TÉCNICO LISBOA

Fine Grained Observability in Distributed Systems

Hugo André Pereira Rita

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor(s): Prof. Miguel Ângelo Marques de Matos

Examination Committee

Chairperson: Prof. Alberto Abad Gareta
Supervisor: Prof. Miguel Ângelo Marques de Matos
Member of the Committee: Prof. Vinícius Vielmo Cogo

October 2024

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

I am grateful to Instituto Superior Técnico for the resources and academic environment provided. Special thanks to my advisor, Prof. Miguel Matos, for guidance, and to my dissertation committee for their feedback.

This work was partially funded by Fundação para a Ciência e Tecnologia (FCT), using national funds as part of the projects INESC-ID UIDB/50021/2020, Ainur (financed by the OE with ref. PTDC/CCI COM/4485/2021) and Composable's bilateral project ScalableCosmosConsensus.

I appreciate the collaboration and support from other research colleagues at Instituto Superior Técnico and Universidade do Minho, especially Sebastião Amaro, Prof. João Paulo, and Tânia Esteves.

A special thank you goes to my friends, Bernardo Castiço, Pedro Pereira and Gonçalo Rodrigues. They accompanied me throughout the whole bachelor's and master's degree journey, their support and companionship were a huge factor in all the success I achieved.

Also thank you to my family, their support was essential to complete this academic phase, finally, I want to thank my girlfriend for her unconditional support, encouragement, and understanding throughout this journey, you are the most important people in my life.

Resumo

A necessidade de alcançar observabilidade e desempenho ótimos em sistemas distribuídos é fundamental porque falhas ou ineficiências impactam diretamente a experiência do utilizador e o uso eficiente de recursos. Atualmente, vários desafios dificultam este objetivo, tornando díficil identificar e resolver *bottlenecks*, problemas de latência e erros presentes em vários componentes interligados. A importância deste problema é salientada pelo papel crucial que os sistemas distribuídos desempenham na computação moderna, onde ineficiências afetam diretamente a experiência do utilizador e a utilização de recursos. Em resposta a estes desafios, várias ferramentas foram desenvolvidas para auxiliar no rastreamento de sistemas distribuídos, cada uma com as suas características e capacidades. Estas ferramentas são concebidas para ajudar a capturar e analisar as interações complexas dentro de ambientes distribuídos, fornecendo detalhes cruciais para diagnosticar e resolver problemas de desempenho. No entanto, o grande número de ferramentas disponíveis pode ser avassalador, e não existe um estudo sistemático das caracteristicas e compromisos das ferramentas existentes. Dada esta realidade, há uma necessidade crítica de orientações mais claras sobre como estas ferramentas de rastreamento funcionam e quais os problemas específicos que conseguem resolver. Cada ferramenta tem os seus pontos fortes, e compreender quando e como utilizá-las pode melhorar significativamente o desempenho e a fiabilidade do sistema. Sem o conhecimento adequado, estas ferramentas podem não oferecer os benefícios esperados, levando a uma observabilidade subótima e a uma resolução de problemas prolongada. Assim, é necessária uma abordagem mais estruturada na seleção e utilização das ferramentas de rastreamento distribuído. Isso implica compreender as capacidades técnicas de cada ferramenta e alinhá-las com as necessidades específicas do sistema distribuído em questão. Ao fazê-lo, as organizações podem melhorar a sua capacidade de monitorizar e otimizar os seus sistemas, de modo a obter uma melhoria de desempenho e maior satisfação do utilizador. Neste trabalho, realizámos uma análise abrangente das frameworks eBPF, fornecendo uma comparação detalhada em várias dimensões, como desempenho, facilidade de uso, requisitos de implementação e consumo de recursos. Ao avaliar as frameworks BCC, libbpf, Cilium, Aya, bpftrace e Eunomia, esclarecemos os melhores casos de uso para cada uma e destacámos os compromissos envolvidos na sua aplicação.

Palavras-chave: Observabilidade, Sistemas Distribuídos, Rastreamento.

Abstract

The need to achieve optimal observability and performance across distributed systems is paramount because failures or inefficiencies directly impact the user experience and the efficient use of resources. Currently, several challenges impede this goal, it is arduous to identify and address bottlenecks, latency issues, and errors seamlessly across numerous interconnected components. The vital role underscores the significance of this problem distributed systems play in contemporary computing, where inefficiencies directly impact user experience and resource utilization. In response to these challenges, various tools have been developed to aid in distributed system tracing, each with its features and capabilities. These tools are designed to help capture and analyze the complex interactions within distributed environments, providing crucial insights for diagnosing and resolving performance issues. However, the large number of available tools can be overwhelming, and there is no systematic study of the characteristics and commitments of the existing tools. Given this landscape, there is a critical need for clearer guidance on how these tracing tools function and what specific problems they are best suited to address. Each tool has its strengths, and understanding when and how to use them can significantly improve system performance and reliability. Without proper knowledge, these tools may not deliver the expected benefits, leading to suboptimal observability and prolonged troubleshooting processes. Thus, a more structured approach to selecting and using distributed tracing tools is necessary. This involves understanding the technical capabilities of each tool and aligning them with the specific needs of the distributed system in question. By doing so, organizations can enhance their ability to monitor and optimize their systems, ultimately improving performance and user satisfaction. In this work, we have conducted a comprehensive analysis of eBPF frameworks, providing a detailed comparison across multiple dimensions such as performance, ease of use, deployment requirements, and resource consumption. By evaluating the frameworks BCC, libbpf, Cilium, Aya, bpftrace, and Eunomia, we have clarified the best use cases for each and highlighted the trade-offs involved in their application.

Keywords: Observability, Distributed Systems, Tracing.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Distributed systems involve the coordination and operation of multiple interconnected nodes that work collaboratively to achieve a common goal. This architectural paradigm has gained immense popularity due to its ability to enhance scalability, fault tolerance, and resource efficiency. The shift towards microservices, cloud computing, and decentralized networks has accelerated the adoption of distributed systems, reflecting a departure from traditional monolithic architectures.

This evolution has introduced a new set of challenges, particularly in the realms of performance monitoring and issue diagnosis. The inherent complexity of distributed systems, with their diverse and dynamic interactions among components, makes it increasingly difficult to identify bottlenecks, diagnose faults, and optimize performance.

Observability is a cornerstone of effective system management, offering valuable insights into the internal workings of a system. By closely monitoring performance metrics, deviations from expected behavior can be quickly identified and addressed, preventing potential service disruptions. Secondly, observability is essential for effective issue diagnosis. In the event of performance degradation or system failures, insights into the distributed system's internal workings are necessary to pinpoint the root causes of the issue.

Moreover, observability is instrumental in optimizing resource utilization. By understanding how resources are allocated and utilized across the distributed environment, one can more easily make decisions to enhance efficiency, scale resources as needed, and minimize operational costs. Finally, in the context of user experience, observability plays a pivotal role. Since it enables organizations to proactively identify and address issues that may impact the end-user, such as slow response times, service interruptions, or incorrect system behavior, observability tools can be useful in ensuring a positive and seamless user experience. This is why observability in distributed systems is fundamental.

Since the existence of distributed systems, effectively monitoring and diagnosing performance

issues has always been a struggle. Traditional tools, often designed for simpler, monolithic architectures, struggle to adapt to the decentralized and dynamic nature of distributed systems. The limitations of traditional tools in distributed systems primarily revolve around their centralized design, inability to adapt to dynamic environments, challenges in handling complex interactions, scalability issues, the potential introduction of latency, and a lack of adaptability to modern distributed technologies.

The nature of distributed systems and the demand for higher levels of performance and reliability necessitate innovative approaches and tools, prompting the development of more advanced systems capable of overcoming these limitations and addressing the unique challenges posed by distributed systems. The challenges stem from the difficulty of adapting to the decentralized and dynamic nature of distributed architectures. Traditional tools, primarily designed for centralized control and static environments, struggle to handle the interactions and dynamic resource allocations across diverse nodes. As distributed systems scale, conventional tools struggle to manage the increasing data from multiple nodes, with challenges in handling communication patterns, dynamic changes, and resource allocations. This highlights the need for more advanced tracing systems.

One common approach to distributed system tracing was the usage of logging mechanisms, where applications and services generated log files containing information about their activities. While it seems like a straightforward approach, this can result in a flood of data that could be challenging to analyze, due to the large data volume, especially in large-scale distributed systems.

Another method involved the use of instrumentation, where developers manually inserted code into their applications to gather specific performance metrics or trace the flow of execution. While this approach is better than logging, because it allows targeted data collection, it comes with the drawback of increased code complexity and potential interference with the normal operation of the system.

For tracing in networked environments, packet tracing tools, such as Wireshark[1], can capture and analyze network traffic, providing insights into communication patterns between nodes. However, such tools were often too low-level for application-level tracing and had several limitations, as they can generate overwhelming amounts of data in large-scale systems, making it difficult to trace and analyze traffic across numerous distributed nodes efficiently.

Overall, traditional tracing methods often lacked the agility, scalability, and precision needed for the complexities of modern distributed systems.

[1] `https://www.wireshark.org/`

The introduction of eBPF (extended Berkeley Packet Filter)[2] opened new possibilities for tracing. eBPF allows for the safe and efficient execution of custom programs in the kernel space, enabling dynamic tracing, monitoring, and networking capabilities. It provides a flexible and programmable way to capture, filter, and analyze events at various levels within the kernel, enabling more granular and precise observation of system activities.

Moreover, eBPF allows for real-time visibility into the kernel, its lightweight nature makes it well-suited for high-performance environments, minimizing runtime overhead and making it possible to capture detailed information without adversely impacting system performance. This adaptability proved crucial in the dynamic landscape of distributed systems, where traditional tracing methods struggled to keep pace with the evolving nature of applications and services.

Despite the significant strides made by eBPF in enhancing tracing capabilities within the Linux kernel, some challenges persist. While eBPF can capture every event needed to perform analysis, this behavior alone is not enough, there is still the need to send the traced information to user space which leads to another challenge: how to effectively send this information to user space for further analysis without losing precision. Moreover, the eBPF verifier has multiple requirements a program must comply with to be executed, to ensure safe and secure execution, minimizing the range of things we can do when tracing a system, more on these restrictions in §2.2.

Despite these advances, there are still notable challenges encountered in the utilization of eBPF. While essential for preventing harmful code, the verification process of the eBPF verifier can result in limitations that impact the flexibility of eBPF programs. Furthermore, the transmission of information from the kernel to user space, a crucial aspect of eBPF-based tracing, is marred by instances of information loss, especially when sending both the traced content and context. This transmission bottleneck can compromise the completeness and accuracy of the traced data.

While eBPF offers tremendous potential for observability, its complexity makes it difficult to work with directly. Writing raw eBPF programs often requires deep knowledge of kernel internals, system architecture, and low-level programming. In addition to the previously mentioned challenges, the intricacies of compiling, verifying, and deploying eBPF programs add another layer of difficulty. As a result, eBPF development can become a time-consuming and error-prone task.

To address these issues, various tools and frameworks have emerged that abstract away many of the low-level details, making it easier to interact with eBPF. These frameworks streamline the process of writing, loading, and attaching eBPF programs to kernel hooks, providing higher-

level APIs, pre-built tools, and improved debugging capabilities. They allow users to focus more on the functionality they want to implement rather than the technical complexities of eBPF itself. However, despite the availability of these tools, a major challenge remains, the lack of clear guidance on how to use them effectively. Understanding these nuances is crucial for selecting the right tool that aligns with the unique needs of their distributed systems. The choice of an eBPF framework involves not only assessing technical features but also considering trade-offs such as performance impact and ease of integration. Some frameworks may be better suited for low-latency environments, while others excel in providing detailed security audits or advanced performance metrics. This gap in best practices and tool selection often leads to underutilization or misuse of eBPF's capabilities, preventing its full potential from being realized in terms of achieving comprehensive observability, networking, and system monitoring.

Furthermore, no definitive benchmarking or comparison is available to help determine which tool is best suited for specific scenarios. This lack of standardized guidance introduces additional challenges, especially for newcomers to eBPF, who are left to experiment with different tools and frameworks to find the best fit for their applications.

