers and DDoS prevention. In the context of data transfer and processing, we see a potential similarity and applicability of eBPF as a solution for programmability, in order to mitigate the data movement wall issue by reducing data transfers.

In this landscape, we then identify a research blank space for solutions that can yield to the reduction of the data movement across datacenters using eBPF. While userspace and hardware-based solutions have already been explored to some degree, the kernel space is still rather unexplored to this date. However, we believe that eBPF has the potential to fill the gap, due to its applicability for packet filtering within the networking software stack. The objective of this thesis is then to investigate this research space and answer the question whether eBPF is a suitable candidate for storage function programmability.

1.3 Research Questions

With this thesis project I aim at exploring the potential for eBPF to be used for programmable storage functionalities, and consequently conducting an in-depth study of the eBPF infrastructure. Since BPF documentation and studies are few, scattered, and lack of a systematic approach, it is a paramount objective of the thesis is to provide an in-depth study and breakdown of the performance impact of this technology and infrastructure.

We decompose the problem of evaluating the use of eBPF for storage programmability in the following research questions:

- (**RQ1**) *What is the performance impact of the eBPF infrastructure?.*
  Although eBPF has been available for nearly 3 decades (3), its complexity, lack of documentation and complex infrastructure make it difficult to evaluate the performance impact. However, it is crucial to understand the footprint of eBPF in order to

estimate its suitability for programmability of storage functions, especially in high-performance scenarios.

- (**RQ2**) *How to repurpose the eBPF toolchain and infrastructure in order to support programmable storage functions?*
  eBPF was not initially designed for the storage stack, being mainly targeted to networking instead. Despite its proven versatility in that field, it is necessary to understand to what degree the current infrastructure supports storage functions. In particular, it is crucial to assess whether it has the necessary functionality to access and process data, as it does within the networking stack.

- (**RQ3**) *What are the parameters to determine which functionalities should be offloaded to eBPF within the storage stack?*
  While eBPF is generally considered a high-performance infrastructure, its verification and compilation still add an overhead to the computation. After quantifying the impact of the eBPF infrastructure it is then crucial to understand which parameters can influence the decision to move part of the computation in eBPF, and what can yield to the maximum gain in terms of latency and reduction of data transfer

All these questions will be tacked throughout my research in different steps, using a variety of research methods.

1.4 Research Method

This thesis project is an experimental systems research. In order to answer the research questions, I will carry an in-depth evaluation of the eBPF kernel infrastructure (**RQ1**). In order to test the applicability to my use case, I will build a prototype of the system (**RQ2**), and design and perform experiments to validate the result (**RQ2** and **RQ3**).

The research was carried out following methodologies in computer systems research. A comprehensive survey of related literature and work was completed before starting the design process (**M1** quantitative research (19) (20)). The design and development of the software artifact followed an iterative and incremental approach to build the prototype (**M2** design, abstraction, prototyping (21) (22) (23)).

The evaluation of the eBPF infrastructure (**RQ1**) as well as the design and implementation of the prototype (**RQ2**) were carried out with methods of experimental research (**M3** (24) (25) (26)) and within the principles of open science (**M4** (27) (28) (29) (30)). All the software and tools that were used are open source, as well as the source code of the prototype and the experiments.

1.5 Main Contributions

The contributions in this thesis project are in three areas: (1) conceptual, (2) experimental contributions and (3) artifacts. The contributions of this work are the following:

- (Conceptual) A design for an eBPF-based data processing framework in the storage stack
- (Conceptual) A detailed guide on the eBPF infrastructure design, usage and extension
- (Experimental) A characterization and breakdown of the overhead of eBPF instrumentation and of storage function instrumentation with eBPF
- (Artifact) A prototype for user-defined storage function programmability through eBPF (available at `https://github.com/giuliafrascaria/ebpf-filter-reduce`)
- (Dissemination) Presentations on the topic of storage programmability with eBPF
  - Open Source Summit Europe 2020: *BPF Tales: or Why did I Recompile the Kernel just to Average Numbers?* (31)
  - eBPF Summit 2020: *Can eBPF Save us from the Data Deluge? A Case for File Filtering in eBPF* (32)

1.6 Reading Guide

This chapter gave a high-level overview of the problem of software programmability and eBPF. In the next chapter I will go in-depth about background knowledge for the topics of modern storage technologies and programmability. in Chapter 3 there will be a thorough analysis and guide to navigate the eBPF infrastructure, as well as its applicability to storage. Chapters 4 and 5 will present the design and implementation of the prototype of a programmable filter-reduce framework using eBPF, and chapter 6 will report on the evaluation. Finally, chapters 7 will put the work in context within the related work, and chapter 8 will expand on the conclusions and future work.

2

Background

In this chapter we present an overview of the topic of programmability, with a focus on its motivations and context. The research problem that this research tackles stems from a convergence of events that entail changes in the datacenter architecture towards disaggregated structures, the shift in the computing landscape from general purpose to specialized infrastructure, the modernization of storage technologies and the Big Data trend and the issue of the data movement wall.

2.1 The Data Movement Wall

Datacenter architectures and organizations evolved throughout the years. Initially all compute nodes used to be bulky machines that could offer both compute and storage resources, so the data and computation resided in the same node and were not heavily reliant on network transfers in order to perform computation. The latest trend in cloud providers however is that of disaggregation of compute and storage nodes, so that not the main cloud provider offer specialized infrastructure for storage-only infrastructure, (e.g. Amazon S3, Azure) (33) (8) (34). This accommodates the requirement of elasticity of resources, allowing fine-grained billing only according to the user's necessity, but it comes at the clear cost of implying data transfer from storage to compute nodes if that is needed for computation.

The need for fast data transfer translated to significant efforts in deploying increasingly more performant infrastructure, going anywhere from the CPUs to networks, storage and interconnect technologies. For example, a lot of attention is dedicated to the PCIe interconnection standards (35). Although datacenter nodes are connected via high-throughput connections (e.g. the current de-facto standard of the 16-lane PCIe, offering 16GB/s bandwidth), there is still a non-negligible overhead introduced by the different throughput of storage nodes and interconnects like the PCIe interfaces that are used within datacenters and machines (1) (see figure 2.1). For this reason, while datacenter hardware is generally improving, the difference in the speed at which different technologies are rolled out, standardized and adopted in datacenters can lead to the introduction of new bottlenecks like that of the data movement wall. This problem introduces a fundamental issue in a computer organization that still follows the traditional Von Neumann architecture featuring a CPU, memory and I/O interfaces.

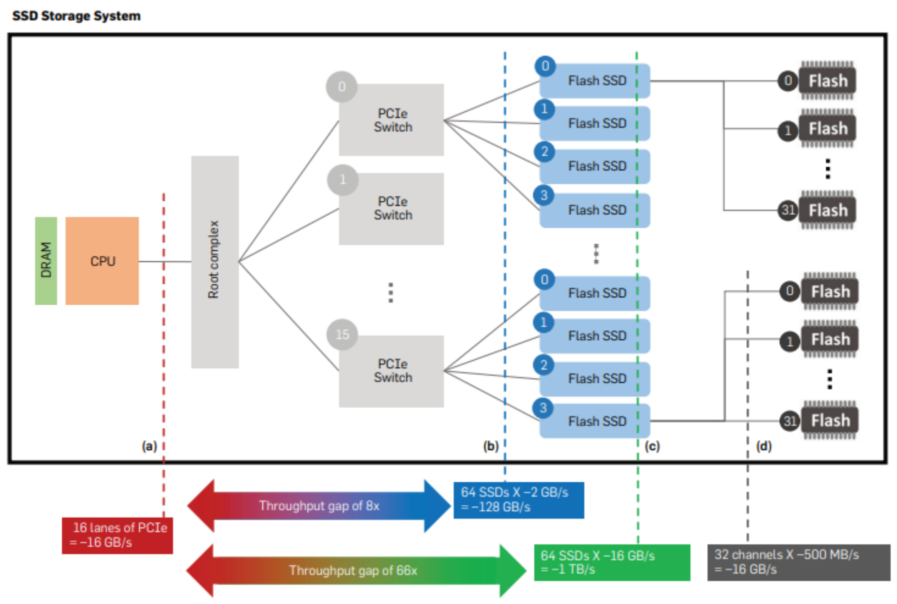

SSD Storage System
DRAM
CPU
Root complex
PCIe Switch
Flash SSD
Flash
(a)
(b)
(c)
(d)
16 lanes of PCIe = ~16 GB/s
Throughput gap of 8x
64 SSDs X ~2 GB/s = ~128 GB/s
Throughput gap of 66x
64 SSDs X ~16 GB/s = ~1 TB/s
32 channels X ~500 MB/s = ~16 GB/s

Figure 2.1: Conventional storage server architecture with PCIe interfaces and SSDs. Figure is taken from (1)

2.2 The Curse of General Purpose

While datacenters started and flourished by providing a general purpose computing infrastructure that provided CPUs, the slowdown of Moore's law (9), when we shift the focus to performance we realize that specialized solutions are needed in order to overcome this issue while still being able to provide ever-increasing performance. This led over time to the increasing availability of special-purpose infrastructure for different kinds of workload, from GPU computation for highly parallel computation to the use of Smart NICs for specialized network functions.

The flourishing of the cloud computing industry is also represented in the interest from both academia and industry to further research on how to evolve the infrastructure to deliver improved performance to the end users. Cloud computing enables a pay-per-use billing option for computing infrastructures that allows to deploy applications at a convenient cost for tenants, since they do not have the responsibility for the maintenance and deployment of the datacenter infrastructure. At the same time, it allows the owners of the datacenters to multiplex resources among different users and increase their revenue by increasing the utilization of the resources (36). This win-win situation caused an avalanche effect on the kinds and scales of cloud-based applications.

With the availability of datacenter infrastructure that can be shared among tenants with a pay-per-use basis, a lot of industries flourished and gained momentum. In particular, use cases that rely on heavy amounts of data are now enabled by the availability of affordable storage technologies and high-performance computing infrastructures. This is manifested in the pivotal role of Big Data, Machine Learning and generally data-intensive computation and data analytics. This trend further fostered innovation, leading to a lot of research happening in the datacenter scenario to find ways to share resources efficiently, reliably and in a safe way. As these core properties were increasingly obtained, a big part of the focus shifted to ways to improve performance and respond to the latest trends in the industry of Big Data and data processing. Many aspects of performance can be addressed and tackled for improvement. However, given the fundamental reliance on computations over massive amounts of data, one of the clear targets for research lies in identifying solutions that can optimize data movement.

This thesis focuses on the problem of efficient data movement across datacenters, and aims at minimizing it by pushing computation closer to storage. I will present a framework that can be used to push user-defined functionality to remote storage nodes, allowing to reduce data transfer by performing computation on the storage nodes, rather than after a costly transfer across the datacenter network.

2.3 Storage Technology Innovations

While every component of datacenters went through significant improvements, from CPUs to networks, one of the most rapid and disruptive improvements happened in the storage stack with the revolution introduced by new storage technologies like Flash SSD and Optane memory (37). These new persistent memory technologies improve the throughput and access latencies by orders of magnitude compared to the HDD, their predecessor (see

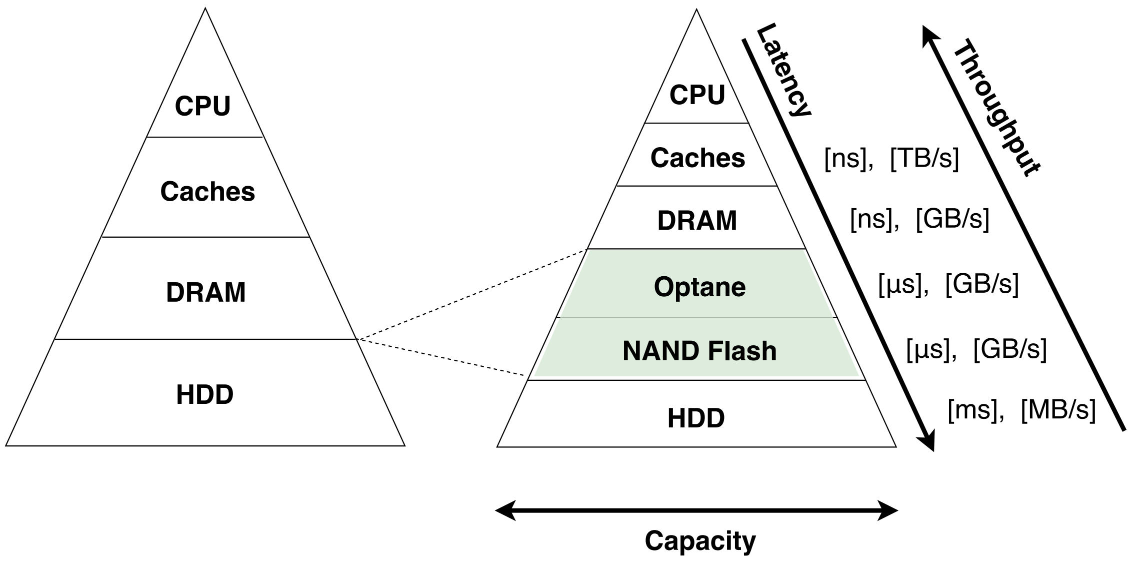


Figure 2.2: Traditional and modern memory latency and throughput (2)

figure 2.2). More details on the evolution of the storage technologies and storage stack can be found in the survey (38).

The great and disproportionate improvement of the new storage technologies when compared with the speed of new standardizations of the PCIe interconnect, as well as the stalling CPUs, introduces an issue especially within the Big Data workloads where operations rely on huge quantities of data to be transferred across the network. For this reason, a trend is gaining popularity. With programmability, part of the Big Data functionalities can be offloaded by the users to be executed closer to the data, on storage.

2.4 Towards Programmability

Modern computer systems put a strong stress on performance all across the board. The push towards an ever-increasing performance gain was rendered necessary by the increasing number of datacenter use cases. With the explosion of datacenter-based heavy computation in the field of Big Data, computer systems are put through a significant stress and the resource requirements only keep increasing. these kinds of workload require both a wide amount of storage, since they operate on massive amounts of data, as well as a high computational capability to deliver results as fast as possible.

However, it's been a while now since the Moore's law stopped accommodating CPU speed increases that can match the requirements. On the other side of the spectrum, storage is instead improving its performance significantly and went from the high latency HDD to the relatively fast NAND Flash memory and Intel Optane memory (37) (1). Fast storage and bottlenecked CPUs aggravate by the issue of data movement with the disconnect between the rapidly-evolving and high-throughput of modern storage media and the slower progress of interconnects and CPUs.

The trend to disaggregate storage resources, paired with the issue of the data movement wall and the improved storage performance all gave motive to the research in the field of programmability for storage functions. The programmability trend is already being widely explored in other computer systems fields like GPU and networks, and it is viewed as a trend for the future of computer architecture (39). Programmability allows to steer away from generale purpose CPUs and their programming model, which is outdated for the current trend of high performance computing that requires specialized hardware and is not satisfied with a one size fits them all solution.

In addition to hardware-based programmability we have software programmability, through which it is possible to use programming languages to define functionality that can be offloaded closer to where the data resides in storage, following the trend of near-data processing (13). Software programmability can be approached using many different programming languages and infrastructures, most of them being executed in user space within a computer system (e.g. using Python, Rust). However, executing code at the operating system level could potentially offer gains since it removes a software layer and helps to bring the computation closer to the data. However, kernel space programmability has the caveat of potentially compromising the computer system if not implemented correctly. For this reason, support for kernel-level programmability is currently limited within the storage stack. However, in the field of networking there is an infrastructure that has been successfully used for nearly 3 decades, called eBPF (extended Berkley Packet Filter), that allows users to push packet filtering functionality to be executed in the Linux kernel, yielding to a gain in performance compared to userspace processing (3). With this infrastructure still relatively unexplored, an interesting research challenge opens as to whether this infrastructure can offer good support for storage function programmability in the same way it does for networking.

2.5 Summary

We are currently witnessing a striking convergence of events in the computing industry. The explosion of Big Data puts a strain on cloud provider to improve their the datacenter resources. As a consequence, technological innovation spans throughout the whole infrastructure. However, the speed of innovation and adoption of different technologies is different. For example, PCIe standards are rolled out more slowly than the modern storage technologies like Flash and Optane memory. The disconnect between storage throughput and data movement speed, coupled with the increasing stall of CPU frequency scaling due to the end of Moore's law, pushes towards a trend to reduce data movement across the machines and datacenter networks by bringing computation closer to the data, with near-data processing. The need to allow multiple users to push their own functionality calls for programmability within the near-data processing landscape, and this can be implemented at different levels and through many approaches, broadly grouped into hardware and software programmability.

With this thesis we aim at exploring the kernel-level flavor of software programmability using eBPF (extended Berkley Packet Filter), in order to assess its suitability for the storage stack compared to the networking stack for which it has been used thus far. In the next chapter we will introduce and explain more details regarding eBPF.

3

The eBPF Infrastructure

In the previous sections we introduced the problem of the data movement wall, as well as the potential for software programmability to be a solution to this. In this section, we will expand on eBPF and examine its potential as a technology to achieve software programmability in the storage stack.

3.1 History of BPF

BPF, or BSD Packet Filter(3), was originally introduced in 1993 as a novel way to filter packets in Unix systems. The authors being from the Lawrence Berkley Laboratory, the acronym later came to be known as the Berkley Packet Filter instead. The original intent of BPF was for it to be a tool for networking filters, moving part of the packet processing from userspace to kernel space in order to avoid unnecessary data copies for unwanted packets. BPF offers a register-based virtual machine that operates inside the kernel and performs packet filtering based on a state machine (figure 3.1). The BPF virtual machine has a specialized instruction that allows to program operations that can be executed in the kernel space. In this way, BPF allows to reduce the number of CPU instructions when compared to userspace filters. BPF rapidly gained momentum thanks to this performance boost, and a wide range of networking applications like tcpdump (40) and arpwatch (41) adopted it and their engine.

The instruction set for BPF is purposefully limited, in order to limit its applicability to simple filter functionality. In the figure 3.2 there is a summary of the original BPF instruction sets. The bytecode of BPF has a standard instruction length, 32-bit registers and a frame pointer, as well as support for load, ALU operations and jump operations. This instruction set can be used to write small packet filters like the snippet in figure 3.3

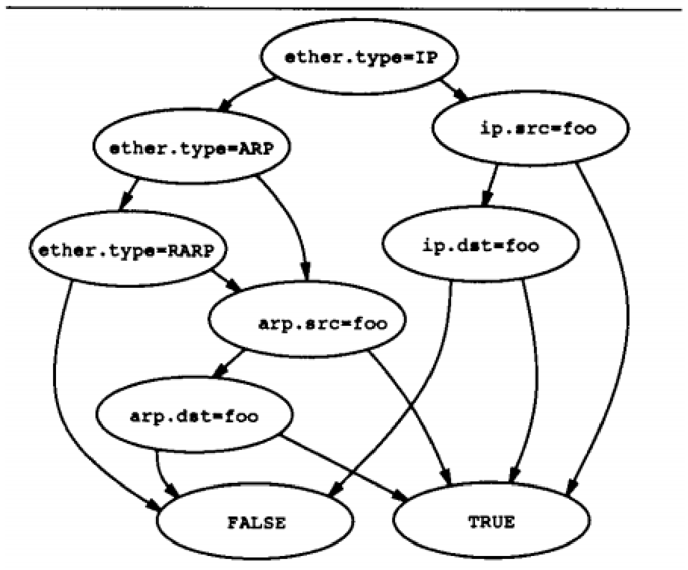
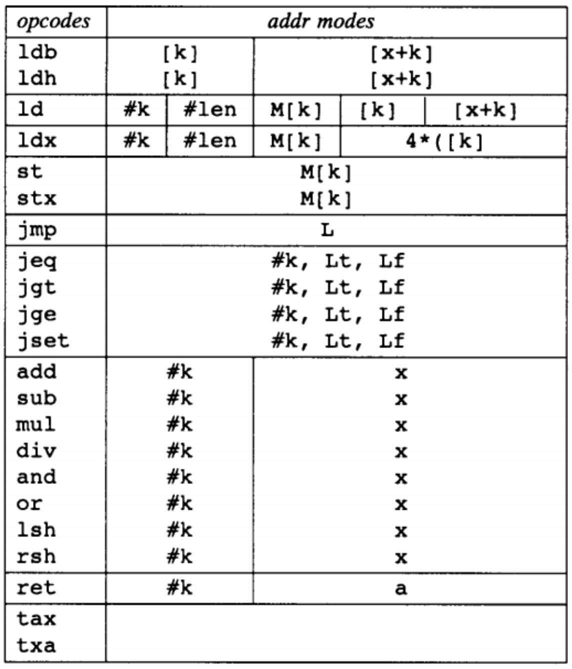


Figure 3.1: An example state machine for BPF-based filtering. Figure is taken from (3)

as well as an example program that performs a filter. Initially, BPF programs had to be written in this assembly-like language and were limited to 4096 lines of bytecode length. Clearly, this initial version of BPF was meant for limited use cases and could only be effectively used by experts that could write in a low level, assembly-like language. As BPF continued to show good performance, after it was included in the standard Linux kernel in version 2.5 the functionalities of BPF were greatly expanded.

After the initial development, BPF was greatly expanded with the introduction of an ever-increasing support in the Linux kernel, starting from the just-in-time compilation that was added in the kernel 3.0 (42). The development work around BPF hasn't stopped since, since the expressivity and power of the BPF interpreter make it a suitable candidate to be considered a universal in-kernel virtual machine (43). A complete summary of BPF instructions can be found in the Linux source code documentation(44). With eBPF the number of registers in the virtual machine was increased and ported to 64-bit architecture, the maximum length of the eBPF programs was brought to 1 million lines of code, and there is added support for the use of helper functions, which execute functionality which wouldn't otherwise be available within the eBPF instruction set.

The increased versatility of the infrastructure led to the name change from BPF to eBPF, Extended Berkley Packet Filter. Notably, the instruction set was ported to 64-bit registers and support was added for multiprocessor systems. The types of instructions were also increased, and a system call was introduced in the Linux kernel to facilitate the usage



opcodes	*addr modes*				
ldb	[k]		[x+k]		
ldh	[k]		[x+k]		
ld	#k	#len	M[k]	[k]	[x+k]
ldx	#k	#len	M[k]	4*([k]	
st	M[k]				
stx	M[k]				
jmp	L				
jeq	#k, Lt, Lf				
jgt	#k, Lt, Lf				
jge	#k, Lt, Lf				
jset	#k, Lt, Lf				
add	#k		x		
sub	#k		x		
mul	#k		x		
div	#k		x		
and	#k		x		
or	#k		x		
lsh	#k		x		
rsh	#k		x		
ret	#k		a		
tax					
txa					

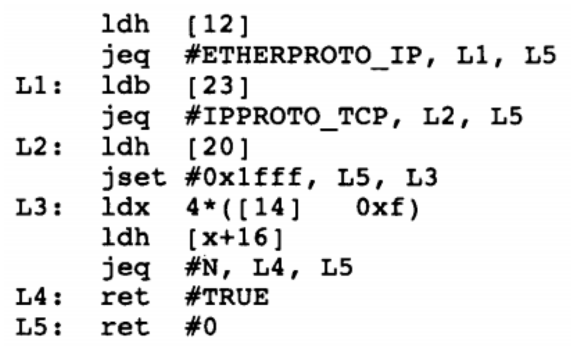
Figure 3.2: The original BPF instruction set, figure is taken from (3)

(45). With this increased support for complex programs, it became vital to have a safety guarantee that in-kernel execution would not compromise the system in case of errors. This is perhaps the most important key to the success of eBPF, that features full verification through symbolic execution of programs before their execution (46). The final arhitecture of the eBPF infrastructure is reported in figure 3.4.

Extensions to BPF followed steadily, and were mostly by Alexei Starovoitov (47) who led the development from BPF to eBPF and is currently the maintainer of the BPF subsystem in the Linux kernel. For example, the use cases of eBPF were initially expanded from networking only to tracing (48). After this happened, eBPF became a staple for kernel tracing. Functionalities continued to be added, the maximum program length was increased in order to allow for more complexity, and new program types are continuously added.

A key to the success of eBPF, in addition to its efficiency, is the fact that there is now an increasingly varied toolchain that is available to develop and deploy eBPF programs. Support for eBPF development is now widespread and goes from GCC, clang/LLVM up until development frameworks like the C/C++ library libbpf (49) (figure 3.5), the Python-based framework BCC (50) (figure 3.6) as well as the high-level language bpftrace (51) (figure 3.7) and Golang libraries (52) (figure 3.8).

With the wide variety of frameworks and toolchains available to develop eBPF programs,



```
      ldh  [12]
      jeq  #ETHERPROTO_IP, L1, L5
L1:   ldb  [23]
      jeq  #IPPROTO_TCP, L2, L5
L2:   ldh  [20]
      jset #0x1fff, L5, L3
L3:   ldx  4*([14]   0xf)
      ldh  [x+16]
      jeq  #N, L4, L5
L4:   ret  #TRUE
L5:   ret  #0
```

Figure 3.3: a BPF code snippet for packet filtering (3)

it is important to understand the details, possibilities and limitations of such a powerful infrastructure.

3.2 Programming eBPF Extensions

As introduced in the previous section, eBPF is an instruction set that can be executed in an in-kernel virtual machine after being verified through symbolic execution, thus giving the guarantee that the operating system will not crash due to user functionality being pushed in BPF code. With the flourishing of development efforts in the eBPF community, the expressivity of eBPF greatly increased and now allows to serve many use cases. However, the main ones still remain networking, tracing and observability as well as security.

As the different use cases have a diverse set of needs, they are separated in a set of eBPF program types that are handled separately by the verifier and execute based on different triggers in the kernel. A list of the updated eBPF program types is in the BPF manual page for eBPF and eBPF helpers (53) (54) but the thesis will focus mostly on the observability and tracing program types, **kprobe** and **tracepoint** that allow to inspect close to any function that is called within the kernel, and to have access to the parameters of the function call.

Within the maximum instruction limit, which is currently set to 1 million BPF bytecode instructions, the eBPF virtual machine supports standard procedural programming in all the program types. However, eBPF is not a Turing-complete instruction set and does not support infinite loops, as well as having a strict limit on the maximum stack size which is set to 512 bytes. Within these limitations, arithmetic and logic operations as well as bytewise operations that allow to operate on single `char` variables are allowed.

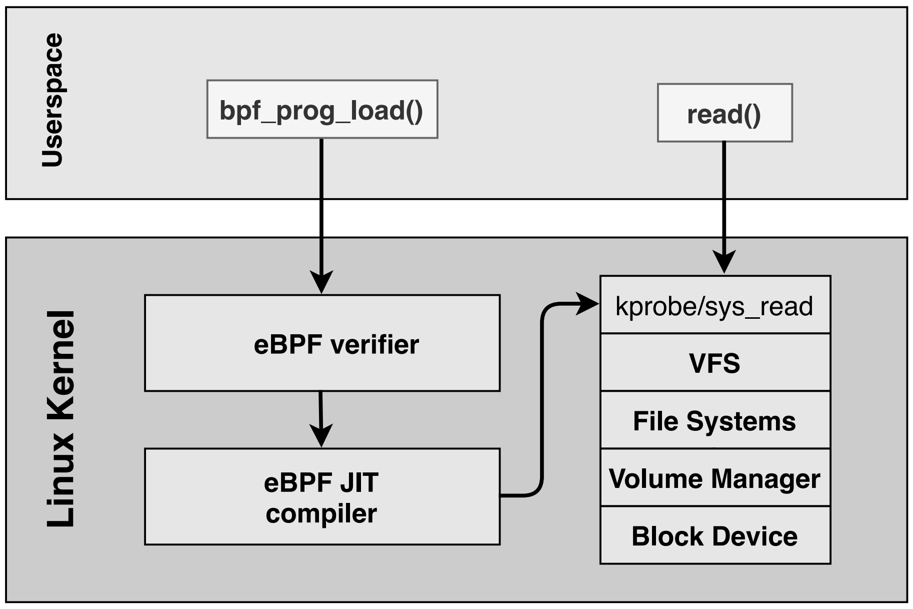


Figure 3.4: The native eBPF infrastructure

Each eBPF program is run within a context (ctx) that gives visibility on data that the BPF extension can operate on, for example function call parameters. The context can be accessed within the extension, but depending on the program type the access might need to be mediated through helper functions (e.g in the kprobe program type). The context can be accessed through macros that expose single parameters, with structure `PT_REGS_PARM1(ctx)`.

In addition to the access to parameters of the function call, eBPF extensions can also communicate with the userspace process through shared maps that can be read and written from both sides. The maps act as a key-value store and can significantly expand the memory space that is available to eBPF programs beyond the 512 bytes on the stack.

A lot of additional functionalities can be provided by the helper functions that can for example read and write the parameters from context, access the shared maps between eBPF and the userspace process, access process-related information like the PID or timestamp information.

The programs written in eBPF will then be verified by the verifier, which will enforce strict security and isolation allowing the users to catch a wide variety of bugs and unsafe behaviors at load time, before the eBPF program is executed in the kernel.

An eBPF extension that passes the verification process will execute (in the case of kprobes) every time the instrumented function is called. This allows to inject code at

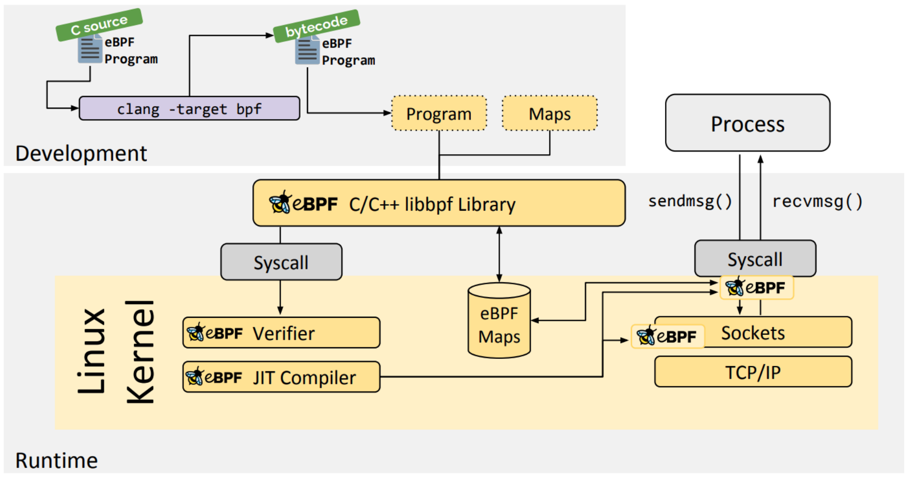


Figure 3.5: The libbpf toolchain. Figure taken from (4)

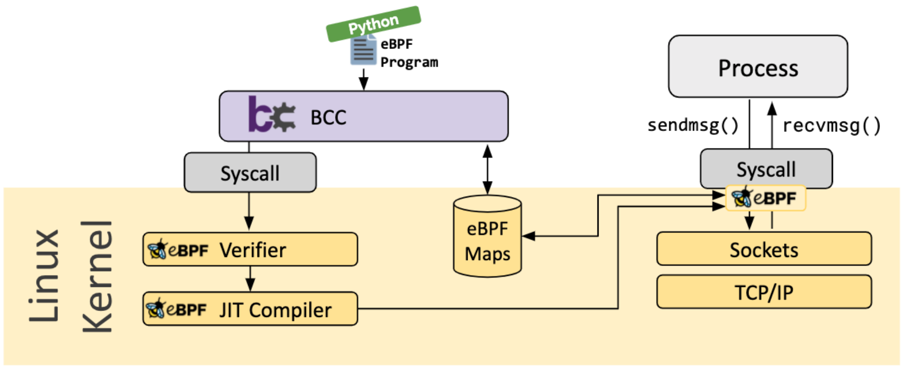


Figure 3.6: The BCC toolchain. Figure taken from (4)

runtime during the execution, and gives an instrument to access the parameters of the function call and operate on them within the limits of the eBPF infrastructure.

3.3 The Verifier

As stated by Dave Miller, "the only thing between us and extreme peril" when programming with eBPF is the verifier (55) (56). This is indeed true, since eBPF allows users to inject their own code in the kernel and it wouldn't be feasible if we didn't have a guarantee that allowed to safely assume that this cannot crash the system, cause significant slowdown or corrupt user and kernel data structures. The verifier (57) acts as a safety net for this, since its thorough verification and symbolic execution allow to catch every malicious or unsafe memory access, data corruption as well as infinite loop in eBPF programs.

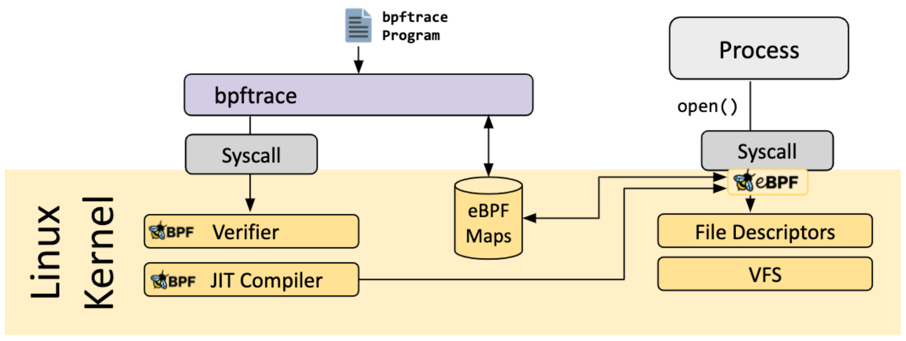


Figure 3.7: The bpftrace toolchain. Figure taken from (4)

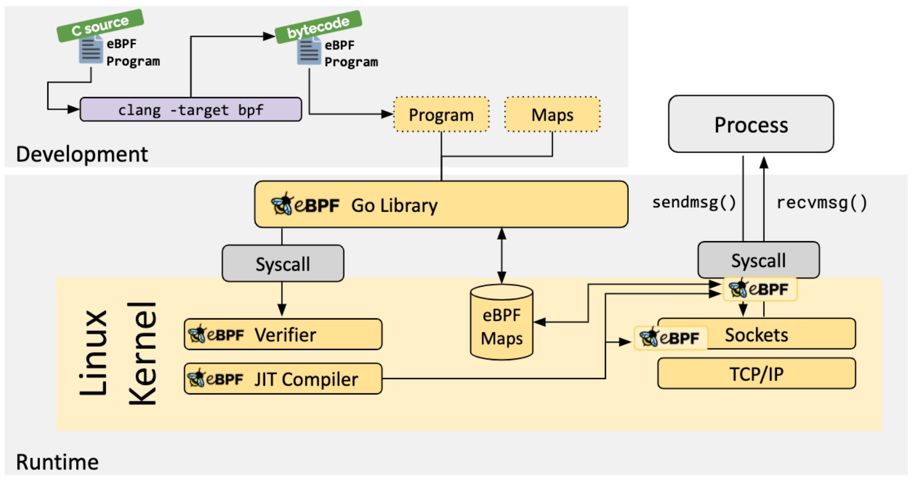


Figure 3.8: The Golang toolchain. Figure taken from (4)

Since eBPF operates in the kernel, one of the paramount priorities for its developers and maintainers is to ensure that the execution of eBPF programs does not indefinitely stall the kernel. For this reason, eBPF is not a Turing-complete language, does not allow unbounded, infinite loops and instead has a maximum number of instruction lines, that is currently set to 1 million BPF instructions.