The objective of this thesis is to conduct a systematic analysis of eBPF frameworks, providing insights into their efficiency, usability, and suitability for different use cases. This includes an exploration of the various tools and frameworks available for working with eBPF, an evaluation of their strengths and limitations, and the identification of scenarios where each framework is most appropriate. Additionally, the thesis will examine different types of probing mechanisms, such as Kprobes, Uprobes, Tracepoints, and XDP, explaining their functionality and offering guidance on their optimal use cases.

By conducting this analysis, our goal is to address the current lack of information and best practices regarding eBPF tools, ultimately offering a clearer understanding of how to effectively utilize eBPF for system observability, networking, and security while avoiding common pitfalls. Through both theoretical exploration and hands-on experimentation, this thesis aims to create a practical guide for users seeking to implement eBPF in an optimized, efficient, and reliable manner.

The remainder of the document is structured as follows. §2 delves into the background, strictly analyzing the state of the art and contextualizing our approach within existing research. §3 details the implementation we followed to conduct the results presented in §4, and finally §5 concludes with a summary of findings and reiterates the problem.

Chapter 2

Background

In distributed systems, the fundamental need for observability becomes paramount. The catchphrase "You can't improve what you can't measure" encapsulates the challenge distributed systems face. Without comprehensive means to measure, understand, and analyze the behavior of distributed systems, it becomes akin to navigating uncharted waters in the dark. The sheer complexity of these systems, with their interactions and dependencies, necessitates a profound need for observability tools that go beyond mere monitoring. Observability offers a crucial means to shed light on the performance, bottlenecks, and inefficiencies of distributed architectures.

Tracing in distributed systems involves capturing and analyzing information about the flow of requests and events across multiple components or services within a distributed architecture. It aims to provide insights into the performance, latency, and interactions between various system elements. It is crucial for understanding and troubleshooting complex interactions between services, identifying bottlenecks, improving overall system performance, and helps in diagnosing issues such as latency spikes, error propagation, and dependencies between services, contributing to better observability and operational efficiency in distributed environments.

Tracing systems are designed to provide insights into system performance by capturing function calls, resource utilization, and runtime behavior. All tracing systems share a core set of objectives aimed at addressing the challenges inherent in tracing environments. The key properties of an effective tracing system are: i) low overhead, ensuring minimal impact on system performance during tracing activities, leading to overall system efficiency; ii) full expressiveness, essential for capturing a comprehensive range of events and behaviors; iii) extensibility, allowing adaptability to evolving system architectures and tracing requirements; iv) precision, enabling accurate and detailed data collection; and v) information representation, which involves presenting traced data clearly. These shared goals ensure that the benefits of tracing are realized without imposing undue strain on the traced systems or the underlying infrastructure.

2.1 Concepts

The eBPF ring buffer[1] is a circular data structure that efficiently manages a buffer for communication between eBPF programs and user space. It enables the asynchronous exchange of data, allowing eBPF programs to produce events and user space applications to consume them at their own pace. The eBPF ring buffer is a real-time communication mechanism, providing a data transfer channel between eBPF programs and user space components

eBPF maps[2] are structured key-value data structures facilitating communication between eBPF programs and user space in a scalable manner. They serve as a bridge, allowing user applications to interact with eBPF programs. eBPF maps can be queried and updated by both user space and eBPF programs.

Kernel space is where the kernel (i.e., the core of the operating system) executes and provides its services. It has direct access to hardware resources, executing critical functions like process scheduling and memory management. Protected from direct user access, interactions occur through controlled interfaces like system calls.

User space is the non-privileged portion of a computer's memory where user applications and their data reside. Operating at a lower privilege level than the kernel, user space has restricted access to hardware resources. Applications are isolated from each other, and their interaction with the system is mediated through system calls, providing a secure interface to request services from the kernel.

Helper functions[3] in the kernel were made to generalize the programmability of eBPF. The reason for such helper functions is that eBPF programs cannot call normal kernel functions since they will be limited to the kernel version and complicate the programs' compatibility. Some of the most used helper functions are random number generators, time, data, and eBPF map access.

BPF CO-RE[4] (Compile Once, Run Everywhere) is a feature introduced in the eBPF ecosystem that enables eBPF programs to be compiled once and run across different kernel versions without requiring recompilation for each target environment. Traditionally, eBPF programs were closely tied to the specific kernel version they were compiled against, meaning one had to recompile them for different kernel versions or distributions, which added complexity and maintenance overhead. BPF CO-RE solves this by using BPF Type Format (BTF), which provides metadata about the kernel's types and structures, allowing the eBPF program to adapt

[1] `https://docs.kernel.org/6.6/bpf/ringbuf.html`
[2] `https://docs.kernel.org/bpf/maps.html`
[3] `https://man7.org/linux/man-pages/man7/bpf-helpers.7.html`
[4] `https://docs.ebpf.io/concepts/core/`

at runtime to differences between kernels. This significantly enhances the portability of eBPF programs, making it easier to deploy and manage them across a wide range of Linux systems, from development environments to production servers, without compatibility issues.

2.2 eBPF

The extended Berkeley Packet Filter (eBPF)[2] is a technology designed to enhance the Berkeley Packet Filter (BPF), which originally emerged as a solution for efficient packet inspection. eBPF allows to write custom programs that can be safely and efficiently executed within the operating system kernel, particularly in the Linux kernel. Unlike traditional BPF, which was mainly designed for filtering network packets at the boundary between kernel space and user space, eBPF extends its functionality far beyond packet filtering. eBPF can attach to almost any part of the Linux kernel, intercepting system calls, monitoring events, and modifying packet data in real-time. Its primary advantage lies in its ability to dynamically execute custom code within the kernel without the need for modifying or recompiling the kernel itself. This provides flexibility for monitoring, debugging, and enhancing system performance.

The eBPF architecture begins with programs written in user space, typically in C or other supported languages. These programs are then compiled into eBPF bytecode, which can be executed within the Linux kernel. Before execution, the bytecode must pass through the eBPF verifier, which ensures that the program adheres to strict safety rules, such as avoiding loops and ensuring safe memory access. This prevents the program from crashing or compromising the system. Once verified, the bytecode undergoes further optimization through the Just-In-Time (JIT) compiler, which translates it into native machine code for efficient execution. The bytecode is then executed within a virtual machine (VM) embedded in the Linux kernel, allowing for dynamic execution without the need for kernel recompilation. eBPF programs can attach to various kernel subsystems, such as tracing, packet filtering, system calls, and event monitoring, enabling flexible monitoring and real-time modification of kernel behavior. This architecture ensures that eBPF programs are both powerful and safe, providing valuable tools for system tracing, networking, and performance optimization.

Now we will do a more detailed view of each component, first, we will take a look into the verification stage, the Linux kernel includes an eBPF verifier that checks whether a program is safe to execute. This stage is crucial because eBPF programs, once loaded into the kernel, run with high privileges and could potentially crash the system or introduce vulnerabilities if not properly vetted.

The verifier checks several aspects:

- Privileges: Ensures the program has the necessary permissions to run.
- Run to Completion: eBPF programs must terminate, so loops are limited to ensure they have guaranteed exit conditions.
- Memory Access: Programs are restricted from accessing memory outside of allocated regions.
- Size and Complexity: eBPF programs are subject to size and complexity limitations, ensuring they are lightweight and efficient, so, programs should have finite execution paths.
- Crash Prevention: The verifier also checks for potential program crashes by simulating different execution paths.

The verification process ensures that eBPF programs will neither consume excessive kernel resources nor expose the system to attacks or instability.

After the verification process, eBPF programs are compiled into native machine code using the Just-in-Time (JIT) compiler[5]. This step is essential for performance optimization because kernel space resources are limited. The JIT compiler translates eBPF bytecode into native instructions that can be executed directly by the CPU, minimizing overhead and improving runtime efficiency.

At the time eBPF was introduced, it came with many improvements over other tools used by the state of the art, mainly its flexibility and extensibility, allowing it to attach to various hooks throughout the kernel, such as network packet handling, system call monitoring, and event tracing. It enables real-time execution and safety, as eBPF programs can be injected and executed at runtime without requiring system restarts. Additionally, the kernel verifier ensures that these programs are safe to execute, preventing system crashes and providing a high level of reliability. eBPF also offers significant performance optimization over traditional tools. Its in-kernel execution model eliminates the overhead of context switches between kernel and user space, while JIT compilation ensures efficient execution, minimizing resource usage and reducing latency. Another key feature of eBPF is its data structures, known as eBPF maps, which facilitate the storage and exchange of data between the kernel and user space. Although eBPF imposes constraints on program size and complexity, mechanisms like tail calls and function calls allow large programs to be divided into smaller subprograms. This helps manage complexity and improves performance by reducing unnecessary code execution, promoting scalability and expandability.

[5] `https://en.wikipedia.org/wiki/Just-in-time_compilation`

eBPF proves valuable in several key areas. First, in tracing and observability, eBPF excels by providing real-time monitoring of system events. It can track CPU, memory, and I/O performance metrics, which aids in diagnosing bottlenecks and optimizing resource utilization. In networking, eBPF offers the ability to intercept and manipulate network packets at different stages of the network stack, providing fine-grained control over network traffic. This makes it essential for packet filtering, load balancing, and traffic shaping. Security is another crucial area where eBPF contributes by detecting and preventing malicious actions, such as unauthorized system access or abnormal network traffic. It blocks these threats in real time and helps enforce security policies to ensure safe process execution. In storage, eBPF allows for real-time monitoring of disk I/O operations, helping to identify performance bottlenecks and optimize interactions with storage systems. It also helps secure storage infrastructure and enforce data integrity. eBPF has been explored for improving file system performance, reducing I/O latency, and enhancing storage efficiency through performance tuning. Additionally, eBPF enables offloading tasks like network packet processing directly to the kernel, reducing the overhead of moving data between kernel and user space. Finally, in scheduler optimization, eBPF can monitor task execution times, CPU utilization, and other scheduling metrics to improve task scheduling. It can implement custom scheduling policies based on real-time metrics, allowing dynamic and intelligent scheduling tailored to specific workloads.

2.2.1 Several Scopes

The state-of-the-art, further described in §2.5.1 can be split by what each system goal is, those are tracing of systems, storage, network, security, scheduling, and hardware

The tracing-focused systems can be divided into two scopes of tracing at the application level and the container level, each with its focus. In application-level tracing, the focus is directed towards understanding and optimizing the performance of individual software applications. These systems delve into the workings of applications, offering insights into function calls, resource utilization, and runtime behavior. Within container tracing systems, a focus emerges on the network dynamics within containerized environments. Container-level tracing systems provide an understanding of communication patterns and network bottlenecks. We focus on how these systems contribute to efficiency, security, and observability, particularly in potential bottlenecks within containerized infrastructures. A limitation arises when tracing must extend beyond the container layer to span multiple nodes or diverse environments.

In storage systems, the focus is to enhance both the performance and adaptability of storage subsystems by enabling in-kernel processing of storage-related operations. Traditional storage

workflows require data to be moved between user space and kernel space, leading to increased latency and resource consumption. By executing storage functions directly within the kernel, closer to the hardware, eBPF reduces this overhead, allowing for faster data processing and more efficient use of system resources. Additionally, eBPF's flexibility allows for dynamic modifications and optimizations tailored to specific workloads or hardware capabilities, such as Non-Volatile Memory Express (NVMe) devices, thus optimizing the performance of modern storage architectures.

In the realm of networking, eBPF systems aim to significantly improve network management by providing real-time control over network traffic directly within the kernel. Traditional networking stacks can suffer from inefficiencies, especially when dealing with modern data center demands like high bandwidth and low latency. eBPF allows for dynamic manipulation of network packets, enabling advanced features such as load balancing, traffic filtering, and congestion control without the need for user-space processing. This results in reduced latency, improved throughput, and the ability to respond quickly to changing network conditions.