The verification process in the verifier happens in two separate steps (58). The first one is a static verification of the number of instructions, as well as the presence of infinite loops and access to helper functions in the kernel. At this step the verifier performs a depth first search in the program control flow graph to ensure that the execution is a direct acyclic graph (57). At this stage the verifier can identify errors like the following *(ndr %d indicated the instruction number)*:

- back-edge from insn %d to %d

- unreachable insn %d
- BPF program is too large. Processed %d insn

The verification process then goes on with a second pass that is more time consuming, where symbolic execution is performed on all the possible code paths of the eBPF extension in order to eliminate errors that originate from improper data access or error condition checking as well as improper use of helper functions. Once both the steps are successfully passed the program can complete loading, and will be triggered in the appropriate conditions as specified by the programmer and the program type. This step identifies errors in the safety of operations and can detect unsafe memory accesses, memory corruptions, prohibited operations

- Reg pointer arithmetic prohibited
- frame pointer is read only: cannot write into reg
- R0 leaks addr as return value *(ndr: R0 is the register that stores the return address)*

For every program type, the verifier has metadata that indicates the kind of visibility and access rights over function parameters, as well as the helpers that are allowed within the eBPF different eBPF programs. The use of different program types allows to keep privileges at a minimum for each of the program types, in order to make the execution safer.

3.4 Debugging eBPF

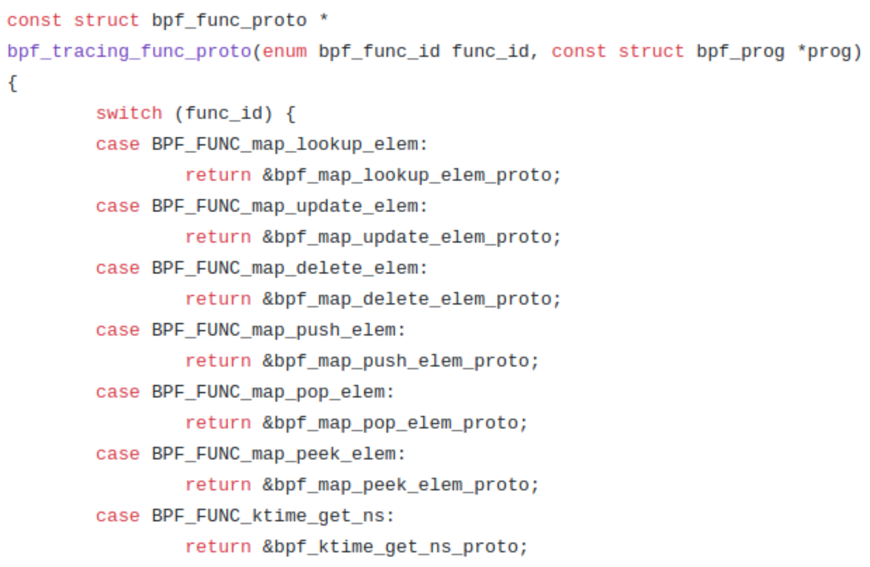
While programming eBPF, it is very frequent to encounter behaviors that are not expected. While most of them will be catched by the verifier, it is possible that some are not errors that are due to unsafe programming and rather just stem from a misjudjement in the design of the extension. For this reason, there are a variety of tools that can help to debug programs at different stages of their lifetime (59), (60).

A first step in the debugging workflow is manual inspection of BPF objects of eBPF assembly after the compilation process. Both the eBPF object and bytecode can be generated by clang or dumped through llvm-objdump. After the objects are loaded debugging can still happen through a set of tools. libbpf offers the option to test programs, and with bpftool it is possible to test run the programs with artificial input. Debugging at runtime is also possible by means of the eBPF helper `bpf_trace_printk` or writing debug information that the user can access through a special map, `bpf_perf_event_array`.

3.5 Extending eBPF

While the eBPF infrastructure offers significant support for a variety of operations, it is important to be aware that it might not always have the required functionality available. If this is the case, one needs to consider the possibility of extending eBPF with new features. This requires work within the Linux kernel and is not a trivial process, however in this section we will give suggestions about initial things that can be used to extend functionality. These indications are based on the Linux kernel 5.7.

Exposing Symbols While the kprobe program type for eBPF offers observability of virtually every function in the kernel, it is still necessary that the function symbol is visible (exported). Not all function signatures are exposed withing the kernel. In case there is a need to observe one of the functions that are not visible by default, it is necessary to modify the kernel source code and explicitly export the symbol with the macro `EXPORT_SYMBOL()` and recompile the kernel.

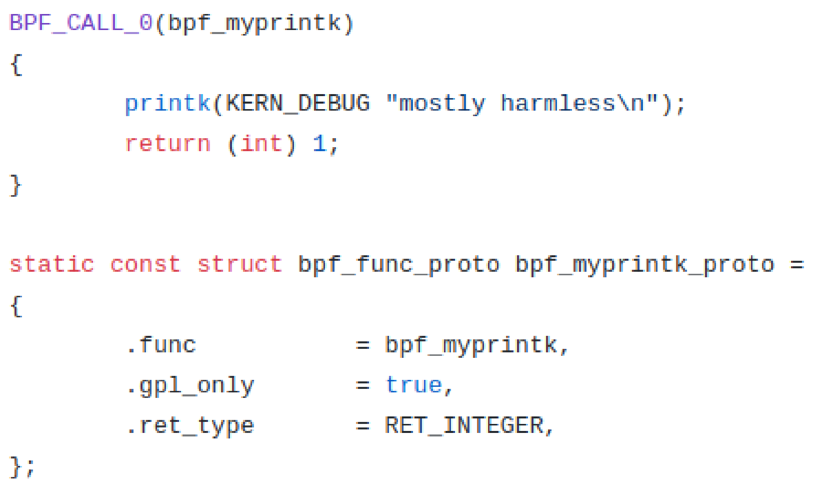


```
const struct bpf_func_proto *
bpf_tracing_func_proto(enum bpf_func_id func_id, const struct bpf_prog *prog)
{
        switch (func_id) {
        case BPF_FUNC_map_lookup_elem:
                return &bpf_map_lookup_elem_proto;
        case BPF_FUNC_map_update_elem:
                return &bpf_map_update_elem_proto;
        case BPF_FUNC_map_delete_elem:
                return &bpf_map_delete_elem_proto;
        case BPF_FUNC_map_push_elem:
                return &bpf_map_push_elem_proto;
        case BPF_FUNC_map_pop_elem:
                return &bpf_map_pop_elem_proto;
        case BPF_FUNC_map_peek_elem:
                return &bpf_map_peek_elem_proto;
        case BPF_FUNC_ktime_get_ns:
                return &bpf_ktime_get_ns_proto;
```

Figure 3.9: Helper function list for kprobe program type, in file /kernel/trace/bpf_trace.c

Enabling Existing Helper Functions Every eBPF program type comes with a set of enabled helper functions that can be used to perform specific functionalities that are not supported within the eBPF code. However, there are many helpers which may be available for a rich set of functionalities, but not supported by one specific program type. In case such helper is already available, it is possible to edit the Linux source code and add that to the allowlist of helper functions that the verifier supports (see figure 3.9) and recompile the kernel.

Programming New Helpers In case the needed helpers are lacking in the existing source code, it is possible to program new helper functions in accordance with the existing eBPF conventions. A helper function prototype needs to be provided, including information about register and return types that the verifier uses in order to verify the program. One such example of prototype is found in figure 3.10



```
BPF_CALL_0(bpf_myprintk)
{
        printk(KERN_DEBUG "mostly harmless\n");
        return (int) 1;
}

static const struct bpf_func_proto bpf_myprintk_proto =
{
        .func           = bpf_myprintk,
        .gpl_only       = true,
        .ret_type       = RET_INTEGER,
};
```

Figure 3.10: New helper function prototype definition in /kernel/trace/bpf_trace.c

3.6 The Future of eBPF

While eBPF was originally designed for networking, its transition to a full virtual instruction set and machine in the kernel made it applicable to many more use cases. While eBPF is now established as a tracing and observability tool in addition to its networking use cases, there is an increasing attention towards the security and programmability use cases (61) (62) (63).

Among the novel developments in the past year a new program type (64), Linux Security Module (lsm) was introduced to implement security security instrumentation. Moreover, Google announced that it now uses eBPF in the dataplane of google cloud (65) that makes use of Cilium (66), an eBPF-based platform for networking, security, and observability that is developed by Isovalent. Overall, a lot of attention is now focused on making eBPF usable in different areas.

The timeliness of the topic also shows in the fact that in 2020 the first eBPF summit was held, with the aim of discussing and presenting new use cases and applications that can use eBPF.

3.7 Conclusion

As we highlighted in this chapter, eBPF is a powerful kernel-level toolchain that allows users to implement their own functionality for use cases in the area of networking, tracing and security. We briefly covered however that it is also possible to extend eBPF functionality if needed. In the next chapters we will cover the ways in which eBPF is suitable as of now in order to support data processing in the storage stack of the Linux kernel, and what is the resulting performance impact.

4

Design of e^{2}BPF

The prototype that is presented in this thesis is a design and framework to implement programmable storage functions that optimize and reduce data movement, both in single machines as well as across the network. The main target being datacenters, the design choices for the prototype focused strongly on satisfying all the non-negotiable multitenancy, security and isolation requirements, while striking a balance with user needs in the programmability aspects. In the following sections we will analyze the requirements, present the prototype and explain the rationale of the design choices.

4.1 Requirements and Methodology

In the previous chapters we examined the research context where this thesis is placed. The need for performance in datacenters, as well the issue of the data movement wall stemming from the modern storage technologies performance dominating that of network interconnects. This performance gap as well as the end of Moore's law for processor scaling gradually led towards a trend of specialization both in hardware and software, in order to fit an increasing amount of use cases and applications while tailoring to their specific computational requirements.

In this landscape, researchers are exploring, among others, the field of software programmability as a possible solution. As we saw in the previous section, programmability is a research area that can be tackled from a variety of angles. However, while user-space and hardware-based programmability already have some production-ready products, the kernel-space programmability based on eBPF is still relatively unexplored in the storage function landscape, although other studies related to this thesis have been presented in the past 2 years (67) (11) (see chapter 7, Related Work).

As highlighted in the previous chapters, a strong candidate for programmability in the kernel space is eBPF. Its widespread use and adoption in fields like networking and tracing supports the hypothesis that eBPF can effectively be utilized in the storage stack as well, in order to implement user-defined functions. The goal of this prototype is to investigate the suitability of eBPF for this use case, by designing implementing and evaluating a prototype that can deliver that. In particular, this thesis also will have a focus on the evaluation of the performance impact of the BPF toolchain, in order to understand when it is a sensible choice to offload the data processing to eBPF, and for which operations one should consider doing that.

The first step in the design process is to explicitly identify all the requirements that need to be satisfied by a viable solution to the problem. For this prototype in particular, we need to collect the requirements from a variety of sources. First of all, we need to identify the stakeholders in the process. Since the goal of the prototype is to optimize data transfer within datacenters, datacenter operators and users are the stakeholders. As such, every solution will need to satisfy some fundamental datacenter requirements (the mandatory ones are indicated in bold). The datacenter stakeholders have the following requirements:

- RQ1: **Isolation**
  It is crucial for datacenter clients to operate in an environment that is isolated from that of other users. This is needed in order to protect their data and operations, and is generally speaking a key requirement for security in datacenters.

- RQ2: **Multitenancy**
  The foundation of datacenter economy is resource multiplexing. It is then crucial that a solution in the kernel space allows multiple users to leverage it at the same time.

- RQ3: **Portability**
  We do not want to make assumptions on the datacenter infrastructure and hardware when designing the prototype for the thesis. It is then important to select a kernel function that is not architecture-specific and that can be executed regardless of the underlying infrastucture

- RQ4: Ease of deployment (for datacenter operators)
  While equipping hardware is a money investment for datacenters, the advantage of software-based programmability is that it does not require an upfront investment in

that regard. Since the eBPF infrastructure is already present in all recent Linux distributions, it is a desirable requirement to implement the prototype without requiring changes and upgrades to the existing infrastructure.

- RQ5: ease of use (for datacenter users)
  As much as the prototype should not require significant changes in the datacenter operations, the same is true for datacenter users. It is a desirable quality for the prototype to allow users to implement functionalities with familiar technologies and languages.

Requirements RQ1-2 are non-negotiable for datacenter operators and tenants, since renouncing to any of those would be an unacceptable step backwards in the datacenter economy and well-established principles and contracts. R3 is a desirable feature because we cannot make assumptions on the datacenter hardware that is present in datacenters, and we aim at producing a general solution that does not depend on specific hardware architectures. In addition to the non-negotiable requirements, it is desirable to choose a solution that can be deployed easily on the existing infrastructure (RQ4), and that can be adopted by users without significant hurdles.

In addition to these requirements, there are additional ones that stem from the programmability and optimization aspect. These requirements are:

- RQ6: **Support for data processing**
  The basic requirement for programmability in the storage stack is that there is some support for data processing. The fundamental primitives in this scenario are at least a read and write primitive for the data, and the possibility to perform processing operations like arithmetic and floating point operations, or string and character manipulation

- RQ7: **Reprogrammability**
  A key advantage of software programmability over hardware programmability is that programming of new code-based extensions should be easier than reprogramming hardware. For this reason it is important that users are free to specify and change their functionalities with ease, in order to leverage this advantage

- RQ8: **Low overhead**
  It is important that the use of this infrastructure does not significantly impact performance in the datacenter

- RQ9: **Reduction of data movement**
  Since the goal of the prototype is to eventually target reduction of data movement across the network, it is important to implement operations that can reduce the amount of data transfer.

These requirements focus on the programmability aspect and are necessary for a viable solution. The language and prototype need to be expressive enough to allow a wide range of users to adopt the technology to offload the relevant parts of their workload, thus leading to the non-negotiable requirement R6. Moreover, since we focus on software-based programmability, it is important to leverage the ease of software programmability over hardware, and deliver a prototype that allows users to easily reprogram functionalities as needed (RQ7). However, it is vital at the same time to make sure that the software changes do not affect the datacenter performance if the data processing functionality is not needed (RQ8). On top of all the previous requirements, since the problem statement of the thesis stems from the issue of the data movement wall, the goal of the research is to specifically focus on mitigations to the issue of data movement across datacenters, both locally within a single node and across the networks. For this reason, one of the design requirements is to implement a prototype that can effectively achieve data movement reduction (RQ9).

Since design space that eBPF can offer is rather limited, the design methodology for the prototype will start with a systematic review of all possible program types to evaluate their suitability.

In the following subsection we are going to analyze the degree to which eBPF can fit all these requirements, in order to explain the reasons behind the choice to use it to implement the prototype.

4.2 eBPF for Storage Programmability

In the Background chapter we examined solutions that have been proposed to implement programmability in software, using a safe programming language like Rust that delivers language-based isolation in user space (12), as well as FPGA-based programmability to offload part of the computation closer to storage (10). Despite the growing interest in programmability however, kernel-space programmability solutions are still few (67) (11). However, it is significant that both the solutions use eBPF as a technology of choice.

These two being the most relevant related work for this thesis, we highlight the gap where this work fits. ExtFUSE focuses mostly on creating an eBPF-based support for FUSE file systems, without a strong focus on data processing. On the other hand, ExtOS instead focuses on offloading part of the data processing to the kernel level. However, the pilot experiment and microbenchmarks show promising results based on the use of kernel modules rather than eBPF. This is where the current design fits in, and tries to quantify the potential gains and limitations of eBPF-based processing.

As we highlighted in Chapter 3, eBPF is a very mature, dynamic and ever-evolving platform that has already seen widespread adoption in other fields like networking and tracing. Its expressiveness, high performance, extensibility and symbolic verification make it an appealing candidate to consider for storage programmability as well. However, limited research so far focused on the potential and missing pieces that would allow for its adoption in the storage programmability subsystem.

eBPF is an in-kernel virtual machine and instruction set that can be programmed in a subset of the C language. As an alternative to that, there's an increasing compatibility and support for other languages like Python that makes it even more accessible to users with the use of development framework like BCC (50). The accessibility from different, widespread programming languages works in favor of the desired requirement RQ5 (ease of use). The possibility to specify user-defined behavior through the use of a programming language also satisfies the requirement RQ7 of reprogrammability.

The user-defined extensions are then JIT-compiled and executed in kernel space without recompiling the kernel, after a comprehensive process of verification. The verification ensures that the extensions do not stall the kernel, are guaranteed to complete in a finite amount of time and do not perform illegal operations on the data. The strict verification process ensures that eBPF extensions fit the essential requirements RQ1 and RQ2 of **isolation** and **multitenancy** for users, data and datacenter operators. the just-in-time compilation can also guarantee a low performance impact, thus leading to requirement RQ8 (**low overhead**) to be met.