The goal of eBPF systems in security is to empower security mechanisms with the ability to operate directly within the kernel, providing protection. eBPF can be used to create custom security policies that monitor system behavior, detect anomalies, and enforce access controls without the overhead associated with traditional user-space security solutions. By being integrated with the kernel, eBPF-based security tools can intercept and analyze system calls, and network traffic. Furthermore, eBPF enables the creation of lightweight, high-performance security applications that can be easily updated and adapted to new threats, offering robust protection for both individual systems and large-scale infrastructures.

In terms of scheduling, eBPF systems aim to provide enhanced insights and control over how tasks are managed and executed within the kernel. The Linux Completely Fair Scheduler (CFS) and other scheduling mechanisms can be fine-tuned using eBPF to ensure fair distribution of CPU resources, optimize for specific workload characteristics, and respond to system load in real-time. This capability is particularly valuable in high-performance computing environments or real-time systems where the responsiveness and efficiency of task scheduling directly impact overall system performance. By allowing custom scheduling policies to be implemented at the kernel level, eBPF helps achieve a better balance between performance, fairness, and resource utilization.

For hardware, eBPF systems focus on leveraging the capabilities of specialized devices, such as Network Interface Cards (NICs) and computational storage devices, by offloading specific tasks from the CPU to the hardware itself. This approach reduces the processing load on

the host system, enabling it to handle more complex or concurrent tasks without performance degradation. By allowing eBPF programs to run directly on hardware components, such as NICs, the system can achieve faster data processing, lower latency, and more efficient resource usage. This is particularly useful in environments where high data throughput and low latency are critical, such as in data centers or edge computing scenarios. Additionally, eBPF's hardware offloading capabilities also enable the possibility of implementing custom processing pipelines and optimizations that are closely aligned with the specific characteristics of the underlying hardware, resulting in a more efficient and scalable system.

2.3 Types Of Attaching To System Calls

In the Linux operating system, the ability to monitor, trace, and manipulate system behavior with eBPF programs is achieved through several mechanisms, each with its unique characteristics and use cases. The main 3 mechanisms are uprobes, kprobes, and tracepoints. Understanding these mechanisms is crucial, as they offer ways to observe and influence the behavior of both user-space applications and the kernel itself.

Uprobes (User-space Probes) is a mechanism in the Linux kernel that allows to dynamically insert probes into user-space applications. Uprobes work by allowing you to set breakpoints or instrumentation points at specific locations in user-space binaries or libraries. When the application reaches one of these points, control is transferred to a handler function in the kernel where the user can examine the state of the application, collect data, or modify behavior. The switch from user space to the kernel handler adds overhead to the overall process.

Kprobes (Kernel Probes) provide similar functionality to uprobes but are designed for use within the kernel space. Kprobes allow the dynamic insertion of instrumentation points into the running kernel, making it possible to monitor or trace kernel functions without needing to modify the kernel source code or reboot the system. This is particularly valuable for debugging and monitoring kernel-level operations, such as system calls, interrupt handlers, or other low-level kernel functions.

Tracepoints[6] are predefined static points within the Linux kernel code that provide a way to hook into specific events or states in the kernel. Unlike uprobes and kprobes, which allow the dynamic insertion of probes, tracepoints are typically hard-coded into the kernel source and are designed to be lightweight and efficient. They are intended for use in performance monitoring, logging, and debugging. When a tracepoint is hit, it can trigger a registered handler or log event data, enabling detailed tracing of kernel activities. Tracepoints are widely used because

[6] `https://www.kernel.org/doc/Documentation/trace/tracepoints.txt`

they are efficient and can be enabled or disabled at runtime with minimal overhead. They provide valuable insights into kernel performance and behavior without the need for custom instrumentation.

2.4 eBPF Frameworks

Transitioning from the eBPF concepts in the §2.1, our focus now shifts to the practical implementation of these concepts through eBPF frameworks. These frameworks are essential because they enable the integration and execution of custom eBPF programs within the Linux kernel, by providing abstractions and high-level APIs for the development and deployment of eBPF programs, they simplify the creation of eBPF-based applications. However, their complexity and sometimes incomplete documentation make them challenging to grasp fully. eBPF frameworks present trade-offs in terms of ease of use versus functionality, aiming to strike a balance between complexity and usability, ultimately fostering a more accessible and robust platform for network and system-level innovations.

Next follows a description of each relevant eBPF framework in the state-of-the-art.

2.4.1 BPF Compiler Collection

BCC (BPF Compiler Collection)[7] is a toolkit for creating efficient kernel tracing and manipulation programs using eBPF. It was introduced to simplify the process of writing and running eBPF programs, particularly for system observability, diagnostics, and performance monitoring in Linux environments. BCC leverages the capabilities of eBPF, which was first added to Linux in version 3.15 and provides a higher-level interface for working with eBPF, abstracting much of the complexity involved in dealing directly with eBPF bytecode.

BCC uses a combination of kernel instrumentation written in C, and a C wrapper around LLVM, which compiles C code into eBPF bytecode that can be loaded into the kernel. On top of this low-level foundation, BCC provides a Python and Lua front-end, allowing it to interact with the kernel more easily. This combination of C and Python allows users to quickly create, test, and deploy eBPF programs for tasks like tracing system calls, monitoring network traffic, and profiling system performance. One of BCC's standout features is its collection of pre-built tools and examples, these tools are designed for common use cases like measuring CPU usage, tracking network packets, or diagnosing I/O latency.

BCC abstracts away the complexity of working directly with eBPF, providing a user-friendly interface that allows one to write more readable, high-level code while still accessing the powerful

[7] `https://github.com/iovisor/bcc`

capabilities of eBPF. This abstraction, coupled with its extensive tooling and documentation, has made BCC one of the most popular frameworks for eBPF development.

2.4.2 Cillium

Cilium[8] is an open-source, eBPF-powered networking and security framework specifically designed to address the needs of containerized environments, particularly those running on Kubernetes[9]. By leveraging eBPF, Cilium provides a powerful, high-performance solution for tasks such as packet filtering, load balancing, and network visibility.

Cilium's strength lies in its ability to bring eBPF's flexibility and efficiency to container workloads, complex networking, and security challenges. Traditional networking solutions often struggle with microservices architectures, dynamic and highly distributed nature, where workloads are constantly being spun up or down across multiple nodes. Cilium addresses these challenges by allowing the definition of custom networking and security policies using eBPF, ensuring that data flows between containers are handled efficiently and securely, even at scale.

One of the key features of Cilium is its high-level API, which abstracts much of the complexity of eBPF, enabling users to express complex networking rules without needing in-depth knowledge of kernel programming. Additionally, Cilium integrates seamlessly with Kubernetes, making it an attractive choice for organizations deploying microservices in containerized environments. Cilium is written in Go, providing a pure Go library for interacting with eBPF. By bringing the advantages of eBPF to Kubernetes and other containerized environments, Cilium provides a scalable, secure, and efficient solution for modern networking challenges.

2.4.3 Libbpf

Libbpf[10] is a C-based library designed to simplify the integration of eBPF programs into the Linux kernel. Libbpf offers both high-level and low-level APIs for user-space programs. The high-level APIs abstract away many of the intricate details of eBPF program management, while the low-level APIs provide more granular control for fine-tuned management of their eBPF programs. These APIs facilitate the interaction between user-space applications and the kernel by providing access to essential BPF-side features, such as BPF maps and BPF helpers, both described in §2.1.

One of libbpf's standout features is its support for BPF CO-RE (Compile Once, Run Everywhere), a powerful mechanism that allows to write portable eBPF programs. Traditionally,

[8] `https://github.com/cilium/ebpf`
[9] `https://kubernetes.io/`
[10] `https://github.com/libbpf/libbpf`

eBPF programs had to be compiled for specific kernel versions, which created compatibility challenges across different environments. With BPF CO-RE, users can compile eBPF programs once and run them on different kernel versions without modification. This portability is a significant advantage in environments where multiple kernel versions are in use, such as large-scale production systems or cloud-based infrastructures, as it simplifies deployment and reduces maintenance overhead.

2.4.4 Eunomia-bpf

Eunomia-bpf[11] is a dynamic loading library and compile toolchain framework designed to build, deploy, and distribute eBPF programs.

At its core, Eunomia-bpf is built on top of libbpf. However, Eunomia-bpf extends libbpf's functionality by simplifying the process of building eBPF tools, allowing one to more easily package, distribute, and run eBPF programs. One of the standout features of Eunomia-bpf is its ability to package eBPF programs in JSON format or as WebAssembly (WASM) modules. This enables to distribute and execute eBPF programs in a highly portable and flexible manner. Moreover, Eunomia-bpf automatically handles the complexity of exposing kernel data and facilitating interactions between kernel-space eBPF programs and user-space applications. Eunomia-bpf also integrates with a WASM runtime, providing an innovative approach for interacting with eBPF programs. This integration allows developers to write eBPF code for the kernel and expose data to user space using WASM, a lightweight, cross-platform runtime.

2.4.5 Aya

Aya[12] is an eBPF library built from the ground up in Rust. Unlike other popular eBPF frameworks such as libbpf§2.4.3 or BCC§2.4.1, Aya is completely independent of these libraries, offering a Rust-native approach to eBPF development. By leveraging Rust's memory safety guarantees and performance, Aya provides an efficient and secure environment for creating and deploying eBPF programs.

One of Aya's advantages is the integration of Rust's ecosystem, particularly through Cargo, Rust's package manager. Cargo streamlines project management, building, testing, and debugging processes, making the development workflow more efficient and developer-friendly. With Cargo, developers can easily manage dependencies, and automate builds, all while benefiting from Rust's compile-time safety checks.

[11] `https://github.com/eunomia-bpf/eunomia-bpf`
[12] `https://github.com/aya-rs/aya`

Aya also supports the Compile-Once, Run-Everywhere (CO-RE) mechanism, this enhances the portability of Aya-based eBPF programs, simplifying deployment across diverse environments and making it easier to maintain and distribute these programs.

2.4.6 Bpftrace

bpftrace[13] is a high-level tracing language designed specifically for Linux, offering a simplified interface for leveraging the power of eBPF. By abstracting away much of the complexity associated with writing eBPF programs, bpftrace makes it easier to create custom, on-the-fly tracing and diagnostic tools for monitoring system behavior. Its syntax is inspired by familiar languages such as awk and C, drawing on the legacy of predecessor tracing tools like DTrace[4] and SystemTap[1], which similarly aim to provide powerful observability capabilities with minimal effort.

bpftrace uses LLVM as its backend for compiling scripts into eBPF bytecode, which can then be loaded and executed in the kernel. This compilation step ensures that the scripts are optimized for performance while still providing the flexibility of high-level scripting. bpftrace also integrates with libbpf and BCC to interact with the Linux eBPF subsystem. These libraries provide the necessary infrastructure for managing eBPF programs, attaching them to various kernel hooks, and collecting data from the kernel.

One of bpftrace's key strengths is its ability to provide ad-hoc tracing without the need for complex, pre-compiled programs. Users can write and execute bpftrace scripts on the fly to trace specific system events.

2.5 Related Work

In the next section, we briefly introduce a set of relevant systems that use eBPF as their main tool.

2.5.1 State-of-the-art Descriptions

PHOEBE [20] is a fault injection framework for reliability evaluation against system call invocation errors. The key of PHOEBE [20] is that, after a phase of system tracing, it synthesizes and injects realistic system call errors, meaning that the injected errors are based on errors that naturally happen, thus contributing to overcoming the most critical problems in the targeted system.

[13] `https://bpftrace.org/`

CAT [7] is a non-intrusive content-aware tracing and analysis framework that, through a similarity-based approach, can comprehensively trace and correlate the flow of network and storage requests from applications. The key contribution of CAT [7] is the content-aware tracing system.

DIO [8] is a generic tool for observing I/O interactions between applications and in-kernel storage systems. It allows us to diagnose potential performance, dependability, and correctness issues. Through a pipeline that automates the process of tracing, filtering, correlating, and visualizing system calls, and by enriching the information provided with additional context, DIO helps users observe I/O issues.

Zpoline [11] is a system call hook mechanism for x86-64 CPUs, that can exhaustively hook system calls at a low overhead without overwriting instructions that are supposed not to be modified. It is a practical way of transparently applying non-verified user-space behavior to system calls.

A protocol-independent container network observability analysis system based on eBPF [13], further referred to as PICNO, non-intrusively collects user application interaction information from L7/L4 layer container network protocols in a cloud-native environment. The information then goes through an analysis phase, the system has a machine learning method used to analyze and diagnose the performance and problems of the application network, mainly through the analysis of timestamps and latency.

The terms L7 and L4 refer to different layers in the OSI (Open Systems Interconnection) model[14], which is a conceptual framework used to understand network protocols. In this context, networking protocols at L7 are high-level application protocols and provide a way for software applications to communicate over a network. Examples include HTTP, SMTP, and FTP. Meanwhile, L4 protocols are concerned with the underlying transport of data between containers, for example, TCP (Transmission Control Protocol) and UDP (User Datagram Protocol).

Lee et al. [12] further referred to as EPTM is an eBPF-based packet tracing method for latency measurement in the container overlay network. To efficiently detect the trace context on an HTTP payload, the trace context position is moved just behind the HTTP request line using a sidecar proxy. This tracing method gathers HTTP packets that have the trace context

[14] `https://www.cloudflare.com/learning/ddos/glossary/open-systems-interconnection-model-osi/`

and measures the latency using eBPF.

Inspektor Gadget [5] is a toolset tailored for debugging and inspecting Kubernetes[15] resources and applications. Functioning as a collection of gadgets, this toolkit streamlines the debugging process by providing a set of specialized tools designed to address different aspects of a Kubernetes environment. Its integration with the eBPF technology allows for the packaging, deployment, and execution of eBPF programs within a Kubernetes cluster, gaining low-level insights from the Linux kernel.

Using eBPF for Database Workload Tracing [6] is an exploratory study with a focus on replacing Database management systems, further referred to as DBMS, with eBPF tracing. Database Management systems have their native implementation to trace requests made to a database, mainly logging, but this implementation is too cumbersome to the database itself, so the authors found that eBPF programs were more or equally performant than the DBMS native implementation.

Nahida [16] is a system designed to trace every HTTP request made by a target application, providing detailed insights into the application's end-to-end flow. Utilizing eBPF, Nahida intercepts each HTTP request and tags it with a unique ID. By tracking the journey of these tagged IDs throughout the system, Nahida offers a view of the application's internal operations, enabling the identification of the overall behavior of the application from start to finish. This approach provides real-time, granular visibility into request flows, making it a powerful tool for monitoring and debugging complex distributed systems.

XRP [21] is a system designed to execute storage operations directly within kernel space, closer to the NVM (Non-Volatile Memory) device. By bypassing certain kernel system layers, XRP reduces the overhead associated with these operations, leading to more efficient storage function execution.

Electrode[23] focuses on optimizing the implementation of the Paxos Protocol [16] by executing critical operations directly in the kernel. This system addresses the inefficiency where a significant portion of CPU resources is consumed by user space-kernel transitions and traversing the kernel's networking stack during Paxos operations. To mitigate this, Electrode offloads the following operations to kernel space: message broadcasting, fast acknowledging, and wait-on-quorums.

DINT [24] is a system designed to optimize distributed transactions by reducing the overhead associated with the kernel networking stack, user-kernel context switching, and interrupt han-

[15] `https://kubernetes.io/docs/home/`
[16] `https://en.wikipedia.org/wiki/Paxos_(computer_science)`

dling. Traditional approaches to improve performance, such as kernel bypass techniques, often sacrifice security. DINT, however, leverages eBPF to run transaction processing logic directly within the early stages of the kernel networking datapath, bypassing much of the kernel stack and avoiding user-kernel transitions. This approach maintains the security and other beneficial properties of the kernel while significantly enhancing performance in distributed in-memory transactions.

eHDL [14] is a system focused on optimizing network packet processing by converting software programs into hardware designs for Network Interface Cards (NICs). As NIC speeds increase, there is a need for more efficient packet processing that does not overburden the host CPU. eHDL addresses this by automatically generating tailored hardware designs from eBPF/XDP programs, allowing software developers to leverage NIC hardware without deep hardware expertise. It functions as a bytecode-to-source compiler, converting unmodified eBPF bytecode into HDL (Hardware Description Language) and integrating the generated hardware pipeline into the NIC.

Delilah [10] is a system focused on enhancing computational storage, specifically within the context of NVMe devices, by utilizing eBPF. The motivation behind Delilah is to explore the potential of offloading computational tasks to storage devices, either through statically installed device-specific functions or dynamically downloadable functions from the host at runtime. The primary goal is to offload eBPF code onto actual computational storage devices, allowing for the evaluation of the overhead associated with eBPF function execution and the exploration of different design options. By doing so, Delilah aims to establish a foundation for future research and development in leveraging eBPF to improve the capabilities and performance of computational storage devices.

EXTFUSE [3] addresses the significant performance penalties associated with user file systems primarily caused by frameworks like FUSE, which rely on a minimal interposition layer in the kernel to forward all low-level requests to user space. This design results in frequent kernel-user context switching and data copying, leading to degraded performance, particularly in high I/O throughput or low latency scenarios such as databases. The motivation behind this framework is to overcome the performance limitations of existing solutions, which are too general-purpose and do not cater to the specific needs of file systems. The goal is to introduce EXTFUSE, a framework that enables the development of extensible user file systems by allowing applications to register specialized request handlers directly in the kernel. This reduces the need for frequent context switching and data copying by allowing fast path (kernel-side) and slow path (user-side) logic to coexist, thus improving performance while maintaining the flexibility

and advantages of user file systems without requiring significant changes to their design.

Zhe Yang et al.[17] is a system focused on optimizing data processing by addressing the significant overhead of transferring large volumes of data from storage devices to host systems for processing in data-intensive applications. The motivation stems from the inefficiencies of traditional approaches, where excessive data movement leads to performance bottlenecks. Computational storage devices offer a solution by allowing computation to occur directly on the storage device, minimizing the need for data transfer. The goal of [17] is to introduce a unified I/O stack that efficiently manages both computational and storage resources across hosts and devices, thereby significantly improving the performance of data processing tasks by reducing the overhead of data movement and optimizing resource utilization.

bpfbox [9] is focused on improving security through process confinement. The motivation behind bpfbox arises from the complexity and inflexibility of existing Linux process confinement mechanisms, which rely on a mix of security primitives that are not specifically designed for this purpose. The goal of bpfbox is to demonstrate that eBPF can provide a simpler, more efficient, and flexible solution for process confinement. Process confinement is critical for enforcing the principle of least privilege, isolating applications, and preventing unauthorized access or interference between processes, particularly in multi-tenant or cloud environments. As a proof-of-concept application, bpfbox utilizes eBPF to implement confinement with a minimal codebase, offering multiple levels of confinement through a straightforward policy language. By restricting the actions and capabilities of processes, bpfbox aims to enhance the security of the system by containing potential damage from compromised or malicious processes.

Durian [18] focuses on monitoring and analyzing the fairness of the Linux Completely Fair Scheduler (CFS), under the assumption that the scheduler may not always be fair. The goal is to introduce Durian, an eBPF-based system designed to track task states within CFS and analyze their fairness. Durian decouples the data collection process from the analysis process, allowing for flexible and generic analysis methods. It collects and exports scheduling statistics to an external store, enabling independent analysis by multiple processes without impacting the scheduler's performance or safety. By comparing the theoretical and actual fair share of CPU time allocated to tasks, Durian helps identify discrepancies and provides insights that can be used to fine-tune system behavior, either to minimize or maximize the time quantum a task receives, depending on the desired outcome.

Poster [15] aims to automatically generate and apply patches to real-time embedded devices without requiring a reboot or system halt. This approach enhances the security of critical devices used in sectors like healthcare, automotive, and industrial control by enabling swift and efficient

vulnerability fixes. The poster presents itself currently as only a theoretical concept that could be implemented using eBPF.

NCScope [22] is a tool designed to enhance the analysis of native code within Android applications. By leveraging hardware assistance and eBPF, NCScope collects execution traces and memory data with minimal overhead, surpassing the limitations of existing dynamic analysis tools. The primary goal is to make native code analysis more effective in detecting complex behaviors, such as self-protection mechanisms, anti-analysis techniques, memory corruption, and performance anomalies. NCScope contributes to improved security and efficiency in app development and malware detection, offering a more robust solution for analyzing native code in Android environments.

Augmenter [19] focuses on enhancing network tracing in data center environments, which are evolving with higher bandwidth-delay products and a shift towards nano-services architecture characterized by many small flows. The traditional TCP/IP stack faces challenges in this context, such as accurately estimating network state due to the short duration of many flows and a lack of flexibility in adjusting parameters like the initial congestion window (IW) for TCP connections. To address these issues, the work introduces Augmenter a framework that enhances the TCP/IP stack's visibility and flexibility using eBPF. Augmenter collects real-time state information from ongoing flows and uses it to dynamically adjust the initial congestion window and optimize the management of current and future flows. This approach aims to improve application performance by making the networking stack more responsive and adaptable, addressing the limitations of the traditional TCP/IP stack in modern data center networks while enabling more intelligent flow management and optimization.

2.5.2 Tracing System Property Analysis

This analysis will focus on the tracing system properties of seven previously presented systems: PHOEBE [20], CAT [7], DIO [8], Zpoline [11], A protocol-independent container network observability analysis system based on eBPF [13], Enhancing Packet Tracing of Microservices in Container Overlay Networks using eBPF [12] and Inspektor Gadget [5].

Information Visualization

One of the key properties of distributed system tracing is representing information to the user.

As tracing systems capture a wealth of information about system behavior, application interactions, and performance metrics, visual representations are indispensable for discerning trends, identifying bottlenecks, and making informed decisions. Neglecting investment in system visual-

ization entails several potential downsides: not understanding complex patterns, dependencies, and anomalies within traced data becomes arduous making it harder to identify performance bottlenecks, recognize trends, and make informed decisions.

From this set of relevant systems, PHOEBE [20] was the first system to engage in a visual representation of the traced data, since its main focus is to identify which system calls break the system, it opts for a tabular format to present the information captured (every system call within the traced period), leaving room for improvement. CAT [7] provides a new feature in the form of a space-time diagram that offers a visualization strategy that captures temporal dynamics and establishes relationships between system calls. This approach enriches the traced data with a visual narrative.

Next came DIO [8], with the use of Kibana's dashboards, which also introduced custom dashboards and filters for a type of event, which matches its goal to allow custom visualization of data. As Zpoline 's [11] objective is to introduce custom behavior into system calls, it does not opt for any information visualization.

In contrast to application-level tracing systems that often leverage intricate visualization strategies, container observability systems opt for simplicity in their approach. Since the output information here is more simplistic, it can be helpful to facilitate the visualization process.

PICNO [13] was the first system to present data visually through the utilization of graphs, metrics such as pod connection and pod connection latency can be graphically visualized. Next EPTM [12] opted for the most basic data visualization: presenting values in the console, this is due to its innovation not being on the visualization part.

Lastly, Inspektor Gadget [5] offers a unique feature, the ability to filter information directly within the command line. This capability allows users to focus on specific metrics or events of interest. While committing to this visualization approach, Inspektor Gadget's capacity for dynamic filtering adds a layer of customization.

Coupled with this feature, the ability to relate system calls to each other is pivotal to understanding application behavior. Understanding the relationships between system calls enhances observability. CAT [7] stands out as the only system to relate events by using a color scheme in the space-time diagram, where events marked with the same color are related or similar.

Enrich Traced Data

For these systems to have meaningful data to visualize, another fundamental piece of them is to enhance traced data with additional information. Enriching traced data with details such

as process identifiers, timestamps, and relevant metadata facilitates further analysis and troubleshooting. Failure to do this can lead to notable downsides: deciphering the meaning and relevance of raw data can be challenging, leading to missed opportunities for insightful analysis. In essence, neglecting the enhancement of traced data limits the depth of analysis, obstructs effective troubleshooting, and diminishes the overall value derived from the tracing process within complex computing environments.