The huge advantage of eBPF is that it is included in Linux kernel distributions and is actively maintained and expanded at a surprising pace. The interest in eBPF as a general Virtual Kernel Instruction Set significantly showed during the development my thesis. A lot of new functionalities have been added within a few months, expanding the already very good expressivity of the language. The avalability of eBPF in the Linux kernel makes it accessible to all Linux-based datacenters right away, making it a good candidate for the requirement RQ4 of **ease of deployment**. Moreover, the continuous development work increses the **expressivity** of the language, making it more and more suitable for use cases that were previously not considered for eBPF.

As this section highlighted, eBPF is a strong candidate that can potentially fit the strict requirements for both the datacenter stakeholders and the programmability aspect. The research that tried to use eBPF for storage programmability however is still fairly limited and didn't answer to the research questions of my thesis. Moreover, some of the requirements, i.e. **performance** and **ease of use**, have not been proven for the use case of storage programmability. This is in fact the goal of the prototype design and experimentation, that will be presented in this and the following section.

4.3 Choosing a Representative Workload

As we highlighted in the problem statement, a major issue with current datacenter performance revolves around the bottleneck that is created by the data movement from storage nodes to compute nodes across the network (1). The data movement wall is especially significant and crucial for big data workloads, since massive amounts of data need to be transferred from storage nodes to compute nodes. The end goal of this prototype is thus that of addressing this issue, and designing a solution that can help to minimize the data transfer across the network by moving user-defined computation as close as possible to the data source.

Although the issue is clear, it is also true that there is a great diversity of workloads being executed in datacenters. In order to design a valid prototype however it is important for us to identify a representative set of operations that can act as a proof-of-concept for a wide range of datacenter operations and workloads.

As an intuition and consequent design choice, we decided to focus on computation that results in a sensible reduction of the size of the data with operations of **reduction** and **aggregation**, rather than computation processes that operators that work on individual data items and perform a **transformation**. This goes in the direction that we aim for in the design process with requirement RQ9, **reduction of data movement**.

This use case can be backed up by an analysis of data-analytics workflows. After examining the TPC-H and TPC-DS benchmarks for databases (68) (69) for example it is possible to highlight that all the queries perform aggregation operations like SUM, MAX, MIN, AVG, COUNT and some form of filtering on the data. This is also present in another representative workload of current datacenter operations, as we can see in the MapReduce benchmarking suite (70), where it is reported that operations like work count or pagerank reduce data from gigabytes up to bytes size after the execution of reduce operations.

This kind of workload suggests that a solution, or at least mitigation, to the data movement wall issue can be to push computation as close to the data as possible, and perform aggregation to reduce the need for data movement. For this reason we decide to focus the design of the system on two representative operations like filter and reduce functionalities.

4.4 A Design of e^{2}BPF

In the previous sections we highlighted the different aspects that need to be taken into consideration in order to navigate the design space of a storage function programmability solution, in particular using eBPF. In this section we will present a high-level design of such a prototype.

As we highlighted in the requirements section, it is important to maintain all the datacenter guarantees and pillars that allow multitenancy for users. eBPF by design can deliver such requirements, since it is executed in an isolated, process-specific virtual machine that has boundaries and protection for memory access. For this reason, we envision a design where eBPF instrumentation is placed within the kernel space and executes user-specified functionality on data. Thanks to the support of the infrastructure, it is possible for userspace and eBPF extension to communicate the required parameters and to execute user-defined verified functionality.

Furthermore, since the goal of our design is to push user-defined code that can reduce data movement, we want to target as a proof-of-concept a subset of operations that can be relevant for this use case, and offer a prototype that can implement a Filter-Reduce mechanism that can process data in the eBPF extension in kernel space.

Figure 4.1 illustrates the high-level design of e^{2}BPF. The user can specify a filter-reduce data processing sequence that is loaded by the eBPF infrastructure to instrument a function within the read path (the choice of the function will be detailed in the Implementation chapter). With data processing happening in the kernel level, the eBPF extension can communicate the result to the user without any processing needed in userspace, for example through shared maps that are offered in the eBPF infrastructure.

In the next chapter we will go in detail about the practical implementation of the prototype going in depth about the specific implementation choices and explaining how the function is instrumented, what is the Filter-Reduce API for a user to program the extensions, and how the eBPF code and userspace program can communicate.

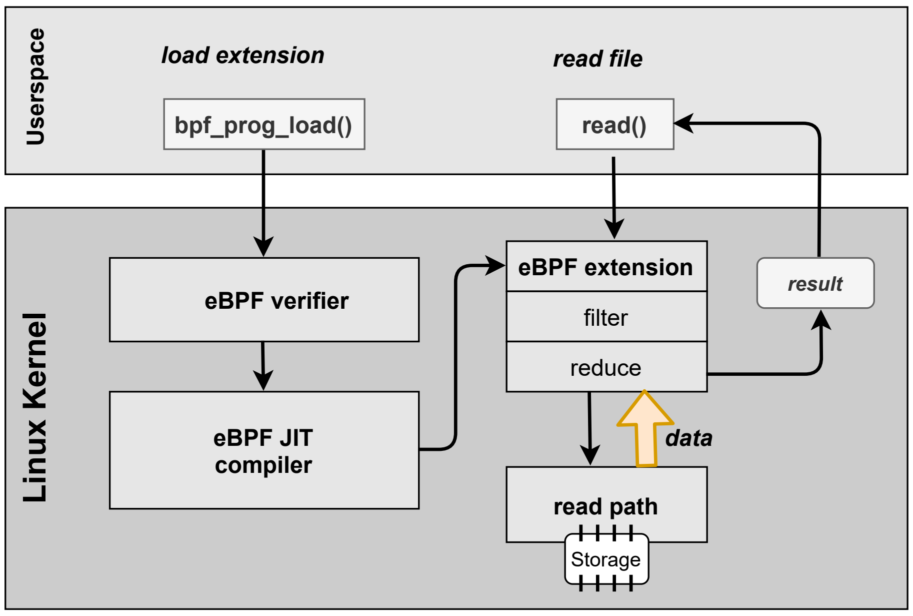


Figure 4.1: High-level design of e^{2}BPF, an eBPF-based extension for storage function programmability

5

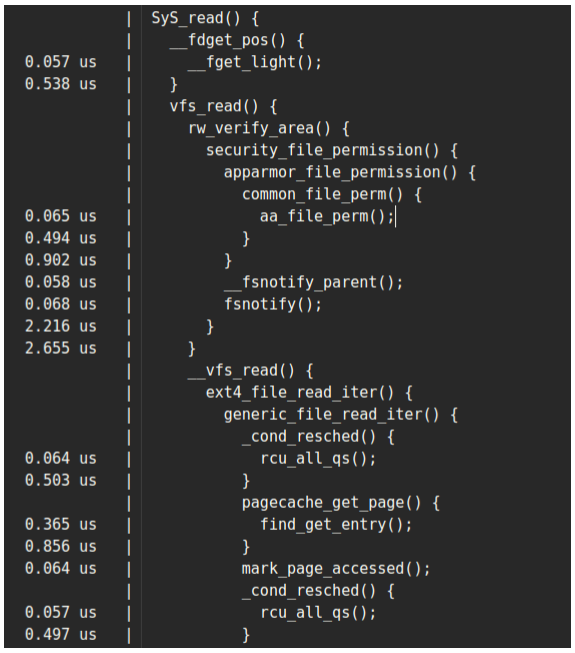
Implementation

According to the preliminary assessment of eBPF functionalities in the networking and tracing use cases, we see the potential for it to be a viable option for storage function offloading. With the design objectives in mind, we implemented a prototype that satisfies the functional requirements. However, as this chapter will highlight, the translation from the design space to the implementation was not easy. In the chapter we will go over the study of the Linux kernel code of the I/O stack, and delve into the prototype structure and eBPF mechanisms that were chosen to implement the Filter-Reduce prototype for storage programming.

5.1 What Happens when we Read a File in Linux

In the networking stack, eBPF has really powerful primitives that can be bound to sockets (file descriptors) and allow to have read and write access to network packets, as well as the power to drop packets if necessary. While this functionality is indeed what we are looking for in the I/O stack, there is no such equivalent available at the moment. For this reason, we need to revert to the standard tracing infrastructure that allows to intercept any kernel function. The first step in the process of is then to find the right anchor point to fit all the design requirements. The methodology that was used for this step was to go through a complete analysis of the options available for a standard file read, in order to enumerate the options that fit the requirements.

Finding the anchor point requires a deeper understanding of the Linux kernel code stack for I/O. In particular, since our target is to reduce the data movement in the machine and towards to network, we need to focus on the `read()` system call to understand how it can be matched with eBPF extensions. We collected a first-level trace of the call stack for a simple file read, to have insight in the kernel code using the ftrace tool (71), which gives a report as indicated in fig. 5.1. While ftrace gives a first insight in the code path, it does not trace all the calls and stops at a certain depth. After examining the trace, it was then necessary to manually examine the rest of the call stack.



```
              |  SyS_read() {
              |    __fdget_pos() {
 0.057 us     |      __fget_light();
 0.538 us     |    }
              |    vfs_read() {
              |      rw_verify_area() {
              |        security_file_permission() {
              |          apparmor_file_permission() {
              |            common_file_perm() {
 0.065 us     |              aa_file_perm();
 0.494 us     |            }
 0.902 us     |          }
 0.058 us     |          __fsnotify_parent();
 0.068 us     |          fsnotify();
 2.216 us     |        }
 2.655 us     |      }
              |      __vfs_read() {
              |        ext4_file_read_iter() {
              |          generic_file_read_iter() {
              |            _cond_resched() {
 0.064 us     |              rcu_all_qs();
 0.503 us     |            }
              |            pagecache_get_page() {
 0.365 us     |              find_get_entry();
 0.856 us     |            }
 0.064 us     |            mark_page_accessed();
              |            _cond_resched() {
 0.057 us     |              rcu_all_qs();
 0.497 us     |            }
```

Figure 5.1: An ftrace snippet for a read() system call to a file. It does not offer full visibility. figure 5.2 extends on the ftrace trace

The code path is complex, but figure 5.2 reports the relevant subset of functions that are called to perform a read system call. In the analysis and research of the right candidate, it was necessary to keep track of the relevant functional requirements. Since the goal of the prototype is to allow **multitenancy**, it is important that the function allows to uniquely identify the user and file that is targeted by the eBPF extension. However, the second requirement that needs to be met is that the function has direct access to data buffers, so that user-defined computation can happen in eBPF in order to **reduce data movement**.

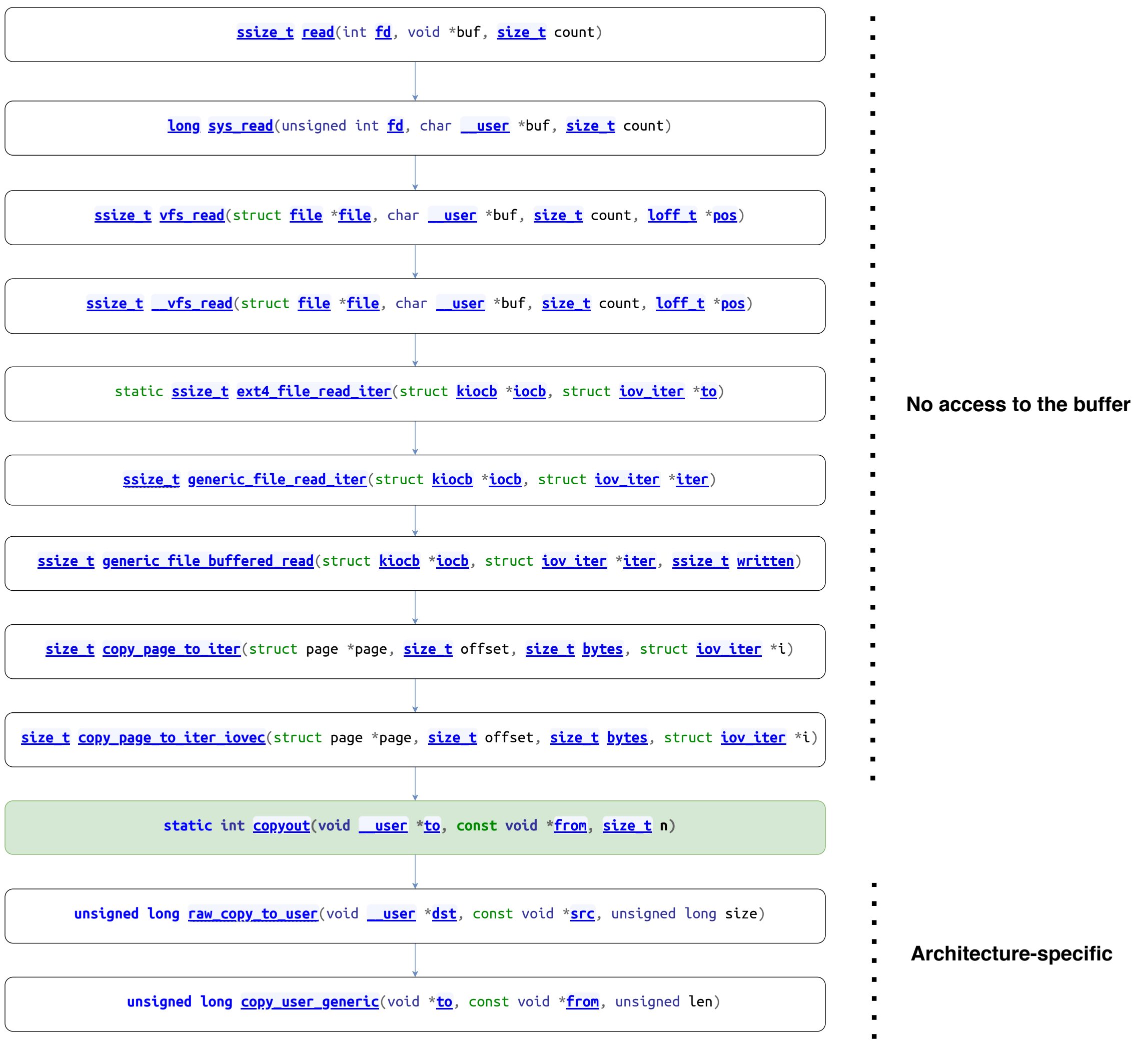
However, while most of the top-level functions preserve identification data such as the file name, file pointer or file descriptor, none of them allows to have direct access to the data buffers and instead exposes kernel metadata that we consider undesirable to expose to user access. In contrast, most of the functions that expose the raw data buffer are **architecture-specific** and written in assembly. This makes them undesirable for the requirement of **portability**, since instrumenting any such function would make the prototype architecture-dependent. The only function that can meet all the requirements is `copyout()`, as shown in the figure 5.2, and it exposes the kernel buffer, that acts as source for raw data, and user buffer where data will be copied, as well as the copy size. The downside of using this function is that, contrary to the high-level functions, it is not exported as a symbol that eBPF can observe so it was necessary to modify the kernel code in order to allow that. This partially goes against the requirement RQ4 (ease of deployment) since datacenters would be required to modify the kernel in order to distribute such prototype. Given the lack of convenient alternatives, and the harmless kernel modification that only consists of the exposure of a function symbol, we decided to use the copyout function as hook point despite this downside.

Since the information about file descriptor and file name is lost at this level, it is important to design an alternative way to isolate and differentiate between users. Once the eBPF code is instrumented, the copyout instrumentation will be triggered on every read call so it is important that every user triggers the relevant extension on the appropriate file. Since we have access to the process ID and the buffer address that the user chooses as destination for a the file read we can distinguish between different users and target files with the identifying tuple **(PID, buffer address)**, assuming that a single user will not reuse the same buffer to perform a read from different files. If that is the case, the prototype will trigger the same instrumentation. Having identified a proper function to instrument, as well as an ID tuple to identify users and files, it is now possible to start building the prototype with eBPF.

5.2 Implementation

The identification of `copyout()` as a suitable anchor point for the eBPF instrumentation allows to go to the next step, and design the prototype mechanisms and implementation details, all while keeping the requirements in mind. The end goal of the prototype is to effectively let users offload part of the computation to eBPF and reduce data transfer. As highlighted in the Design chapter, the objective is to find a way to maximize data reduction and consequently data movement. On a practical level, the solution needs to be implemented with eBPF and this introduces a new set of questions:

- **Q1**: What kind of infrastructure support does eBPF offer to execute code in the storage stack?
- **Q2**: How will the user and eBPF extension communicate?
- **Q3**: What functionality is available to the user programming an extension?



```
ssize_t read(int fd, void *buf, size_t count)
long sys_read(unsigned int fd, char __user *buf, size_t count)
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
ssize_t __vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
static ssize_t ext4_file_read_iter(struct kiocb *iocb, struct iov_iter *to)
ssize_t generic_file_read_iter(struct kiocb *iocb, struct iov_iter *iter)
ssize_t generic_file_buffered_read(struct kiocb *iocb, struct iov_iter *iter, ssize_t written)
size_t copy_page_to_iter(struct page *page, size_t offset, size_t bytes, struct iov_iter *i)
size_t copy_page_to_iter_iovec(struct page *page, size_t offset, size_t bytes, struct iov_iter *i)
static int copyout(void __user *to, const void *from, size_t n)
unsigned long raw_copy_to_user(void __user *dst, const void *src, unsigned long size)
unsigned long copy_user_generic(void *to, const void *from, unsigned len)
```

Figure 5.2: A subset of the read path in the Linux kernel. The `copyout()` function is a viable candidate since it offers access to the kernel buffer containing raw data, and copies it to the user buffer

- **Q4**: What API will the user have?