PHOEBE [20] set the standard by enhancing traced data with additional context, CAT [7] stands out for its content-aware system, a feature that captures the context and the content of traced events. This mechanism involves sending context and content through a ring buffer and kernel Maps, respectively. However, the approach introduces a noteworthy consideration, it may lead to the truncation of events. This truncation is a trade-off that facilitates the transmission of data but comes with the potential drawback of a miss relation between events. The risk of events being truncated and consequently, correlating unrelated events, stands as a challenge within CAT's content-aware system, thus leaving room for further improvement.

DIO [8] matched the state of the art by also enhancing traced data with additional context. These tracing systems incorporate extra details such as process identifiers, timestamps, and comprehensive metadata.

On the container tracing systems side, all three, PICNO [13], EPTM [12], and Inspektor Gadget [5], meet the standards for the state of the art by enhancing traced data with additional context.

Expressiveness

The third property every tracing system aims to achieve is full expressiveness.

Full expressiveness means tracing every possible scenario, for example, wanting to trace every system call `x` from pid `y` to pid `z` and being able to do so. This entails a goal that every tracing system should aim for, the ability to capture all the system calls scenarios within a given period. When a system lacks full expressiveness, it may lose important system call scenarios during the tracing process, which can result in an incomplete and potentially misleading representation of application behavior. Overall, the downsides of limited expressiveness underscore why tracing systems aim to provide a detailed and exhaustive account of system calls.

Since Phoebe's [20] goal is to detect which system calls can corrupt the system state, it can capture the full range of information about system calls, thus achieving full expressiveness.

CAT [7] has the downside of potentially losing information, either by truncating events within

its content-aware system or by losing some event information due to constraints within the ring buffer, so it does not achieve full expressiveness. Since DIO [8] does not include content in its tracing enhancing phase, it improves upon what CAT built by not having truncated events, but it also has the downside of potentially losing events due to constraints within the ring buffer, these constraints are due to its limited space, and the user space not being able to keep up with the amount of events gathered.

Finally, since Zpoline [11] redirects every system call to a user-defined function, it can capture the full range of information about system calls.

On the container tracing side, PICNO [13], EPTM [12], and Inspektor Gadget [5] fail to capture every network request that is made in the containerized environment, EPTM [12] reduces it search space only to requests with the HTTP header, for PICNO [13] and Inspektor Gadget [5], it is not specified which requests are missed, only that full expressiveness is not achieved.

Behaviour introduced Verified

The struggle with the eBPF verifier is something that most systems face in terms of what behavior can be added to systems calls, so Zpoline's [11] capacity to introduce non-verified behavior into kernel functions seems to stand as a groundbreaking feature. Zpoline's [11] achieves this by redirecting system calls into a function defined in user space, so the behavior of the function is not verified by the eBPF verifier, but it is crucial to acknowledge the practical limitations associated with this flexibility. The potential to customize behavior confronts the challenge that, in practice, this capability is more constrained than initially envisioned. The practical implications of Zpoline's flexibility become apparent when evaluating the extent of modifications that can be effectively applied, comparing this with CAT [7], DIO [8] and PHOEBE [20], the complexity introduced by Zpoline [11] is not valuable for the gains observed.

Latency Measurement

Measuring latency on the container network is fundamental, latency, representing the time taken for data to travel between two points in the network, serves as a critical metric for assessing the efficiency of containerized applications.

First and foremost, latency directly impacts the responsiveness of applications, and elevated latency levels mean network congestion, inefficient communication, or other performance bottlenecks. These systems can pinpoint specific areas of the network that may be contributing to delays. Moreover, observing latency in real-time enables rapid detection and resolution of issues that could otherwise impact the user experience and the performance of containerized

applications.

Measuring latency on the container network within observability systems is a crucial feature in optimizing performance and maintaining a responsive and efficient containerized infrastructure. PICNO [13], EPTM [12], and Inspektor Gadget [5] take a similar approach to this latency measurement. All three operating within Kubernetes (k8s) clusters on worker nodes, share similarities in their approach to information collection with their utilization of the helper function bpf-ktime-get-ns()[17] to acquire precise timestamps, latency can then be deduced. This function becomes a linchpin in the systems' ability to perform latency analysis, offering a granular understanding of the time taken for various operations within containerized environments.

Extra Features

Finally, while Inspektor Gadget [5] shares the commonality of measuring latency within Kubernetes clusters, it stands out for all its other functionalities. Inspektor Gadget provides an array of built-in gadgets (eBPF programs that can be deployed) that extend its capabilities. These functionalities range from observing and recommending system configurations to auditing specific subsystems within the cluster. Moreover, Inspektor Gadget can dynamically report the current captured scenario during the tracing activities. This real-time reporting feature provides instant insights into ongoing system behavior.

2.6 Discussion

Across the state-of-the-art systems studied, there is a clear emphasis on improving observability, troubleshooting efficiency, and overall system performance. These systems, though targeting different areas, such as storage, networking, security, scheduling, and hardware, utilize eBPF in various ways to enhance performance. Common strategies include offloading computation to the kernel to reduce overhead, enabling detailed performance monitoring directly in the kernel, and allowing real-time analysis and optimization. This unified approach allows systems to maximize resource efficiency, improve responsiveness, and provide deeper insights into system behavior. In conclusion, all the systems analyzed contribute to advancing eBPF technology, playing a pivotal role in optimizing performance and facilitating efficient troubleshooting. Address issues or limitations in several Linux components.

Despite the widespread use of eBPF frameworks in these subsystems, there is often no clear explanation of why a particular framework is chosen or what its specific advantages are. Our study focuses on six eBPF frameworks, BCC, libbpf, Cilium, Aya, bpftrace, and Eunomia,

[17] `https://man7.org/linux/man-pages/man7/bpf-helpers.7.html`

providing clarity on their best use cases. We compare these frameworks across several criteria, including ease of use, performance, dependencies, and deployment requirements, highlighting their strengths and limitations in different contexts. A deeper exploration of these aspects is presented in §3 and §4, where we offer both theoretical and practical analysis.

Chapter 3

Theorical analysis

The decentralized nature of distributed systems makes traditional monitoring tools inadequate and new ones are not yet capable of solving this issue to its fullest, hindering our ability to understand and optimize system performance.

Furthermore, eBPF frameworks are crucial for integrating and executing custom eBPF programs within the Linux kernel, offering abstractions and high-level APIs that simplify the development of eBPF-based applications. While they make eBPF more accessible, their complexity and incomplete documentation can pose challenges for developers. These frameworks balance ease of use with functionality, aiming to create a more accessible and powerful platform for network and system-level innovations.

Our work will focus on analyzing the eBPF frameworks across several criteria, divided into two parts. The first is the theoretical part of the ebpf framework analysis done in this chapter, this will provide better insights into how the frameworks differ from each other. The second one is the practical analysis described in §4. So in the next two chapters, we aim to answer the following questions:

- Which eBPF framework is most used by the state-of-the-art and for what purpose?
- Which eBPF frameworks are used in the industry and for what purpose?
- Which eBPF framework is faster according to our implementation?

Besides these questions we also aim to provide clearer insights into every ebpf framework studied regarding the following criteria:

- Architecture and implementation details.
- Use cases.

- Deployment requirements.
- Code complexity.

3.1 Theorical Methodology

Our theoretical analysis will focus on the six ebpf frameworks studied, focusing on various aspects to provide a better understanding of their capabilities and trade-offs. This analysis will cover key dimensions, including the design and architecture of each framework, focusing on how the framework is structured and how it interacts with the system kernel and user space.

We will also examine the range of eBPF use cases each framework supports (e.g., tracing, networking, security). Furthermore, the analysis will explore the implementation details, such as the programming languages used (e.g., C, Python, or Go) and how these choices impact overall accessibility and runtime efficiency. Another aspect of the analysis is the examination of each framework's dependencies and deployment requirements, such as the need for specific kernel versions, libraries, or toolchains. These factors can significantly affect the ease of integration and compatibility with various system environments. Finally, we will evaluate the code complexity of each framework, considering how difficult or straightforward it is for developers to implement and maintain eBPF programs within each framework.

3.1.1 Framework Architecture and Implementation Details

BCC is built as a high-level abstraction over the complexities of eBPF programming. It simplifies the creation of eBPF programs by allowing to write them in Python[1]. BCC handles the compilation and loading of eBPF programs into the kernel, making it easier to work with. This abstraction reduces complexity for users but adds overhead with its reliance on external dependencies like LLVM.

libbpf offers a more direct, low-level C library for interacting with eBPF, providing fine-grained control over critical tasks such as the compilation of eBPF bytecode, loading programs into the kernel, and attaching them to specific kernel hooks. Its architecture is designed to be lightweight and closely integrated with the Linux kernel, providing minimal overhead. This closer interaction with the kernel allows for highly optimized eBPF usage, but it comes at the cost of increased complexity, as libbpf does not provide the high-level abstractions found in other eBPF frameworks.

Eunomia-bpf prioritizes simplicity and ease of use in eBPF development and deployment.

[1] `https://www.python.org/`

By allowing eBPF programs to be packaged as WebAssembly (WASM) modules[2], Eunomia-bpf streamlines the process of running pre-compiled eBPF programs, making it easier to deploy these programs across different environments without needing to recompile or handle complex kernel dependencies. However, it also limits the level of customization available. Since programs are pre-compiled and packaged in WASM or JSON formats, it provides less flexibility to modify or fine-tune eBPF code at runtime.

bpftrace is designed with a scripting-based architecture that simplifies eBPF programming, making it accessible to users with minimal kernel or eBPF expertise. It uses a high-level language inspired by awk[3] and C, allowing users to write concise scripts for system diagnostics and tracing tasks. bpftrace compiles these scripts into eBPF bytecode using LLVM[4], which is then injected into the Linux kernel via libbpf and BCC, enabling real-time tracing through kernel-level and user-level hooks like kprobes, uprobes, and tracepoints. bpftrace simplifies tracing tasks, but it is mostly limited to system diagnostics.

Aya, built entirely in Rust[5], bypasses traditional C-based libraries like libbpf and instead leverages Rust's toolchains for direct access to eBPF. By using Rust's ecosystem, Aya integrates seamlessly with tools like Cargo[6] for managing dependencies, building, and testing, streamlining the development workflow for eBPF applications. Aya's support for Compile Once, Run Everywhere (CO-RE) also enhances portability, allowing eBPF programs to run across different Linux kernel versions without modification.

Cilium's architecture is optimized for high-performance networking and security in cloud-native environments, with a particular focus on Kubernetes[7]. It integrates deeply with Kubernetes to provide seamless service discovery, network observability, and the enforcement of security policies, making it highly suitable for large-scale microservices deployments. This architecture allows Cilium to handle the dynamic, distributed nature of containerized workloads, ensuring secure and efficient communication between services. Cilium's architecture is built using Go[8], which it uses to manage the loading of eBPF programs into the Linux kernel.

3.1.2 Use Cases

BCC is ideal for system monitoring, performance tuning, and security auditing. Its high-level abstractions and its set of pre-built tools make it accessible to rapidly create and deploy eBPF

[2] https://webassembly.github.io/spec/core/syntax/modules.html
[3] https://www.awk.dev/
[4] https://llvm.org/
[5] https://www.rust-lang.org/
[6] https://doc.rust-lang.org/cargo/
[7] https://kubernetes.io/
[8] https://go.dev/

programs for tracing and diagnostics, without requiring deep kernel expertise. BCC is especially suited for real-time observability and troubleshooting, offering tools for gaining insights into system performance and behavior with minimal development effort.

libbpf is best suited for production environments in which performance, efficiency, and low overhead are critical. It provides low-level access to the eBPF subsystem, making it ideal if control to build custom eBPF solutions tailored for networking, security, or system monitoring is required. Its lightweight nature and close integration with the kernel allow for highly optimized eBPF programs, ensuring minimal resource consumption and maximum performance.