In this section we will answer these questions (respectively in the following 4 subsections) and present the basic mechanisms and primitives that will be used as building blocks for the prototype, explaining the constraints that they introduce and the ways in which they affect the design decisions and process.

5.2.1 Choosing eBPF kprobes

eBPF offers many different program types, and its functionality is continuously expanded. However, the design requirements put some limits and guidance for the choice of the program type. In particular, while in the networking stack it is possible to attach eBPF programs to sockets (file descriptors), there is no equivalent support for such functionality in the storage stack. This imposes to use the standard tracing infrastructure, that can offer increased flexibility and observability throughout the whole kernel by instrumenting function calls within the kernel, i.e. the read stack and in particular the `copyout` function in our case. The eBPF `kprobe` program type is the most suitable candidate for this use case.

The options in the tracing infrastructure at the time of the design decisions were three: kprobes, tracepoints and raw tracepoints. While tracepoints are usually deemed more stable than kprobes because they do not depend on the function signature, tracepoints are less desirable for us because they require active support within the kernel. It is not possible to instrument a tracepoint for a function that does not have the support for it within the linux kernel. Support for tracepoints is hardcoded within the function calls in the kernel, and maintainers keep the number of such hooks to a minimum since it can affect the performance of the system, while uninstrumented kprobes do not add any overhead in standard kernel execution since there is no dynamic linking and JIT compilation happening. Instead, when tracepoints are not instrumented with an eBPF extension they are substituted by NOP operations in the running kernel code (72). Since we are instrumenting a function that is commonly used in the kernel within the read path, and the prototype is not necessarily active in the kernel, we believe that using tracepoints would have a performance penalty that is not justified by the improved stability of the entry point, compared to the kprobe.

Kprobes on the other hand can be dynamically instrumented on any kernel function, and do not need active kernel maintenance, as long as their function symbol is exposed in the kernel code. As we saw in the previous section, we decided to accept this in the design and modified the kernel to export the copyout symbol as a minimal change in the infrastructure. Kprobes are convenient because they do not have any performance penalty when they are not instrumented, Although they come at the cost of instability in case the function name and signature is changed. In that case, the kprobe instrumentation will fail (73) unless the kernel is recompiled and kprobe updated to recognize the new function signature.

Another alternative to kprobes was introduced late in the development process of the thesis. A more efficient alternative to kprobes is the use of the program type `fentry` (74), that was beginning to be presented and discussed at end of 2019, and subsequently merged in the mainline Linux kernel. However, the ftrace program type was still under development for the majority of the thesis development, and lacked libbpf support as well as support for some necessary helper functions (lack of return override support). Moreover, there was no usage example provided in the kernel code and the active discussion in the bpf development mailing list made it an unstable target to base the development on. Due to the novelty and immaturity of the ftrace program type, we decided to continue with kprobe for the development of this thesis. We believe that the similarity in semantic and use cases of kprobe and fentry program types will allow for a smooth transition to the new program type once the support is more solid.

5.2.2 Communication Primitives between Userspace and Extension

After identifying the right eBPF program type it is necessary to understand how the user and kernel space can communicate the necessary information. The information needs to flow in both directions. As we highlighted in the previous section, the use of copyout() as a base for the instrumentation forces us to use the buffer address as part of the identification tuple for users and files. The user then needs to communicate this to the eBPF code so that the appropriate data processing functions can be triggered according to the user and file. eBPF provides a convenient mechanism for such communication, through the use of shared maps between user and kernel programs. The maps act as a key-value storage that can be used to communicate both ways. Given its convenience, we use a map in order to communicate the buffer address from userspace to kernel space.

The communication of the result of the data processing however offers two different alternatives. Once the data has been processed and aggregated, it is clearly possible to communicate the result through another map so that the user can access the result. As an alternative, it is possible to override the execution of the instrumented function with the use of `bpf_override_return()` and use the destination buffer to pass the result to the user. This helper function modifies the instruction pointer of the underlying function to have it only execute a return instruction, instead of calling the original function code. This is a very convenient and desirable functionality for the data movement reduction, so we decided to test it (see figure 6.5). While we will show in the evaluation how this can potentially impact performance positively, we were not able to effectively use it reliably due to an unreliable behavior of this helper function. The override return does not execute reliably especially for large file transfers, so it is not a reliable option to use the user buffer to communicate results. For this reason, although we lose on the potential performance gain given by the return override, we decided to share the result through a map in order to have reliable communication.

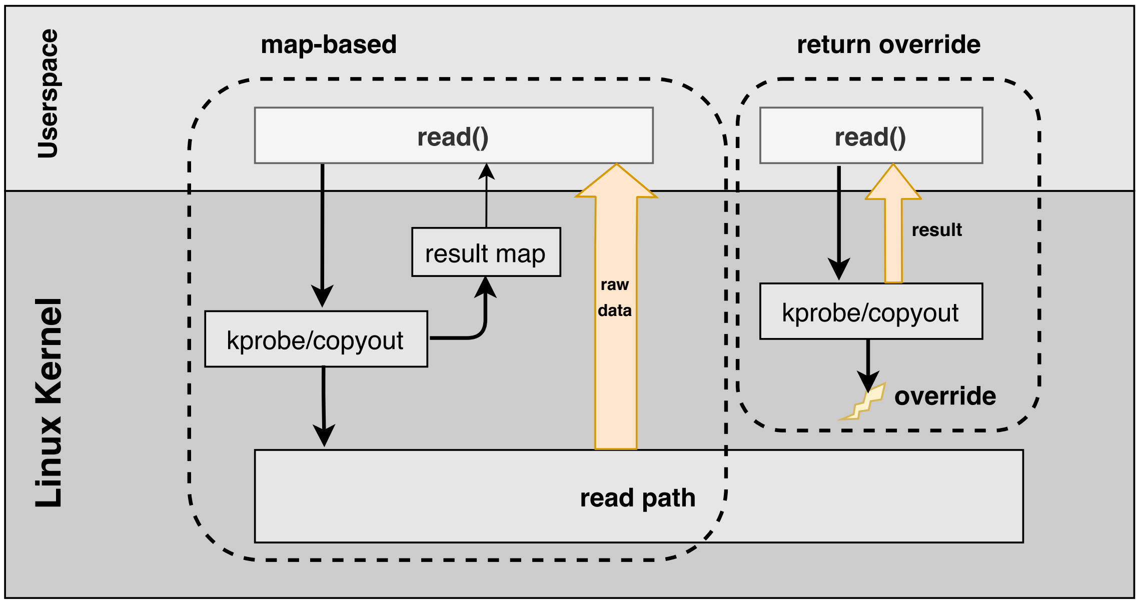


Figure 5.3: The different options for communication between userspace and eBPF extension, based on maps or on function override

5.2.3 Helper Functions

As highlighted in Chapter 3, eBPF allows for standard arithmetic operations in a subset of the C language, but is restricted for operations such as pointer dereference and read/write accesses to the buffer. For this reason, we need to use bpf helper functions in order to perform some necessary operations in order to perform data processing. While the buffer read and write accesses are available through the standard kprobe helper functions of `bpf_probe_read` and `bpf_probe_write_user`, an important aspect that is missing in the default infrastructure is the conversion from char to integer data type.

While this functionality is not available in the kprobe program type, it is nonetheless present for other eBPF program types. In order to use this functionality it is then necessary to whitelist the helper functions within the kprobe program type, thus leading to an additional kernel modification. The conversion helpers that are available are `bpf_strtol` and `bpf_strtoul`, allowing to parse and convert long and unsigned long integers from char arrays.

These functions, in addition to the standard map communication helpers and the `bpf_tail_call` function (see next subsection) allow to have all the access and modification primitives that are necessary in order to implement data processing in the kernel. A summary of the helper functions that we used for the thesis is reported in the table 5.1

Helper	**Default support**
bpf_map_lookup_elem	y
bpf_map_update_elem	y
bpf_map_delete_elem	y
bpf_probe_read	y
bpf_probe_write_user	y
bpf_tail_call	y
bpf_override_return	**n**
bpf_strtoul	**n**

Table 5.1: Summary of helper functions that are used in the Filter-Reduce prototype. The helper functions that do not have default support in the eBPF infrastructure cannot be used without a modified Linux kernel

5.2.4 Function Composition

While users can program a single eBPF extension to implement their functionality, we believe that renders the prototype not sufficiently modular and flexible, in addition to it being more error prone. The first step of the execution of the kprobe has to differentiate between spurious calls to the `copyout` function coming from other paths in the live system, and the ones that a user has explicitly instrumented. As such, the first instructions in the eBPF extension access the buffer address on the map, parse and convert the parameters, and if necessary follow up with function execution. This first stage is standard and shouldn't be left to the user to program, so that we can ensure isolation between different users. For this reason, the user-defined code is isolated in different functions that are chained together after this initial step is performed by the framework.

The function chaining can be done by leveraging the eBPF tail calls. Tail calls are a mechanism that allows to chain different eBPF programs that are verified independently of each other. Independent verification of different programs allows users to execute code that is more complex than the 1M Lines of Code limit that is imposed by the verifier for a single eBPF program. By doing so, the user has more flexibility to write longer programs as well as to perform a more dynamic flavor of function composition within the eBPF infrastructure.

For these reasons, we decided to make use of tail calls as a means to implement function composition in the prototype. Upon loading of the eBPF modules, two special maps need to be populated and they will act as jump tables in the program execution. This mechanism allows for two positive gains. First of all, the instrumentation of the core `copyout()`

function can be left to the framework and doesn't need to be touched by users. This is a good safety guarantee to ensure proper parsing of buffers and access to data. Second, the setup of a tail call infrastructure makes it easy to transition to more complex execution schemes where different tail calls can be chained in sequence based on events, even though the current prototype only makes use of a standard sequence of two nested tail calls.

5.3 e^{2}BPF: the Filter-Reduce Prototype

Given the primitives and mechanisms that we presented in the previous section, it is now possible to design a structure for the prototype. As we highlighted, it is convenient for verification and security purposes to break the execution flow into smaller steps. In particular, we decided to implement a two step processing mechanism that can be leveraged by the users called e^{2}BPF.

The prototype provides a standard instrumentation for the `copyout` function by the means of a kprobe. This instrumentation, by design of eBPF, is triggered regardless of the user and file once the eBPF kprobe is loaded. For this reason it is crucial at this stage to quickly dismisses kprobe invocation from spurious code paths and instead executed the Filter-Reduce functionality in the appropriate cases.

The first step of the processing is then a fast process and file identification. At this stage the parameters of the function call are accessed together with the buffer address that is present in the map, and if the identification tuple matches with data on the map then the processing steps are called through the tail call mechanism that we explained in the previous section.

After the process and file identification starts the data processing logic. The first step of the processing is a Filter. The filter should access data from the kernel buffer, filter it according to user logic and write the filtered buffer in the user buffer of the function call. At this point, the second step of the execution can take place and the reduce operation is triggered with a second tail call. The reduce operation can read the filtered buffer from the `to` buffer of the copyout function, that was just written by the filter function, and goes on aggregating the result in order to produce a single numerical result. This result is then passed to the userspace through a shared map. An overview of the architecture of the prototype can be found in figure 5.4

With this mechanism in place, thanks to the use of independently linked, loaded and executed tail calls it is possible for users to develop and select functions for the two steps of the execution, as long as the communication API remains the same. Within the single functions that are concatenated in tail calls there happen the parsing, copying and processing of the buffers in order to implement data processing functionalities.

In the next section we will rapidly examine the limitations of this design, and the compromises that needed to be made in order to reach this result.

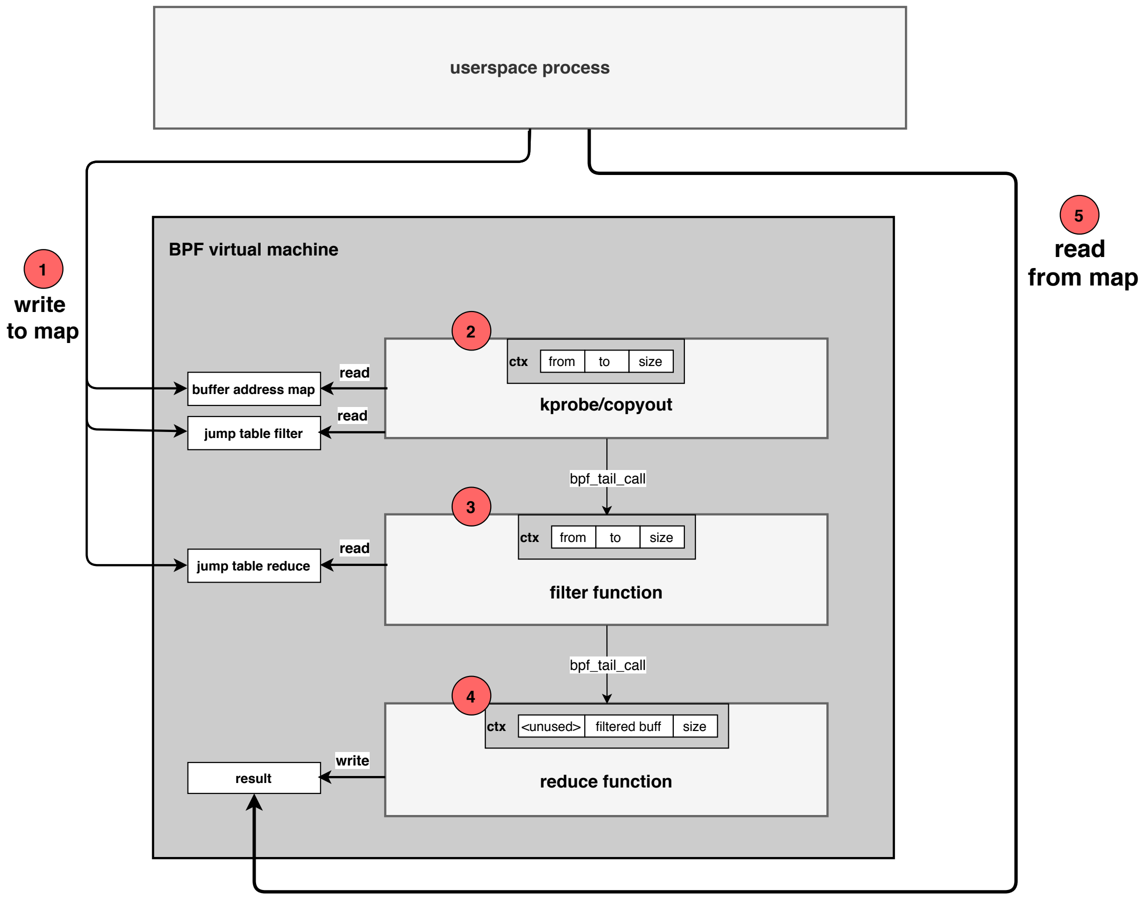


Figure 5.4: Architecture and main operations (numbers indicate order of execution) of the Filter-Reduce prototype

5.4 eBPF reality-check

With all the implementation pieces set in place, it is important to assess the results and compromises that needed to be made in order to deliver the prototype. As the evaluation will further show, although eBPF has good potential for data processing in the storage stack, a lot of access primitives and mechanisms are still not refined and optimized for this use case, and lead to the need to compromise on a lot of desired functionalities.

First of all, the unpredictable behavior of `bpf_override_return` prevented from reliably being able to save CPU cycles of an unnecessary buffer copy. In addition to this, the use of helpers to mediate even read-only access to the data buffer introduces unnecessary complexity in the program that could instead be leveraged to give the user more leeway to perform more complex data processing tasks.

Last but not least, the operations that are currently allowed in the eBPF infrastructure are restricted to integer arithmetic and logic operations, so it was not possible to implement all the most common functions that were identified in the Design chapter, for example the average function, since they would require floating point operation support.

5.5 Summary

In this chapter we examined all the main implementation choices and details that can translate the design requirements intro a well-functioning prototype. In the evaluation chapter these components will be evaluated individually as well as end-to-end to verify and assess the feasibility of an eBPF-based data processing framework within the software stack of the Linux kernel individually

6

Evaluation

To the best of our knowledge, there has not been an extensive quantitative study of the performance of eBPF-based instrumentation. In order to test the design and implementation of the prototype, it is vital to assess the performance impact that eBPF can have on kernel performance. In particular, this information is crucial to understand if and what kinds of computation make sense when offloaded. In this chapter we will report on findings from the evaluation of the prototype, and comment on the reasons why eBPF filters lead to a 2x slowdown compared to userspace filters using the current eBPF support, and what are possible mitigations to it.

6.1 Experimental Setup and Plan

In order to evaluate the Filter-Reduce prototype we need to measure the system at different levels. In order to gain confidence in the end-to-end evaluation, all the individual steps that contribute to the overhead are also measured separately and through different, more fine-grained experiments. Having a one-level-deeper measurement allowed to cross check the results and validate the soundness of the end to end measurements (24).

As figure 6.1 summarizes, the experiments will target the different components in the eBPF infrastructure as well as the prototype. Measurements will be conducted for the end to end latency, the verification process, the just-in-time compilation, as well as fine grained measurements of the eBPF helper functions that were used in the prototype. In addition to this, the baseline time for read operations is gathered and compared to the eBPF-instrumented code path in order to gain insight on the potential gains as well as the impact of the buffer copy from kernel to userspace.