Eunomia-bpf excels in lightweight, cross-platform environments such as cloud and containerized infrastructures. Its zero-dependency approach using pre-compiled eBPF programs packaged as WebAssembly (WASM) modules makes it perfect for scenarios that prioritize ease of deployment, portability, and minimal resource consumption.

bpftrace is designed for real-time, interactive system diagnostics. Its simple, high-level language allows one to write tracing scripts without the complexity of writing full eBPF programs. bpftrace is ideal for quick, ad-hoc tracing and performance analysis, making it an excellent tool to gather immediate insights into system behavior.

Aya is the go-to framework for Rust-based applications, providing a safe, efficient, and native eBPF integration with Rust. Aya is ideal for high-performance tracing and monitoring, making it perfect for building robust and performant eBPF applications in Rust.

Cilium is optimized for cloud-native networking, especially in Kubernetes environments. It uses eBPF to provide advanced networking features such as load balancing, network policy enforcement, and deep observability. Cilium is essential for managing large-scale, distributed microservices architectures, offering transparent service discovery and security for modern, dynamic environments. Its deep integration with Kubernetes makes it a natural choice for organizations prioritizing scalability, security, and performance in cloud-native deployments.

3.1.3 Deployment Requirements

BCC relies on external tools like LLVM/Clang[9] for compiling eBPF programs and uses Python libraries for writing and interacting with eBPF code. This increases its overall dependency footprint, requiring both the LLVM/Clang toolchain and Python to be installed. Additionally, it demands a more recent Linux kernel (version 4.1+ for most features), making it more suited for environments where these dependencies can be easily managed.

libbpf has minimal dependencies, only requiring a Linux kernel with version 4.9+ for full fea-

[9] `https://clang.llvm.org/`

ture support. As a low-level C library, libbpf does not need external toolchains like LLVM/Clang.

Eunomia-bpf follows a "zero-dependency" approach, eliminating the need for compilation tools like LLVM/Clang by using pre-compiled eBPF programs packaged as WASM modules. This makes it a lightweight solution with very few external dependencies.

bpftrace requires the LLVM/Clang toolchain to compile its high-level tracing programs into eBPF bytecode and demands a Linux kernel of version 4.9+ or higher. While it simplifies tracing tasks, its dependency on external compilation tools increases setup complexity.

Aya requires the Rust toolchain and a Linux kernel version 5.4+ for full feature support. Its integration with Rust makes it well-suited for Rust-based environments, but it requires familiarity with the Rust ecosystem, including tools like Cargo for managing dependencies and builds.

Cilium relies on libbpf as its underlying layer for interacting with eBPF. It also depends on Kubernetes for full functionality, given its deep integration into cloud-native environments. Cilium requires a modern Linux kernel of version 4.9+, and while its setup might be more complex due to its dependencies, its scalability and deep integration with containerized systems make it perfect for large-scale, distributed environments.

3.1.4 Code Complexity

BCC simplifies the complexity involved in eBPF programming by abstracting away low-level details. However, the framework itself is quite large, as it includes dependencies such as LLVM and Python bindings. Writing eBPF programs using BCC is relatively straightforward, requiring only knowledge of Python, which lowers the entry barrier for developers.

libbpf, on the other hand, offers a more direct, low-level interface to eBPF, providing developers with greater control over program execution and performance. However, this comes at the cost of increased code complexity. To effectively use libbpf, developers must be proficient in C and have a solid understanding of kernel internals.

Eunomia-bpf takes a different approach by minimizing code complexity through pre-compiled WASM modules for eBPF programs. This significantly simplifies both the writing and deployment processes. Still, it limits the flexibility and customization options compared to frameworks that allow direct interaction with eBPF code, making it less suitable for complex, tailored use cases.

bpftrace greatly reduces the complexity of eBPF programming by offering a high-level scripting interface. It is particularly well-suited for quick, interactive tracing tasks and requires very little programming knowledge, as simple commands can handle many tasks. However, bpftrace's

flexibility is limited, as it is primarily focused on quick simple tracing, rather than more complex or varied eBPF applications.

Aya leverages Rust's memory safety and modern programming paradigms to simplify the development process, reducing common errors like memory leaks or unsafe pointer access. While Aya offers more control than higher-level frameworks, it requires deep knowledge of both Rust and C for effective use, which increases the learning curve.

Cilium abstracts much of the complexity involved in networking and security tasks, providing a high-level interface for managing network policies and observability in Kubernetes environments. However, its deep integration with Kubernetes adds to the overall complexity, and developers need a good understanding of Go for deploying and extending Cilium, which can be a barrier for those unfamiliar with the Go ecosystem.

3.2 State Of The Art Usage

In this section, we will analyze which and what eBPF frameworks are used by the state-of-the-art works that were discussed in §2.5.1.

3.2.1 System Application

In terms of system application, tracing is the most frequent use of eBPF frameworks, appearing in 11 instances, mainly EPTM[12], CAT[7], DIO[8], Container Network Observability [13], inspektor Gadget [5], Nahida [16] and PHOEBE[20]. Tracing allows developers to monitor system events and behavior in real time, making it highly valuable for performance tuning, debugging, and observing kernel or user-space activities. Storage systems show 4 instances where eBPF is used IO[17], EFFSUS[3], Delilah[10] and XRP[21]. In these cases, eBPF is applied to monitor I/O performance, track storage-related events, and optimize file system operations. Offloading tasks to the kernel is observed in 2 instances, primarily in the networking domain. Here, eBPF is used to offload packet processing and filtering tasks directly to the kernel, significantly improving performance and reducing latency for high-throughput networking environments.

Security applications for eBPF are documented in 2 cases, Poster[15] and bpfBox[9]. In these cases, eBPF is used to enforce security policies, monitor suspicious activities, or control traffic between services. In scheduling, there is one instance where eBPF is applied, in Durian[18]. eBPF is used here to optimize task scheduling by monitoring and adjusting resource allocation at the kernel level. This allows for more efficient process management and system performance improvements in environments where workload balancing is crucial. Lastly, the usage of eBPF to improve hardware is mentioned in 1 case, eHDL[14]. In this instance, eBPF is employed to

interact with hardware components. It demonstrates the versatility of eBPF but also highlights that its primary strength lies in software system monitoring and optimization.

3.2.2 eBPF Framework Usage

In terms of eBPF framework usage, libbpf is the most widely used eBPF framework, appearing in the implementation of [16], [21], [24], [3], [17] and [19]. It is favored for tasks that require direct access to the eBPF subsystem with minimal overhead, such as high-performance storage monitoring, networking, and low-level system tracing, with this being the reason authors say they chose libbpf over other frameworks. BCC is used in four cases, primarily for tracing and monitoring tasks, the systems that use BCC are NCScope[22], bpfBox[9], DIO[8], and PHOEBE[20].

bpftrace is not used in the state of the art, as expected. While bpftrace is known for simplifying eBPF program writing with a high-level scripting interface, making it suitable for quick, interactive tracing of kernel and user-space events, it is not widely used in distributed systems. This is largely due to bpftrace not scaling well for distributed environments and also does not cover specific cases that are needed because its high-level abstractions and dynamic scripting model introduce more computational overhead, making it less efficient for the continuous, large-scale data processing required in distributed systems. Its ad-hoc, real-time tracing approach is more suitable for local diagnostics and system monitoring, where ease of use and fast deployment are priorities, but it becomes less effective in handling the complexity and scale of distributed systems.

Cilium appears in zero use cases in the state-of-the-art, but is it commonly utilized in the industry, as we will discuss in §3.3.

Eunomia and Aya are less frequently mentioned in the state-of-the-art, which suggests that these frameworks are still emerging. Both Eunomia and Aya represent newer, specialized approaches to eBPF development, and while they offer specific advantages, their current limited mention in the state-of-the-art reflects their ongoing development and emerging adoption.

While not an ebpf framework, XDP[10] is mentioned in 2 cases for offloading networking tasks to the kernel. XDP is designed for high-performance packet processing, allowing networking operations to be handled with minimal latency directly in the kernel, the systems that use XDP are eHDL[14] and Electrode[23].

Unfortunately, in the state-of-the-art analysis, there are 9 instances where the specific eBPF framework used is unknown or is based on a custom implementation without the usage of any

[10] `https://en.wikipedia.org/wiki/Express_Data_Path`

framework.

3.3 Industry Usage

eBPF is also used in the industry, where many major tech companies are utilizing ebpf for a wide range of applications across networking, security, and observability, some of the main companies are Meta[11] uses ebpf extensively in its data centers for load balancing and packet processing, which significantly improves throughput and scheduler efficiency, Shopify[12] and Apple integrate ebpf via Falco[13], which is built on libbpf, for kernel security monitoring, Google[14] employs ebpf for runtime security and observability, processing most data center traffic through ebpf. Netflix[15] relies on ebpf for performance monitoring and analysis, enhancing troubleshooting and visibility across its production servers, Android[16] leverages ebpf for network usage, power, and memory profiling, demonstrating its utility in mobile OS performance monitoring, Cloudflare[17] uses ebpf (specifically XDP) for network security and observability, helping manage and secure its traffic, Microsoft[18] improves observability in Kubernetes environments with ebpf, using tools like Inspektor Gadget[5] for process inspection. Sky[19] and Capital One[20] both utilize Cilium for cloud networking, with Sky highlighting its ability to replace iptables and enhance DNS and HTTP connectivity while conducting functional and non-functional testing through Cilium's provided test suite, Ikea[21] also leverages ebpf through Cilium for networking and load balancing in their private cloud. By using Cilium, Ikea optimizes its cloud infrastructure, enhancing network performance and ensuring efficient load balancing across its services and LinkedIn[22] utilizes ebpf for observability, allowing the company to collect valuable data points with minimal overhead. This use of eBPF provides LinkedIn with deep insights into system performance while maintaining efficiency.

Besides these companies, many others make use of ebpf to optimize their systems, they can be found in the ebpf official website[23].

[11] https://business.facebook.com/latest/home?nav_ref=bm_home_redirect&asset_id=270061450382723
[12] https://www.shopify.com/uk/plus/solutions/retail-and-point-of-sale
[13] https://falco.org/tags/ebpf/
[14] https://www.google.pt/?hl=pt-PT
[15] https://www.netflix.com/pt
[16] https://www.android.com/intl/pt_pt/
[17] https://www.cloudflare.com/
[18] https://www.microsoft.com/pt-pt
[19] https://www.sky.com/
[20] https://www.capitalone.com/
[21] https://www.ikea.com/pt/pt/
[22] https://www.linkedin.com/home?originalSubdomain=pt
[23] https://ebpf.io/case-studies/

3.4 Framework Analysis

When analyzing the eBPF frameworks, it becomes clear that each framework is designed with specific use cases, trade-offs, and strengths. The key factors that influence the choice of framework include ease of use, performance, system requirements, code complexity, and the target environment. The following analysis will base itself on the properties studied, stating what the theoretical framework choice should be if we want to maximize that property.

Regarding ease of use and abstraction BCC is by far the most user-friendly framework for eBPF programming, especially for those who prefer high-level programming languages like Python. The framework abstracts away most of the complexities involved in interacting with the kernel, providing a clear API. bpftrace also prioritizes ease of use, but it focuses on fast, interactive tracing. Its high-level scripting approach makes it the easiest to use. However, its scope is more limited than BCC, as it does not offer much flexibility outside of quick and simple tracing. These frameworks, especially BCC, should be the best for those who want to quickly implement tracing, monitoring, or diagnostics without deep kernel knowledge, but they lack in terms of resource efficiency and flexibility across several use cases.

Regarding performance and low-level control over eBPF, libbpf provides the most granular control over eBPF, allowing developers to interact directly with kernel hooks and optimize performance. Its minimal dependencies and closer integration with the kernel make it a lightweight, high-performance option. However, this framework requires in-depth knowledge of C and kernel internals, it should be the choice if the target is a production environment requiring high performance, fine-tuned control, and minimal overhead. Aya, a Rust-based framework, is another excellent option for low-level control with added benefits like memory safety, thanks to Rust's programming paradigms. Aya bypasses C-based libraries like libbpf, however, Aya requires familiarity with the Rust ecosystem, which is a steep learning curve for developers not familiar with the language, and it should be the go-to for developers looking for low-level control with the safety guarantees of Rust, particularly in performance-critical environments. Both these systems require deep knowledge of the kernel works and programming languages, C and Rust, respectively.