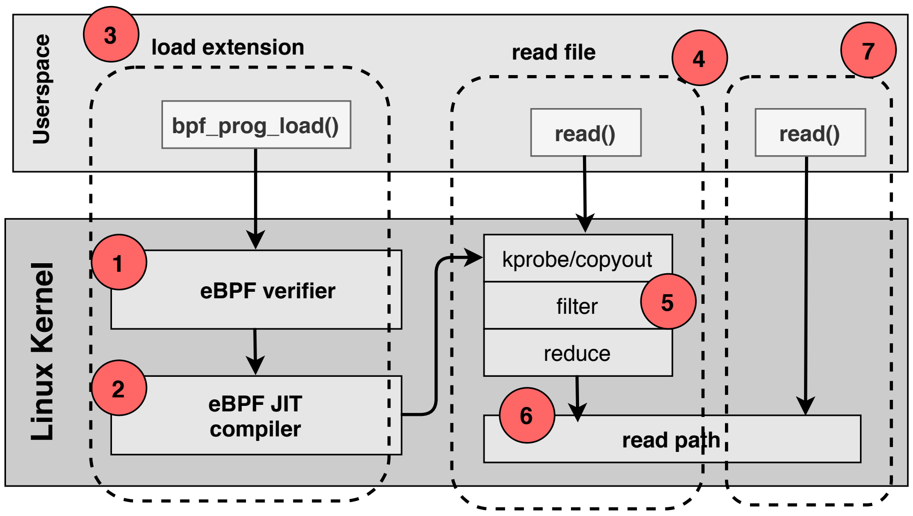


Figure 6.1: An overview of the target points of measurement and benchmark for the Filter-Reduce evaluation

The experimental setup for the prototype is a QEMU-based virtual machine with Ubuntu 18.04. We are using a modified version of kernel 5.7 that allows to instrument the `copyout` function. The VM can use 4 cores, and is equipped with 8GB of data and 150GB of hard disk drive. The experiments focus on evaluating a single process performance to collect fine-grained information about the steps and additional overhead that eBPF introduces. In order to gather accurate timing measurements we relied on the CLOCK_MONOTONIC clock in the Linux kernel, measuring at nanosecond granularity. This clock is the same that is accessed both from the userspace timekeeping function `clock_gettime()` as well as the eBPF timestamp mechanism `bpf_ktime_get_ns()`, and thus offers a more uniform measurement. The clock of the virtual machine accesses the host clock in order to avoid clock skews and drifts deriving from network synchronization and adjustments, as well as from VM sleeping time in the system (75).

6.2 eBPF Microbenchmarks

In this section we will present results from the evaluation of microbenchmarks in the eBPF infrastructure. The benchmarks target verification, just-in-time compilation, helper functions, tail calls and buffer copy overheads.

LOC	**mean** [ms]	**standard deviation** [ms]
60k	47.60	6.52
125k	87.16	6.53
250k	161.28	5.78
500k	315.93	8.46
1M	644.90	8.94

Table 6.1: Mean and standard deviation of verification times, by Lines of Code. JIT time is negligible with mean 2.8us and standard deviation 770ns

6.2.1 Verification and JIT-ting

As we highlighted in Chapter 3, eBPF provides a symbolic execution and verification process for its extensions. While the complexity of such process is contained by the non-Turing completeness of the extensions, that sets a maximum program length to 1 million lines of eBPF opcodes, verification and symbolic execution is still a time-consuming process. For this reason, given the reliance of the prototype on the eBPF infrastructure, it is important to evaluate the performance of the verification process and its impact on the overall pipeline. The same is true for the just-in-time compilation that happens every time a kprobe-instrumented function is called in the Linux kernel.

We tested the verifier by measuring different program complexities and their verification VS execution time (figure 6.2, table 6.1). he program complexity was increased by looping over the same set of operations more times (including helper function calls), since the compilation process unrolls all the loops and verifies the instructions in executing in order from the first to the last instruction.

We tested the verifier complexity on the `copyout` function, loading the eBPF extension and executing read() call for a single page (4096 bytes) from file. Every program complexity was tested 50 times to have a cumulative execution time that exceeds that of the single experiment by 2 orders of magnitude. Caching of the data in memory does not affect this measurement since I/O overheads are not detected in this benchmark.

As is is possible to see from the graph, the verification time dominates the execution time and exponentially grows with the complexity of the program, whereas execution time appears to grow more linearly. This is a positive finding since verification cost is one-off component in the execution, happens at load time of the eBPF module and does not repeat during the program execution. This means that over time the verification cost can be amortized with long executions, making it negligible compared to the execution time if reprogramming of the filter or reduce function does not happen continuously. The JIT-ting cost is not visible in the graph and represents a very negligible portion of the cost. It is not shown in the graph because JIT-ting costs are in the microsecond order of magnitude.

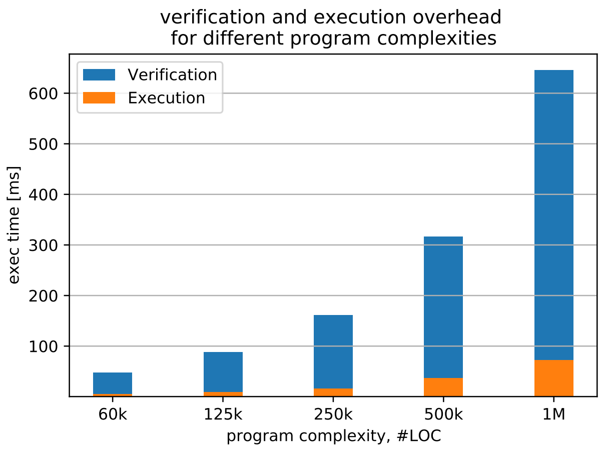


Figure 6.2: Verification overhead for 5 different program sizes

6.2.2 Helper Functions

The kprobe program type in the eBPF infrastructure allows to gain observability in virtually every function executing in the kernel, gaining access to the parameters the function s called with. However, due to the verifier and its constraints, it is not possible to access and manipulate parameters in the context of the kprobe invocation without the use of helper functions that are provided by the eBPF infrastructure.

We then evaluate the three helper functions that are relevant for the prototype, in order to assess whether they introduce significant overhead in the computation. The functions that we benchmarked are `bpf_probe_read()`, that is used to gain read access to the data, `bpf_probe_write_user()` that is used to write content in the user buffer, and `bpf_strtoul()` that is needed in order to convert chars to numbers (unsigned long in this case) in order to perform arithmetic operations.

All the measurements are based on single helper function calls, and we vary the buffer size in order to estimate how that affects the overhead. The buffer size where data will be copied to and from can go up to 256 bytes for the read and write helpers. This is a huge limitation since it imposes to operate in a loop in order to parse the whole 4096 bytes of the source kernel buffer. However this limitation cannot be overcome as it derives from the eBPF-imposed limitation on the stack size of every eBPF extension, that is set to 512 bytes. Since the stack needs to be used to store other intermediary results and variables, we decided to cap the read/write buffer size to 256 bytes in order to leave the user more freedom to create additional variables.

We tested every helper function call for different buffer sizes, and each measurement on a set buffer size was repeated for 1000 iterations in order to have a cumulative execution time that is orders of magnitude greater than the single benchmark. The workload that was used for testing is composed of only read calls from file (1 page each). Caching of the data in memory does not affect this measurement since I/O is not part of the measured cost.

The results of the experiment are reported in figure 6.3. As the figures show, the most overhead comes from the read helper, that dominates the other two by 1-2 orders of magnitude. This can possibly be attributed to the fact that the source buffer of the helper is a kernel buffer, and has more security mechanisms in place under the function call. Under the hood of the read helper execution is in fact distinguished whether the buffer that is being accessed is in userspace or in kernel space. This finding is however unfortunate since it places a lower bound on the execution time of eBPF probes. Parsing of the source buffer cannot be avoided so it will always yield to significant overhead.

On the other hand, the write helper function has a significantly lower impact on performance as the buffer size if increased. As it happens for the read helper, this can be explained with the fact that the buffer that is being accessed for write is a userspace buffer and the security mechanisms around it are less strict. While the read helper cost cannot be avoided, the write cost can change depending on how much data is filtered out by the filter function. The write helper is used to pass the intermediate filtered buffer from the Filter to the Reduce tail call, and improving the percentage of filtered data can reduce this cost, although it is unfortunately not the dominating cost in the execution.

The cost of strtol conversions is also worrisome. While it is true that the order of magnitude is the same of the write helper, strtol and strtol functions can only convert one number at a time, and the limit for this is the 8 bytes of space that are defined for an unsigned long type in the machine. The cost of strtol is then very significant because it needs to process the whole buffer of 4096 bytes, and introduces a significant cost when called repeatedly. Once again this is an unfortunate finding since the parsing and conversion cost of the numbers in the buffer cannot be avoided and introduce a lower bound in the cost of execution.

The high cost of the helpers is not limited to the time overhead. Using helpers to mediate access to the buffers introduces complexity in terms of lined of code. This is not desirable and in fact yields to the necessity to split execution into smaller pieces through the help of tail calls. Without the use of tail calls, a simple read-parse-write cycle is enough to overrun the maximum number of lines of code for a source buffer of 4096 bytes. For this reason, it is important to split the computation with the use of tail calls.

6.2.3 Tail Call Latency

As shown in the previous experiment, helper functions add a significant overhead both in terms of processing time and especially lines of code. In fact this became evident when designing the first version of the prototype without tail calls, we found that processing the whole 4096 bytes of kernel buffer in a read-strtol-write cycle already gets beyond the maximum eBPF instruction length and does not leave space for code execution that is user-defined. For this reason, it is undesirable to keep the programmable logic in a single eBPF program, in order to leave the user more leeway to implement their functionality. In order to deliver this, the prototype adopts a multi-step sequence of processing that happens through tail calls.

Tail calls allow to link eBPF programs that are verified and loaded separately within the infrastructure, while being linked together through jump-tables. The tail call is supposed to be a low-overhead mechanism since it does not cause context switches, reuses the same stack of the caller program and doesn't return to the caller. This allows to chain eBPF programs in order to increase the program complexity without adding significant overhead, except for the jump to the tail call function code.

We tested the performance impact of tail call by comparing the execution of the same eBPF logic happening in a single program, and comparing it with a program that is composed by 2 tail calls like Filter and Reduce. The workload is a standard read cycle with increasing read size. This time the measurement happens over read sizes greater than 1 page since it is not possible to detect any difference in execution time within a single execution. For this reason, we iterate on more significant read sizes to accumulate the cost of multiple tail calls trying to detect if there is a trend of increasing cost. For every read size the measurement is repeated 1000 times.

As the figure 6.4 shows, the tail calls do not seem to add any significant overhead to the program execution even with an increasing number of function calls. This means that using tail calls will allow to scale up the computation complexity up to 32 million lines of code (the limit the verifier imposes is 32 nested tail calls) without any overhead that derives from accumulation of tail calls.

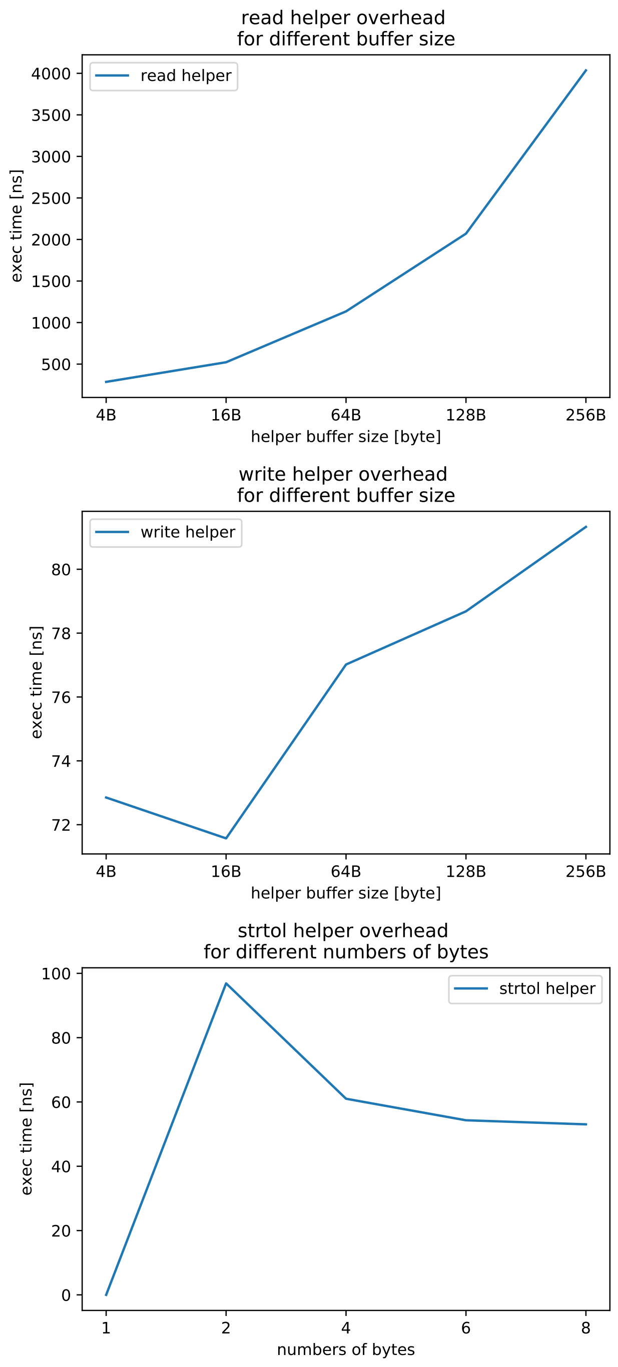


Figure 6.3: Helper function overhead to parse a 4096-bytes buffer and convert to integer, varying the buffer size in the helper.

6.2.4 Cost of Data Copy

As we decided to instrument the copyout function, we are in the position where it is possible to avoid the buffer copy from the kernel to the user buffer if the computation is offloaded to

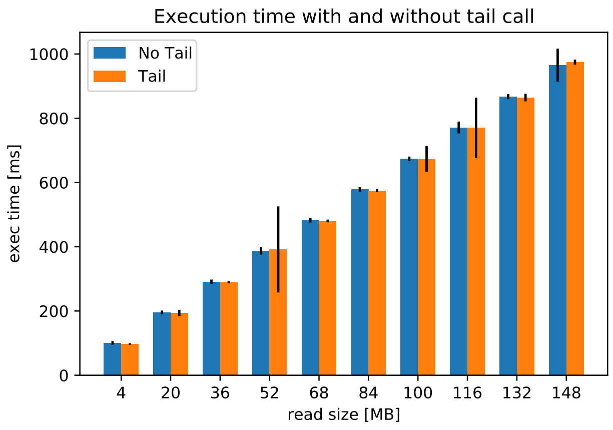
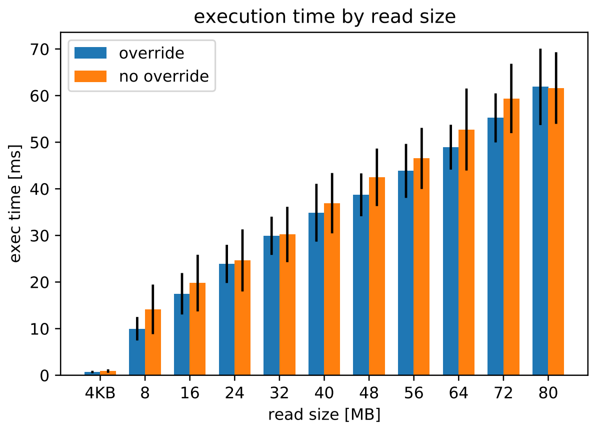


Figure 6.4: Comparison of the execution of the same programming logic within a single program and through tail calls. Average and standard deviation are reported

the kernel. This can happen because eBPF allows to hijack the execution of the underlying function, and avoid its execution. With the mechanism of `bpf_return_override` it is possible to prevent the buffer copy from happening, thus leading to a gain in the data movement.

As we explained in the implementation Chapter, this mechanism does not behave as expected with the `copyout` function, leading to unreliable function overriding that does not allow to reliably use it in the prototype. However, we decided to evaluate it regardless to show the potential gain in performance that could stem from reliable execution. Figure 6.5 shows that is is possible to measure a difference in I/O performance if the buffer copy is avoided. While the function starts to fail more and more as the number of invocations increases, we can observe that within a single invocation there is an approximate gain of 20% of the execution time within the read path if the data copy is avoided. We stopped the experiment at a read size of 80MB because the failure-rate of the function override reached a stall point there.

Although execution is not reliable over time, we plotted in figure 6.6 the difference in microseconds between the two cases, with and without function execution. Avoiding the function execution can yield to up to millisecond-scale gains so it is important to investigate the effect of reliable function overriding on long data transfers.