Eunomia-bpf is the best for simplicity as it offers a minimalist approach by using pre-compiled eBPF programs. This simplifies deployment and reduces the complexity of working with eBPF. However, it trades flexibility for simplicity, and optimizing the eBPF programs is limited compared to more full-featured frameworks like libbpf or BCC. Its main use case is for cross-platform environments like cloud and containerized applications, where simplicity, ease of deployment, and portability are critical.

Cilium is highly specialized in framework integration and specialization for cloud-native environments, especially Kubernetes. It uses eBPF to provide advanced networking and security features such as load balancing, network policy enforcement, and observability. By integrating deeply with Kubernetes, Cilium offers networking solutions at scale. However, this specialization comes with added complexity, as users must be familiar with both Go (for extending Cilium) and Kubernetes for optimal usage. It is best for large-scale, distributed environments that focus on microservices, especially in Kubernetes clusters, it isn't ideal for non-cloud-native environments or smaller-scale applications where Kubernetes overhead might be unnecessary.

Finally, on code Complexity and Developer Expertise BCC significantly reduces code complexity by providing Python bindings and abstracting low-level details, making it an excellent choice for developers uncomfortable with kernel internals or C programming. libbpf requires advanced knowledge of kernel internals and C programming, resulting in more complex code but also offering full control over the eBPF interaction layer. This makes it ideal for high-performance applications, but the complexity may be a hurdle for teams without specialized expertise. Aya offers a middle ground, with the control and efficiency of low-level frameworks like libbpf. However, the learning curve of Rust will be a barrier for those unfamiliar with the language.

3.4.1 Usage Analysis

Crosschecking information between the system application and framework used, we can see that libbpf is the most widely used framework in the state-of-the-art, especially for tracing and high-performance tasks. For example, Nahida [16] leverages libbpf for tracing, highlighting its capability to offer low-overhead system monitoring in real time. Similarly, XRP[21] and IO[17] use libbpf in storage systems to optimize I/O performance and monitor storage-related events. The choice of libbpf in these cases demonstrates that it is particularly well-suited for scenarios where fine control over the kernel is required without sacrificing performance. Its frequent use in DINT[24] and EFFSUS[3] further reinforces libbpf's reputation as an excellent tool for low-level system tracing and monitoring in performance-critical environments. From these examples, we can conclude that libbpf is the preferred framework for tracing tasks where performance and kernel efficiency are paramount.

In the state-of-the-art, BCC is used in several systems, including NCScope[22], bpfBox[9], and PHOEBE[20], which further solidifies its role as a reliable framework for tracing and monitoring tasks, especially when ease of use is a priority.

In the industry, eBPF has proven to be a versatile and powerful tool across various sectors of

the tech industry, with widespread adoption for tasks such as observability, security, networking, and performance monitoring. Companies like Meta, Google, Netflix, and Cloudflare use eBPF to enhance throughput, improve visibility, and optimize system performance, while organizations like Apple and Microsoft rely on it for kernel-level security monitoring and process inspection in cloud environments.

Although Cilium was not prominent in the state-of-the-art research, its strong presence in the industry, used by companies such as Ikea, Sky, and Capital One, demonstrates its effectiveness in cloud networking and load balancings. This indicates that while academic research may focus on other frameworks, Cilium has become a go-to solution for networking and security in large-scale, cloud-native environments within the industry.

3.5 Conclusion

In the context of eBPF framework selection, libbpf stands out as the preferred choice for high-performance and low-overhead tracing tasks, particularly in performance-critical systems and state-of-the-art research. Its widespread adoption in storage systems and real-time monitoring applications demonstrates its capability to provide fine-grained control over kernel operations. BCC, while less efficient than libbpf theoretically, is often used in environments where ease of use and quick deployment are prioritized, especially in academic systems for tracing and monitoring. In industry settings, eBPF has seen substantial adoption across sectors for observability, security, and performance tasks, with frameworks like Cilium playing a pivotal role in cloud-native networking and load balancing, especially within Kubernetes environments.

Ultimately, the choice of framework depends on the specific requirements of the task at hand, whether it be ease of development, performance optimization, or deep kernel-level control.

To end this section, Table 3.1 summarizes the conclusions that we bring out of it, the first column represents the property or use case desired, the second column, the best framework to achieve the desired property, and the last one, an explanation of why it is the best suited.

Property	Best Framework	Explanation
Ease of Use & Abstraction	BCC, bpftrace	BCC simplifies eBPF programming with high-level Python bindings; bpftrace is excellent for quick, interactive tracing, though limited in scope.
Performance & Low-level Control	libbpf, Aya, Cillium	libbpf offers fine control and minimal overhead, ideal for production environments; Aya and Cillium provide similar performances.
Simplicity	Eunomia-bpf, bpfTrace	Eunomia offers pre-compiled eBPF programs, simplifying deployment, though it lacks flexibility for custom optimizations. bpftrace prioritizes ease of use, allowing users to quickly create and deploy eBPF programs with just one command.
Cloud-native Specialization	Cilium	Deeply integrates with Kubernetes for advanced networking and security, ideal for large-scale, cloud-native environments.
Code Complexity & Developer Expertise	libbpf, Aya and Cillium	libbpf requires deep C knowledge; Aya requires Rust deep knowledge and Cillium requires Go knowledge

Table 3.1: **Summary of the framework analysis**

Chapter 4

Experimental evaluation

In this chapter, we will focus on the practical analysis of the six eBPF frameworks studied, with the goal of providing better insights into the burden a framework brings into the target system, mainly focusing on overhead introduced into the system.

To achieve this goal, we developed five eBPF programs, implemented across all the selected frameworks, ensuring that each implementation was as consistent and comparable as possible. These programs were designed to cover four key Linux subsystems: File, Network, Process Management, and Memory Management.

4.1 Methodology

We will gather three primary quantitative metrics: throughput expressed in operations per second (Ops/s), which reflects the framework's ability to handle large amounts of traffic without degrading performance. For instance, eBPF frameworks designed for packet filtering must maintain high throughput to keep up with real-time traffic without introducing bottlenecks. When throughput is high, the system can efficiently handle tasks like load balancing, tracing, or intrusion detection, which are critical in environments where every millisecond of delay could translate into performance losses or security vulnerabilities. The second quantitative metric is the time of each experiment, measured in milliseconds and finally, we will measure latency, expressed in microseconds, this metric allows us to assess how efficiently each eBPF framework handles micro-level timing, which is crucial in performance-sensitive scenarios like tracing or network traffic filtering, the units for all metrics are the ones used in the YCSB output, thus the difference. We will express all metrics' mean across every experiment, along with the standard deviation. The mean provides a central tendency of the data, allowing us to understand the average performance of each eBPF framework. By averaging the results, we can reduce

the influence of outliers. The standard deviation is equally important, as it measures the variability or dispersion of the data. A low standard deviation indicates that the metric values are clustered closely around the mean, suggesting consistent performance across different runs. Contrarily, a high standard deviation indicates more variability, signaling that the framework's performance might fluctuate under different conditions or workloads. Together, the mean and standard deviation provide a comprehensive view of the typical performance and the stability of each framework.

In addition to these three metrics, we will capture resource usage metrics to provide a more comprehensive analysis of system performance, which are:

- User CPU usage (%) reflects how much CPU time is spent on user-space tasks across the frameworks.
- System CPU usage (%), measures the percentage of CPU time spent on kernel-level tasks.
- Waiting for I/O CPU (%), reflects how much CPU time is spent waiting for I/O operations to complete, which is crucial in write-intensive scenarios.
- Memory used (KB), which captures the total memory consumption during the experiments, this is critical in evaluating how each framework impacts system resources, especially in memory-intensive tasks like writing.

4.1.1 Distribution Of eBPF Programs

The eBPF programs are distributed across the Linux components as follows:

Two of the five eBPF programs developed focus on the store subsystem. The first program monitors the number of files the system opens, capturing key information such as the process ID (PID) responsible for opening the files. This data is then relayed to user space using eBPF maps for further analysis. The second program traces every read-and-write operation performed by the system, recording details such as the process ID, thread ID (TID), and the name of the program (comm) responsible for the operation. This information is efficiently transferred to user space using a ring buffer.

The network eBPF program is attached to the traffic control (TC) hook to trace every TCP request in the network subsystem. Since filtering by process ID was not possible at that stage, the program filters network activity based on the network interface instead. It stores the last 1,024 requests, capturing essential details such as the source IP, destination IP, and a portion of the packet payload. This data is shared with user space through an eBPF map.

For the process management subsystem, the eBPF program tracks every process spawned by a target process ID, capturing both the newly created process ID and the parent process ID. Additional relevant metadata is also collected, with all the information sent to user space via eBPF maps for tracking.

Lastly, in the memory subsystem, the eBPF program monitors memory access patterns by counting cache hits and misses associated with a target process ID. This performance data is similarly shared with user space using eBPF maps.

The five programs we selected are representative for our evaluation because they cover key Linux subsystems, store, network, process management, and memory Management, which are fundamental to the performance and observability of distributed systems. These subsystems are critical areas where eBPF frameworks can provide valuable insights and optimizations. By targeting these diverse operations, our programs allow us to evaluate how well each framework handles a range of typical system tasks, providing a comprehensive view of their strengths, weaknesses, and overall efficiency across different types of workloads. This variety ensures that the evaluation reflects real-world usage scenarios where distributed systems rely on efficient handling of these core operations.

4.1.2 Practical Tests Performed

For the practical evaluation, two sets of performance tests were conducted on eBPF frameworks using the YCSB (Yahoo! Cloud Serving Benchmark) with the version 0.17.0[1] tool, a standard benchmarking tool widely used for database performance evaluation. Each framework was tested across two workloads: write-intensive (95 percent writes) and read-intensive (95 percent reads). For each workload, a set of nine runs was performed, ensuring a robust dataset for analysis.

The tests were conducted on MongoDB[2] v7.0.14 with two nodes, MongoDB was chosen because it is a database designed to handle large-scale, distributed workloads, which makes it a representative system for real-world scenarios in distributed environments. MongoDB's architecture involves intensive use of disk I/O, networking, memory management, and process handling, which align with the subsystems covered by our eBPF programs. Additionally, MongoDB is known for its flexible data models and horizontal scalability, which introduce varying levels of resource consumption and system interaction. Two workloads were used, designed to stress both the system's write and read capacities. They were carried out in a controlled environment using a cluster setup with 1GB of memory and 2 CPU cores running Ubuntu 22.04. To gather

[1] `https://github.com/brianfrankcooper/YCSB`
[2] `https://www.mongodb.com/pt-br`

resource consumption metrics dstat[3] was used. Vagrant was utilized to ensure root access for eBPF operations, providing the necessary privileges for monitoring and tracking.

In each run of the performance tests, all eBPF programs were activated for each framework to evaluate their full impact on system performance under the workloads. This approach ensured that the analysis captured the overhead and effects of the eBPF programs in real-world scenarios. It is important to mention, that because of framework limitations, the eBPF program that is attached to the traffic control (TC) was not run in the framework bpfTrace, as the framework currently does not support the attachment of eBPF programs to TC. Additionally, a baseline was established by running the same workloads without any eBPF programs enabled. This baseline allowed for a clear comparison, making it possible to determine the overhead and performance effects introduced by each eBPF framework.

This experimental setup allowed for a point of comparison between the eBPF frameworks under both write-heavy and read-heavy scenarios, providing valuable insights into their performance under different types of system loads. The results will serve as a basis for evaluating the efficiency and suitability of each framework for various types of workloads. We will discuss the results next on §4.2.

4.2 Results Discussion

We will first analyze the results in the read-intensive scenario and then move on to the write-intensive scenario.

4.2.1 Read Intensive Scenario

The results from the read-intensive scenario are expressed in Figures 4.1, 4.2, 4.3, 4.5, 4.4, 4.6, and 4.7.