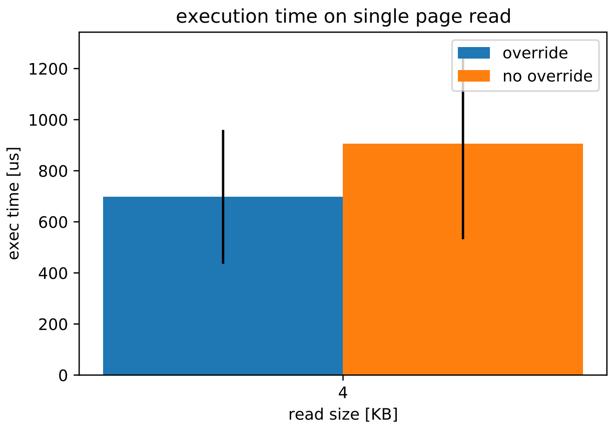


Figure 6.5: Comparison between the latency of an instrumented read() call with and without overriding the execution of the copyout function

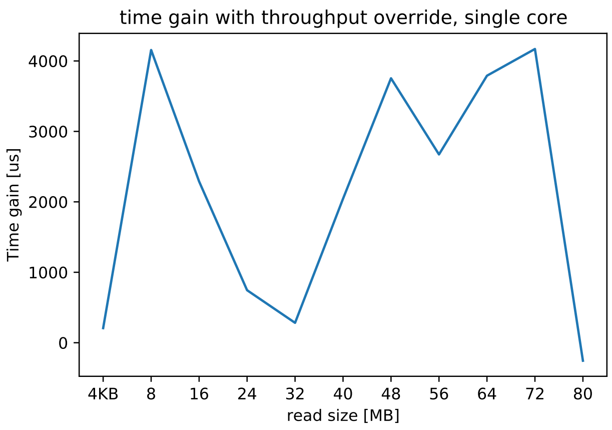


Figure 6.6: Throughput and throughput gain for single process, read size from 4MB to 160MB

6.3 Filter-Reduce evaluation

Having the eBPF microbenchmarks as a support, it is possible to design experiments and evaluate the system end to end. For this experiment we execute a sample Filter-Reduce workload and time the verification, eBPF execution time, native read time as well as an equivalent userspace-based Filter-Reduce sequence.

Since we observed the impact of helpers in function execution, we repeated the experiment varying both the read size and the percentage of data that is filtered in the Filter step, going from 0% to 100% (indicated as ff0-100). With these experiments we can compare the execution time of the eBPF-based Filter-Reduce with the userspace Filter-Reduce, as well as observing the breakdown of the different cost components, namely I/O, verification and filter execution.

This experiment was run for 5 different filter factors and 7 different read sizes (16MB increase every time) to extrapolate if the growth of the complexity had some non-linear component. Every measurement was repeated 50 times. The program complexities are pre-determines, so for example filter-factor 25 performs the reduce operation over a buffer that is 25% the size of the initial one. For this reason, the verification cost decreases with higher filter factors since the reduce step is smaller by design (see figure 6.7). In a real-world scenario the verification cost will be constant and the execution complexity is the only factor that changes at runtime, but for simplicity the experiment was designed with static filter factors in mind.

Figure 6.7 reports the stacked cost of the execution of Filter-Reduce functions for both userspace and eBPF, with the 5 different filter factors ff0-100. As shown in the figures, the cost of eBPF-based processing is approximately 3 times that of the native execution and reflects the findings of the microbenchmarks, that goes up to 5x for 100% filter factor. The verification cost is constant throughout the execution, even increasing the file read size. The cost impact of the different phases of the execution for every filter factor is plotted in figure 6.8.

As the figures highlight, the eBPF filter is less efficient than the userspace one since, despite increasing the filter factor, it cannot avoid the heavy performance penalty of the read helper function. This also shows in the fact that increasing the filter factor yields to a proportionately bigger gain in the userspace execution when compared to the eBPF-based one, confirming the finding of the dominating cost deriving from the read helper. Figure 6.9 summarizes the finding, highlighting how the cost of the Filter-Reduce code

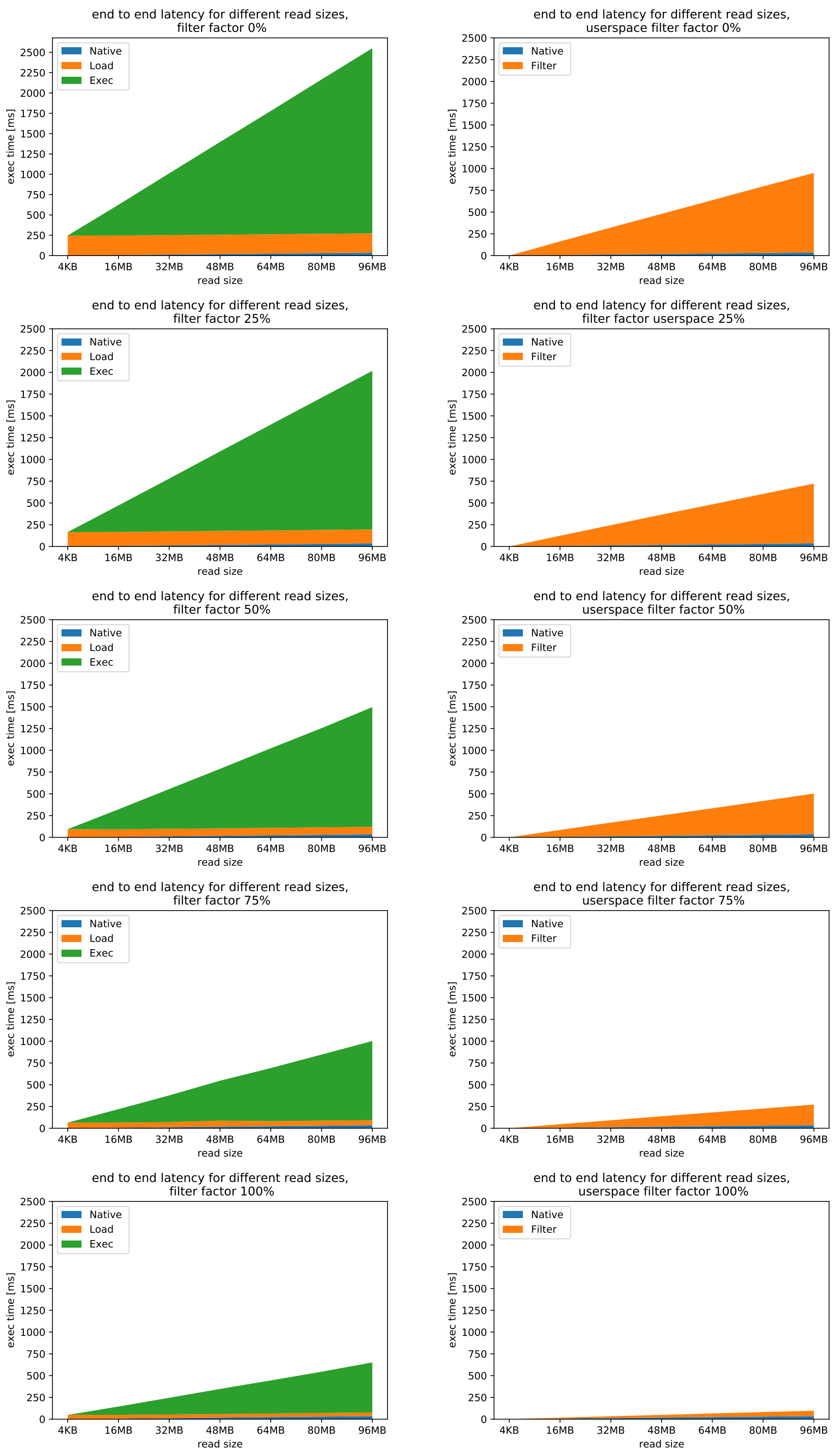


Figure 6.7: Stacked latency increasing the number of iterations over the file. Different filter factors are reported, the filter function filters from 0% to 100% of the data

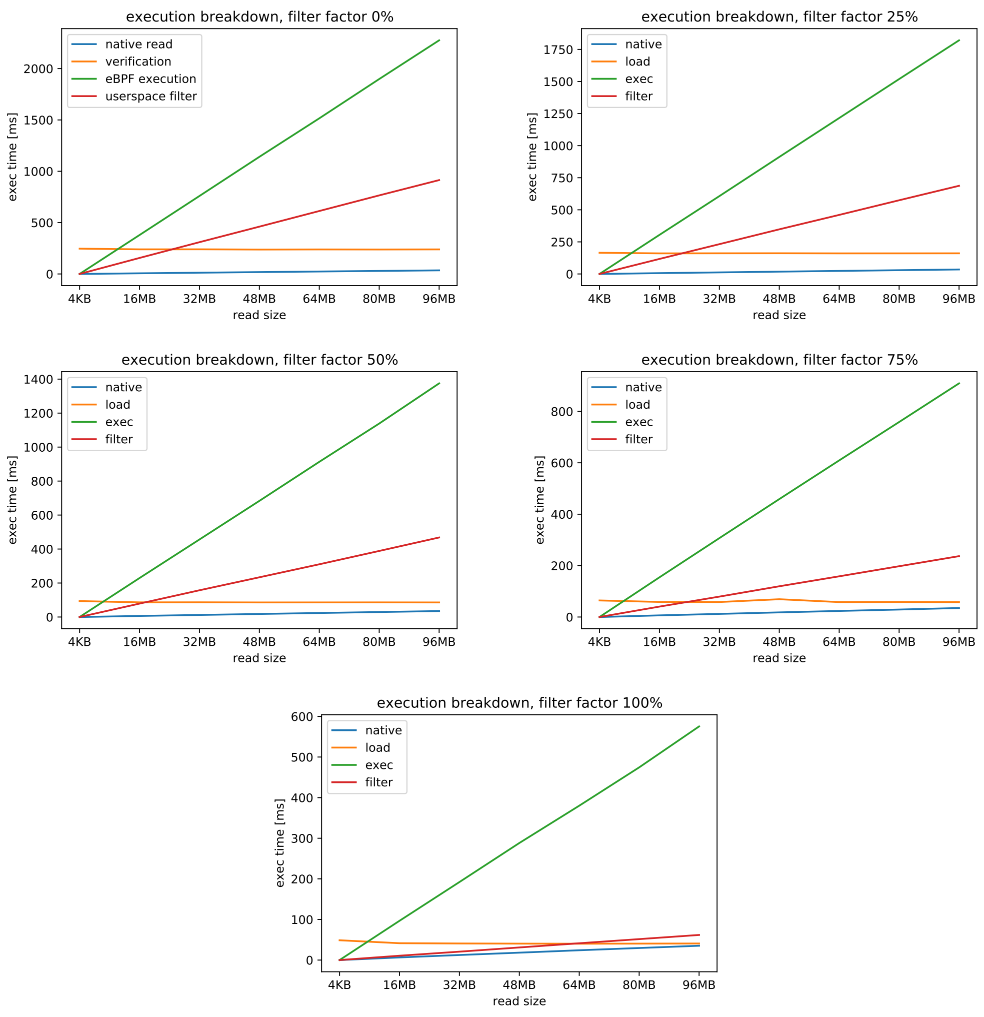


Figure 6.8: Summary of execution overhead increasing file read size, varying filter factor in the instrumentation. The number of read iterations over the file increases linearly

decreases proportionately more in the userspace execution when compared with the eBPF instrumentation that is instead stalled by the read and strtol helper function lower bound.

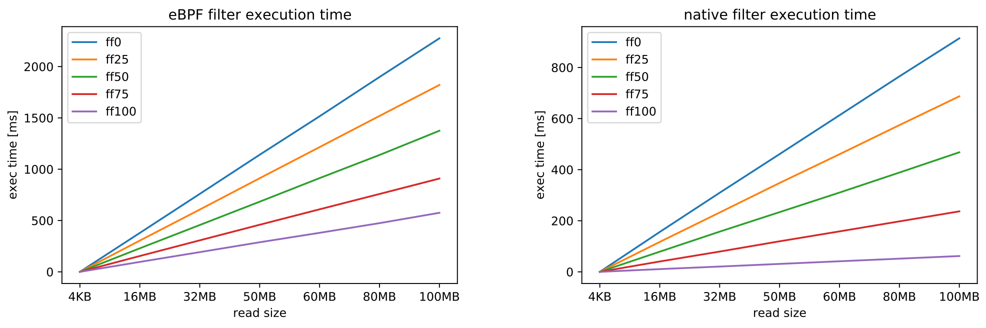


Figure 6.9: Comparison of the execution of a userspace filter-reduce and an eBPF-based filter-reduce mechanism)

6.4 Single Machine Cost Evolution

As the experiments reported, the eBPF infrastructure support for data processing in kernel is still rather lacking. This is mainly due to the lack of support for native access to parameters in order to process data, forcing to use helper functions that are not optimized for this kind of workload. However, we want to summarize the findings with a figure. Figure 6.10 reports a summary (not in scale, due to the wide differences in order of magnitude of the various components) of the different steps in the execution of both native and eBPF Filter-Reduce functionality.

It is possible to see from the figure that the cost components are different. However, it is important to stress how their evolution is different over time. While the verification step appears to dominate the eBPF cost initially, over time that remains constant is amortized over large file reads. The cost of JIT compilation will increase if the function is called multiple times, but its microsecond-scale impact does not significantly affect execution. On the other hand, the cost components that linearly scale if the file read is increased are the cost of the filter-reduce processing, as well as the time to perform a buffer copy from kernel space to userspace. This is a cost that can only be cut through eBPF instrumentation.

Under the light of this finding it is possible to estimate that with a functioning return override execution the cost of the eBPF infrastructure and instrumentation can become worthwhile based on the buffer copy gain when operating on large file read sizes. This together with an optimization of the read helper function can lead to interesting future work, as we will expand on over the next chapter.

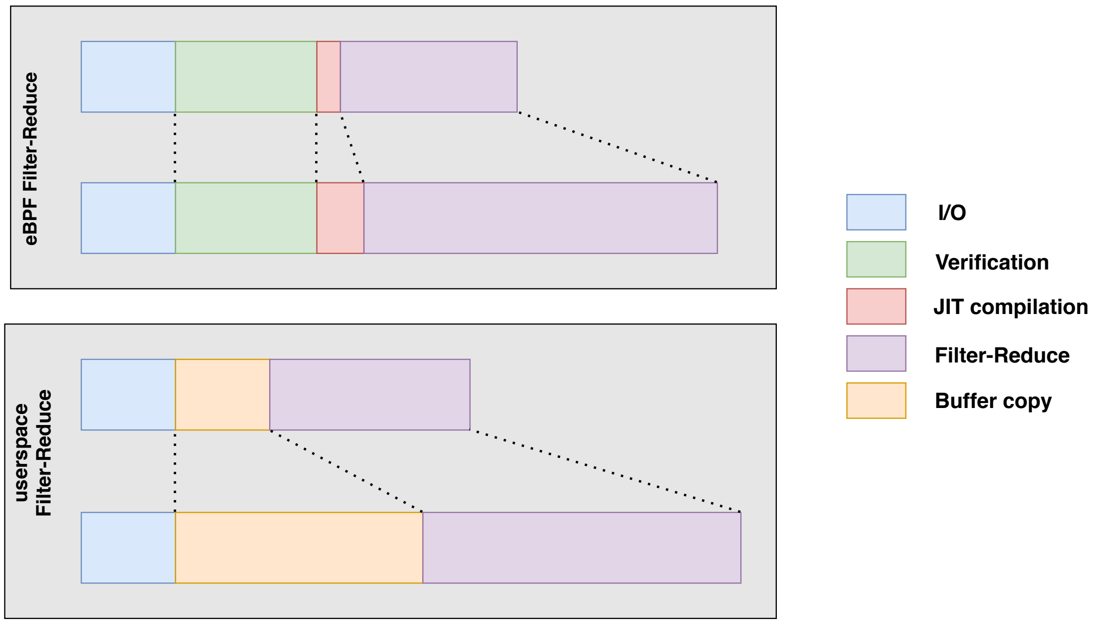


Figure 6.10: Visualization of the stacked costs for filter-reduce in userspace and eBPF-based and how they evolve increasing the size of a file read. The scale is indicative because the different components differ by orders of magnitude

6.5 Summary of Evaluation

In this chapter we reported the results of the eBPF microbenchmarks and evaluation of the Filter-Reduce infrastructure. The main findings from the experiments highlight how the cost components that stack in the execution of an eBPF-instrumented code are very diverse and impact in disproportionate ways.

While the verification phase introduces a relevant cost (up to 1 second overhead for complex eBPF programs up to 1 million lines of code) it is only executed once at load-time of the extension and can potentially be amortized by long-running executions. However, the dominating component in the overhead of the function execution derives from the use of helper functions that are needed to read the parameters of the function call and convert them to integers in order to perform operations. The parsing overhead introduces a performance penalty that is on average 3x worse than that of an equivalent filter-reduce functionality implemented in userspace, and up to 5x slower with high filter factors.

However, while these costs can be mitigated by changes in the infrastructure, we also highlighted the peculiar advantage of eBPF-based processing that allows to save approximately 20% of the execution time with a function execution override that prevents the data buffer from being copied from kernel to userspace. This aspect is the distinguishing point that sets eBPF-based instrumentation on a different level compared to both hardware-based and software-based programmability that happens in userspace.

In the next chapter we will put this research into context highlighting the different solutions that existed previously to this work, and how they implement programmability for storage functions

7

Related Work

As we saw in the previous chapters, the issue of data movement is crucial in modern datacenters and one of the possible solutions that are being explored is that of programmability (1). However, although this area of research is increasingly gaining momentum in the storage stack, it is not a recent concept and can be tackled in many ways that substantially differ from the solution that this thesis explores. In this chapter we will highlight relevant related work within the field of storage programmability.

The idea of bringing computation closer to the storage was explored in 1998 with the notion of active disks (76) (77). The case for active disks it that offloading parts of the computation to the storage can improve performance in data transfer. However, given the unavoidably low performance of Hard Disk Drives, the motivation to pursue such solution on a large scale was not sufficient and the idea remained limited to research prototypes.

The situation changed rapidly with the advent on modern storage technologies like Flash SSD and the more recent Optane memory. As we highlighted in chapter 2, these technologies have lower access latency and higher throughput, and thus justify the idea to bring computation closer to storage, a notion knows as Near Data Processing (NDP) (13). With modern storage technologies, the notion of NDP was brought back in the spotlight and explored as a means to gain performance in data processing (78) (79).

However, the challenge remained as to how one can allow the functionalities to be defined by the users through programmability. NDP can be combined with programmability through many different approaches. The use of specialized hardware like FPGAs or ASICs can be classified under the name of hardware programmability, while the use of language-based approached is called software programmability.

Hardware-based Programmability One of the approaches to implement user-defined logic closer to the storage is through the use of programmable hardware. In this area a lot of solutions have been proposed, and they usually encompass the use of an FPGA board alongside the storage device in order to program subsets of user-defined functionality. An example of this is Willow (80), a programmable SSD that allowed users to push user-defined functionality to a Storage Processing Unit that can execute user code and communicates with the host with a specialized Remote Procedure Call Protocol. While Willow focuses on generic user-defined functionality offloading to the storage device, other solutions try to offload specialized subsets of operations like database-related workloads, thus leading to the implementation of smart SSDs with query processing support (81) that can offload parts of their operations (e.g. filtering, aggregation) to the SSD. While this direction can efficiently implement programmability on the SSD, the downside is that this type of solutions is specific to a single application, namely a DBMS, rather than allowing multitenancy and programmability that is generic and applicable to a set of users. A generalization of such a capability is Biscuit (82), which allows to offload generic functionality within the SSD.

While these examples show performance improvements that derive from equipping an SSD with a CPU, other hardware-based solutions have been proposed that make use of FPGAs instead. An example for an FPGA solution is presented by INSIDER (83), where users can offload parts of the program to FPGA. The challenge with the use of FPGAs is how to ensure that users only access their data, without corruption. INSIDER solves this by allowing compute-only workloads to execute on the FPGA, which does not directly have read access to the data stored on the device.

While hardware-based solution provide performance improvements for data processing, the challenges of isolation and multitenancy suggest to also investigate solftare-based programmability where isolation and multitenancy can be handled more easily through the mediation of the operating system and the execution in separate processes.

Language-based Programmability While hardware-based programmability shows performance improvements, it requires to have specialized hardware, that might not be available to all users. Moreover, the isolation and multitenancy requirements can more easily be guaranteed with the mediation of the operating system and the execution in separate processes. For this reason, it is vital to also investigate software-based programmability. An example solution in the area of software-based programmability is Splinter (12), which allows users to program their functionality in Rust, a type-safe programming language that has strong guarantees on the safety of memory accesses. Splinter implements a key-value store that users can extend and customize by pushing their own operations to storage using Rust. While userspace programmability can be leveraged by providing the necessary isolation between users, for example through the use of Rust, other research efforts also focused on eBPF as an infrastructure to deliver software-based programmability in the Linux kernel. Two works in this direction are extFUSE (67) and extOS (11). In extFUSE, the framework provides a FUSE-based file system that has an underlieing eBPF support to offload part of the operations in the kernel, executing in eBPF. This work focuses on generic file system functionality rather than data processing workloads, and shows how eBPF has the potential to be extended in the storage stack. A solution that is more oriented towards data processing is extOS. In this work, the authors argue in favor of Near Data Processing by offloading data processing workloads to the kernel can lead to an improved performance, and identify eBPF as a possible infrastructure to support that, after an extension of the functionality in order to support more expressive functionalities.

8

Conclusion and Future Work

Data processing is ubiquitous in modern Big Data workloads, and thus represents a key task to perform within datacenters. While datacenter networks offer high throughput, the improvement of storage media like Flash SSD and Optane memory introduces a bottleneck in data transfer. For this reason, near-data computation through programmability is gaining momentum. While there are plenty of userspace and hardware-based solutions, the kernel space and eBPF remain relatively unexplorred within the storage landscape. In this thesis we implemented a prototype to assess the suitability as well as the performance impact of data processing in eBPF in order to reduce the data movement. In this chapter we summarize the findings and the answers to the research questions that motivated this work.

8.1 Conclusion

In this thesis, we explored the potential for eBPF to be used in data processing scenarios for storage function programmability. As the design and evaluation highlighted, eBPF is still not suitable for high-performance data processing when compared to userspace execution, despite the potential CPU cycle gain that comes from avoiding the buffer copy from kernel to userspace.

We summarize the findings of the thesis and relate them to the relevant research question

RQ1: What is the performance impact of the eBPF infrastructure? The impact of eBPF is composed of different components. While the verification overhead is significant and grows exponentially with the size of eBPF programs, it is a cost that happens once and can easily be amortized in long executions. JIT compilation does not add a significant overhead, as well as chaining tail calls to compose a more complex program. However, the execution of the eBPF extension adds a significant overhead (3x-5x) compared to a userspace Filter-Reduce implementation due to the significant performance impact deriving from buffer parsing and writing that is introduced with the use of helper functions.

RQ2: How to repurpose the eBPF toolchain and infrastructure in order to support programmable storage functions? eBPF has read, write and override primitives that can be used in order to access and modify the user data that travels in kernel buffers. This, paired with arithmetic operations and tail calls that increase the maximum instruction length allows to offload functionality in the eBPF extension to implement for example Filter and Reduce mechanisms. However, parsing overhead when mediated by helper functions is significant, and can easily overrun the maximum number of lines of code that a single eBPF program supports. For this reason, the overhead of eBPF-based processing is currently not leading to a performance improvement over userspace processing.

RQ3: What are the parameters to determine which functionalities should be offloaded to eBPF within the storage stack? When deciding on the type of functionalities that should be offloaded to storage it is important to keep in mind what are the limitations that eBPF introduces. The significant parsing overhead, especially on read, suggests that is is better to offload functionality that progressively reduces the data size, rather than performing transformations on data. In this way, subsequent tail calls with eBPF can gain from reduced buffer sizes. In addition to this, it is important to consider that parsing of data reduces the number of lines of code for data processing. For this reason, it is advisable to split the computation in small steps chained through tail calls, in order to overcome the limitation

8.2 Future Work

Within this thesis the focus was on the assessment and evaluation of the existing infrastructure. Now that its limitations and potential are known, there is plenty of room for future work and extensions.

First of all, given that the kprobe program type only offers helper-mediated access to the function call parameters, it would instead be convenient to create a separate program type for eBPF-based data processing that can have direct read access to the kernel buffers. In this way, processing would become more similar to what is available in the networking stack and allow to mitigate the significant performance penalty that comes from the buffer parsing in the current infrastructure.

A second important point of work is adding support for floating point and string operations within the Linux kernel, in order to expand on the possible use cases of the prototype. At the moment only arithmetic operations are allowed, and there is very limited support for string operations that need to be split in char-wise parsing. Adding native or helper-mediated support for these two use cases would greatly increase the applicability of the prototype and its potential use cases.

A third point for future work is focused on the deployment scenario for the prototype. Since the primary driver and motivation for this work is the datacenter scenario, an important part of future work is to expand the system for a distributed system scenario and systematically re-evaluate the gains. This is especially significant to do in the moment when the previous changes have taken place, since this thesis highlighted that it is currently not convenient to offload functionality to eBPF rather than performing it in userspace. This part of future work could be further expanded by bringing the prototype even closer to the storage medium. While this might mean losing generality that comes from the current prototype not being architecture specific, it would be a valuable contribution to quantify the gain of near-data processing as close as possible to the storage medium.

References

[1] Jaeyoung Do, Sudipta Sengupta, and Steven Swanson. **Programmable solid-state storage in future cloud datacenters**. *Communications of the ACM*, **62**(6):54–62, 2019. v, 2, 10, 13, 32, 63

[2] **Memory Performance in a Nutshell**. v, 12

[3] Steven McCanne and Van Jacobson. **The BSD Packet Filter: A New Architecture for User-level Packet Capture.** In *USENIX winter*, **46**, 1993. v, 5, 6, 13, 15, 16, 17, 18

[4] **What is eBPF?** v, 20, 21

[5] Aran Ali. **Here's What Happens Every Minute on the Internet in 2020**, Sep 2020. 1

[6] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. **A view of cloud computing**. *Communications of the ACM*, **53**(4):50–58, 2010. 1

[7] **Worldwide Public Cloud Services Market Totaled $233.4 Billion in 2019 with the Top 5 Providers Capturing More Than One Third of the Total, According to IDC**. 1

[8] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. **Network support for resource disaggregation in next-generation datacenters**. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, pages 1–7, 2013. 2, 9

[9] Thomas N Theis and H-S Philip Wong. **The end of moore's law: A new beginning for information technology**. *Computing in Science & Engineering*, **19**(2):41–50, 2017. 2, 10

[10] Zsolt István, David Sidler, and Gustavo Alonso. **Caribou: Intelligent distributed storage**. *Proceedings of the VLDB Endowment*, **10**(11):1202–1213, 2017. 2, 30

[11] Antonio Barbalace, Javier Picorel, and Pramod Bhatotia. **ExtOS: Data-centric Extensible OS**. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 31–39, 2019. 2, 27, 30, 65

[12] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. **Splinter: Bare-metal extensions for multi-tenant low-latency storage**. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 627–643, 2018. 2, 30, 64

[13] Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. **Near-data processing: Insights from a MICRO-46 workshop**. *IEEE Micro*, **34**(4):36–42, 2014. 2, 3, 13, 63

[14] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. **A reconfigurable fabric for accelerating large-scale datacenter services**. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24. IEEE, 2014. 4

[15] Oliver Pell and Oskar Mencer. **Surviving the end of frequency scaling with reconfigurable dataflow computing**. *ACM SIGARCH Computer Architecture News*, **39**(4):60–65, 2011. 4

[16] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. **A cloud-scale acceleration architecture**. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016. 5

[17] Torvalds. **https://github.com/torvalds/linux**. 5

[18] Joab Jackson. **How eBPF Turns Linux into a Programmable Kernel**, Oct 2020. 6

[19] Yair Levy and Timothy J Ellis. **A systems approach to conduct an effective literature review in support of information systems research.** *Informing Science*, **9**, 2006. 7

[20] Naim Kheir. *Systems modeling and computer simulation*, **94**. CRC Press, 1995. 7

[21] Alexandru Iosup, Laurens Versluis, Animesh Trivedi, Erwin Van Eyk, Lucian Toader, Vincent Van Beek, Giulia Frascaria, Ahmed Musaafir, and Sacheendra Talluri. **The atLarge vision on the design of distributed systems and ecosystems**. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1765–1776. IEEE, 2019. 7

[22] Richard R Hamming. *Art of doing science and engineering: Learning to learn.* CRC Press, 1997. 7

[23] Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. **A design science research methodology for information systems research**. *Journal of management information systems*, **24**(3):45–77, 2007. 7

[24] John Ousterhout. **Always measure one level deeper**. *Communications of the ACM*, **61**(7):74–83, 2018. 7, 47

[25] Per Nikolaj D Bukh. **The art of computer systems performance analysis, techniques for experimental design, measurement, simulation and modeling**, 1992. 7

[26] Gernot Heiser. **Systems Benchmarking Crimes**. 7

[27] Mark D Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E Bourne, et al. **The FAIR Guiding Principles for scientific data management and stewardship**. *Scientific data*, **3**(1):1–9, 2016. 7

[28] Emery D. Berger, Stephen M. Blackburn, Matthias Hauswirth, and Michael W. Hicks. **Systems Benchmarking Crimes**. 7

[29] Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. **Is big data performance reproducible in modern cloud networks?** In *17th USENIX Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 513–527, 2020. 7

[30] Sonja Bezjak, April Clyburne-Sherin, Philipp Conzett, Pedro Fernandes, Edit Görögh, Kerstin Helbig, Bianca Kramer, Ignasi Labastida, Kyle Niemeyer, Fotis Psomopoulos, Tony Ross-Hellauer, René Schneider, Jon Tennant, Ellen Verbakel, Helene Brinken, and Lambert Heller. *Open Science Training Handbook*. 04 2018. 7

[31] Giulia Frascaria. **BPF Tales, or Why Did I Recompile the Kernel to Average Some Numbers?** 8

[32] Giulia Frascaria. **Can eBPF Save Us from the Data Deluge? A Case for File Filtering in eBPF**. 8

[33] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. **Legoos: A disseminated, distributed {OS} for hardware resource disaggregation**. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 69–87, 2018. 9

[34] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. **Flash storage disaggregation**. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–15, 2016. 9

[35] **PCI SIG**. 9

[36] Luiz André Barroso and Urs Hölzle. **The datacenter as a computer: An introduction to the design of warehouse-scale machines**. *Synthesis lectures on computer architecture*, **4**(1):1–108, 2009. 11

[37] **Intel® Optane™ Technology**. `https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html`. Accessed: 2020-12-13. 11, 13

[38] Giulia Frascaria. **The Impact of Non-Volatile Memory on Modern Computer Systems**. 2019. 12

[39] John L Hennessy and David A Patterson. **A new golden age for computer architecture**. *Communications of the ACM*, **62**(2):48–60, 2019. 13

[40] **https://www.tcpdump.org/manpages/tcpdump.1.html**. 15

[41] **https://linux.die.net/man/8/arpwatch**. 15

[42] Jonathan Corbet. **A JIT for packet filters**. 16

[43] Jonathan Corbet. **BPF: the universal in-kernel virtual machine**. 16

[44] **filters.rst**. 16

[45] Matt Fleming. **A thorough introduction to eBPF**. 17

[46] Alexei Starovoitov. **bpf:add eBPF verifier**. 17

[47] Alexei Starovoitov. **extended BPF**. 17

[48] Alexei Starovoitov. **tracing: accelerate tracing filters with BPF**. 17

[49] **https://github.com/libbpf/libbpf**. 17

[50] **https://www.iovisor.org/technology/bcc**. 17, 31

[51] **https://github.com/iovisor/bpftrace/blob/master/docs/reference**$_{guide.md}$*.* 17

[52] **https://github.com/cilium/ebpf**. 17

[53] **man bpf**. 18

[54] **man bpf-helpers**. 18

[55] Jonathan Corbet. **BPF: what's good, what's coming, and what's needed**. 20

[56] Elena Zannoni. **BPF and Linux Tracing Infrastructure**. 20

[57] **https://github.com/torvalds/linux/blob/master/kernel/bpf/verifier.c**. 20, 21

[58] Greg Marsden. **BPF In Depth: The BPF Bytecode and the BPF Verifier**. 21

[59] Quentin Monnet. **Tools and Mechanisms to Debug BPF Programs**. 22

[60] David Calavera and Lorenzo Fontana. *Linux Observability with BPF: Advanced Programming for Performance Analysis and Networking*. O'Reilly Media, 2019. 22

[61] Glauber Costa. **How io_uring and eBPF Will Revolutionize Programming in Linux**. 24

[62] Joab Jackson. **How eBPF Turns Linux into a Programmable Kernel**. 24

[63] Thomas Graf. **eBPF - Rethinking the Linux Kernel**. 24

[64] Jake Edge. **Kernel runtime security instrumentation**. 24

[65] **New GKE Dataplane V2 increases security and visibility for containers**. 24

[66] **cilium.io**. 24

[67] Ashish Bijlani and Umakishore Ramachandran. **Extension framework for file systems in user space**. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 121–134, 2019. 27, 30, 65

[68] **TPC BENCHMARK H**. 32

[69] **TPC BENCHMARK DS**. 32

[70] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. **The HiBench benchmark suite: Characterization of the MapReduce-based data analysis**. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pages 41–51. IEEE, 2010. 32

[71] **ftrace, function tracer**. 36

[72] Brendan Gregg. *BPF Performance Tools*. Addison-Wesley Professional, 2019. 39

[73] Alexei Starovoitov. **tracing: attach eBPF programs to tracepoints/syscalls/kprobe**. 39

[74] Alexei Starovoitov. **Introduce BPF trampoline**. 39

[75] Vitaly Kuznetsov. **An introduction to timekeeping in Linux VMs**. 48

[76] Anurag Acharya, Mustafa Uysal, and Joel Saltz. **Active disks: Programming model, algorithms and evaluation**. *ACM SIGOPS Operating Systems Review*, **32**(5):81–91, 1998. 63

[77] Kimberly Keeton, David A Patterson, and Joseph M Hellerstein. **A case for intelligent disks (IDISKs)**. *Acm Sigmod Record*, **27**(3):42–52, 1998. 63

[78] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R Ganger. **Active disk meets flash: A case for intelligent ssds**. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 91–102, 2013. 63

[79] Devesh Tiwari, Simona Boboila, Sudharshan Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter Desnoyers, and Yan Solihin. **Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines**. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 119–132, 2013. 63

[80] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. **Willow: A User-Programmable SSD**. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 67–80, 2014. 64

[81] Jaeyoung Do, Yang-Suk Kee, Jignesh M Patel, Chanik Park, Kwanghyun Park, and David J DeWitt. **Query processing on smart SSDs: opportunities and challenges**. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1221–1230, 2013. 64

[82] Boncheol Gu, Andre S Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, et al. **Biscuit: A framework for near-data processing of big data workloads**. *ACM SIGARCH Computer Architecture News*, **44**(3):153–165, 2016. 64

[83] Zhenyuan Ruan, Tong He, and Jason Cong. **INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive**. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 379–394, 2019. 64