The following tables describe the results of the experiences performed, Table 4.1 represents the time it took to perform each experiment measured in milliseconds (ms), and Table 4.2 the throughput achieved in each experiment, measured in the number of operations per second (Ops/s). The tables can be read in the same way, each line indicates the framework being tested and each column indicates the test number, and the last 2 columns represent the mean and the standard deviation regarding the nine runs performed, respectively.

In this read-intensive scenario, we analyze both time and throughput together, as they are inherently related. A decrease in time reflects an increase in throughput, and vice versa. This relationship allows us to understand how each eBPF framework impacts overall system

[3] `https://linux.die.net/man/1/dstat`

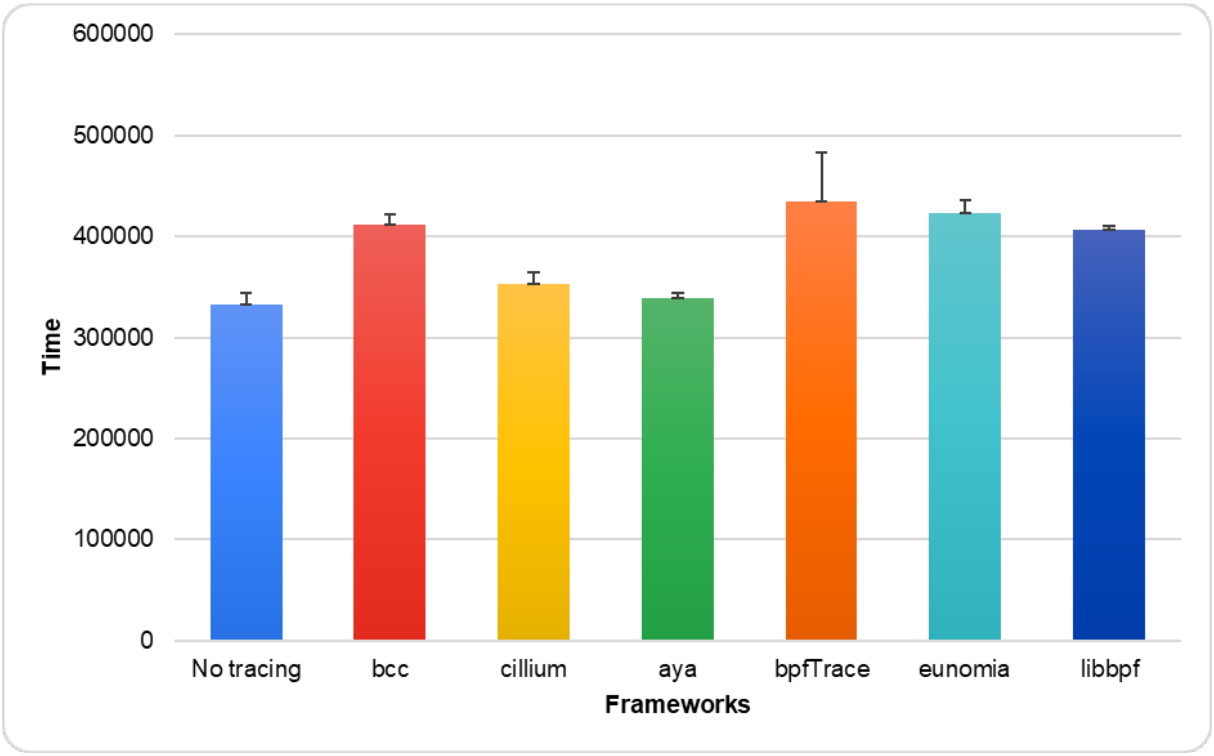

Figure 4.1: Time of experiments by framework on a read-intensive scenario in miliseconds

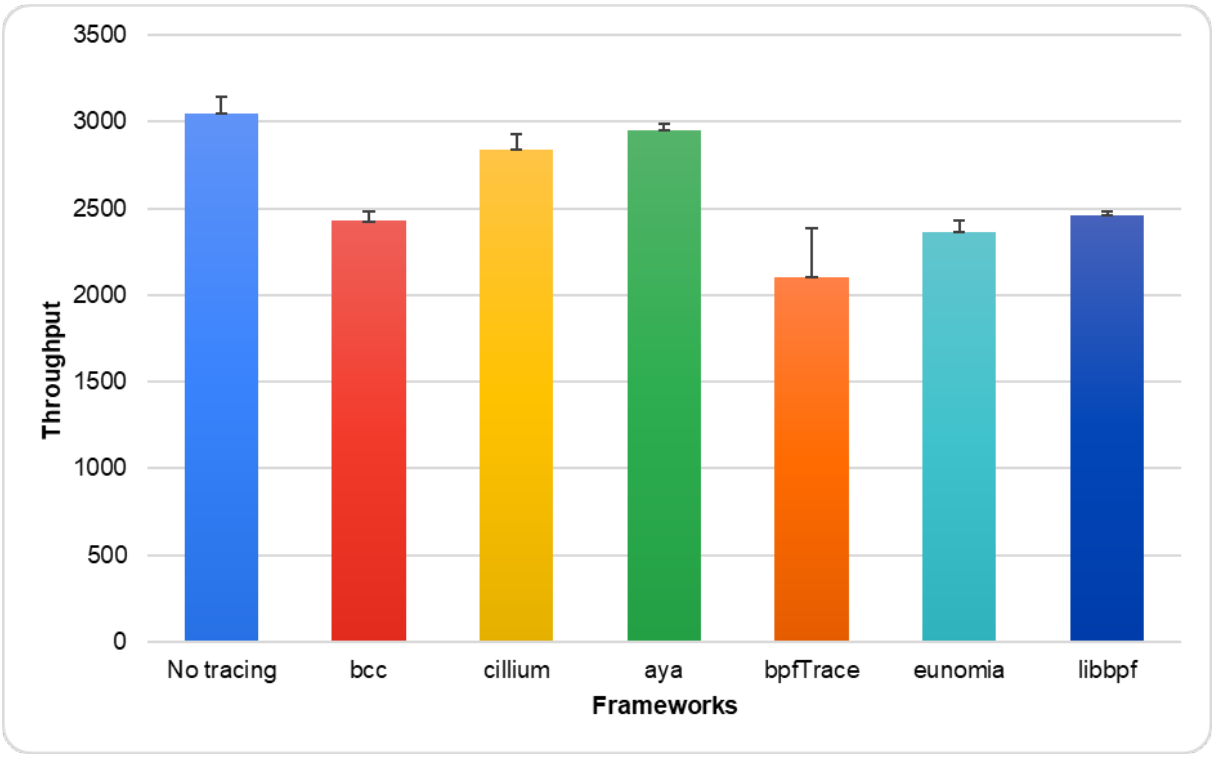

Figure 4.2: Throughput of experiments by framework on a read-intensive scenario in number of operations per second

performance. We will explore how each framework handles the workload, considering both its ability to maintain high throughput and the amount of time required to process the operations.

In the absence of tracing, our baseline, the system achieves the best possible performance

Framework	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Mean	Standard Deviation
libbpf	407697	405453	401216	402241	406736	407914	409747	413646	402686	406371	3978
bcc	409985	402990	426696	406479	401042	429617	410169	409621	412743	412149	9820
cillium	368956	345011	337825	351948	356067	341336	348946	371395	350141	352402	11483
aya	341139	345441	341202	331171	337196	333456	337342	341174	341805	338880	4491
bpfTrace	364455	463812	461590	353618	471510	487232	440398	404441	461360	434268	48542
Eunomia	446241	413841	416214	415121	412954	418821	423446	424339	441934	423656	12283
No tracing	354771	335731	33232	338546	326336	316728	318315	336340	337708	332977	11569

Table 4.1: **Time in a read-intensive scenario measured in miliseconds**

Framework	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Mean	Standard Deviation
libbpf	2452	2466	2492	2486	2459	2451	2440	2417	2483	2461	24
bcc	2439	2481	2343	2460	2493	2327	2438	2441	2422	2427	56
cillium	2710	2898	2960	2841	2808	2929	2865	2692	2855	2840	91
aya	2931	2894	2930	3019	2965	2998	2964	2931	2925	2951	39
bpfTrace	2743	2156	2166	2827	2120	2052	2270	2472	2167	2102	282
Eunomia	2240	2416	2402	2408	2421	2387	2361	2356	2262	2362	66
No tracing	3209	2978	3009	2953	3064	3157	3141	2973	2961	3049	97

Table 4.2: **Throughput in a read-intensive scenario measured in number of operations per second**

as expected, with a mean time of 332977 ms in execution time and a mean throughput of 3049 Ops/s. This scenario represents the ideal condition, where no additional overhead from eBPF frameworks is present, and the system can operate at its maximum efficiency. All the eBPF frameworks introduce some level of overhead, causing an increase in time and a corresponding decrease in throughput compared to this baseline.

Among the eBPF frameworks, Aya stands out as the best performer. It has a mean time of 338880 ms and a mean throughput of 2951 Ops/s, reflecting a very minor drop in performance compared to the no-tracing baseline. Aya's low standard deviation in both time (4491 ms) and throughput (39 Ops/s) indicates consistent and reliable performance across different runs.

Aya's ability to maintain high throughput and low time overhead shows that it introduces minimal interference with the system's normal operations. This makes Aya an excellent choice for environments where performance is critical, such as high-throughput, real-time systems. The tight coupling between time and throughput here highlights that Aya's low time overhead translates directly into its ability to process more operations per second efficiently.

Cilium has a mean time of 352402 ms and a mean throughput of 2840 Ops/s. While it introduces slightly more overhead than Aya, it still performs impressively in terms of throughput and time. The moderate standard deviation in time (11483 ms) and throughput (91 Ops/s) suggest some variability in performance, but it remains within an acceptable range for most use cases. As explained in §3, Cilium specializes in networking tasks likely contributes to its strong performance. In networking-intensive environments, Cilium's overhead is minimal enough to maintain high throughput, ensuring that packet processing and network tasks do not introduce significant delays.

Surprisingly, libbpf and BCC show similar performance profiles, it is surprising because libbpf is designed for low-level control and efficiency, whereas BCC includes higher-level abstractions that typically, introduce more overhead, so a larger performance gap was expected. libbpf records a mean time of 406371 ms and a mean throughput of 2461 Ops/s, while BCC shows a mean time of 412149 ms and a mean throughput of 2427 Ops/s. The standard deviation for libbpf in time (3978 ms) and throughput (24 Ops/s) indicates consistent performance, making it a stable option for scenarios where fine-grained control over eBPF programs is necessary. BCC shows slightly higher variability, with a standard deviation of 9820 ms for time and 56 Ops/s for throughput.

Contrary to the expected behavior explained in §3 where we expected libbpf to be more performant than BCC, it is not in this scenario, the fact that we are in a read-intensive scenario can explain this fact, as it is a less computational scenario. Nevertheless, for now, we have no clear reason to explain the underperformance of libbpf compared to Aya and Cillium.

As expected, Eunomia and bpftrace are the worst performers among the frameworks as they prioritize ease of use and simplicity, relying on higher-level abstractions, which introduce more overhead and limit their efficiency in performance-critical environments. Eunomia with a mean time of 423656 ms and a mean throughput of 2362 Ops/s, and bpftrace with a mean time of 434268 ms and a mean throughput of 2102 Ops/s. The standard deviation for Eunomia (12283 ms in time and 66 Ops/s in throughput) reflects moderate variability, suggesting that its performance is not stable enough. bpftrace has a standard deviation of 48542 ms in time and 282 Ops/s in throughput, suggesting considerable inconsistency in its performance across

different runs. Eunomia provides a good balance between simplicity and overhead, making it suitable for less critical tasks where absolute performance is not the primary concern. However, in high-performance environments, the extra overhead introduced by Eunomia can become a limiting factor, especially in scenarios requiring low latency or high throughput.

Similarly, bpftrace prioritizes ease of use and rapid development, making it a valuable tool for ad-hoc tracing and diagnostics. However, its design introduces significant overhead and variability, making it less reliable for high-performance environments that demand consistent low latency and high throughput. While useful for quick troubleshooting, bpftrace is not ideal for workloads requiring strict performance standards.

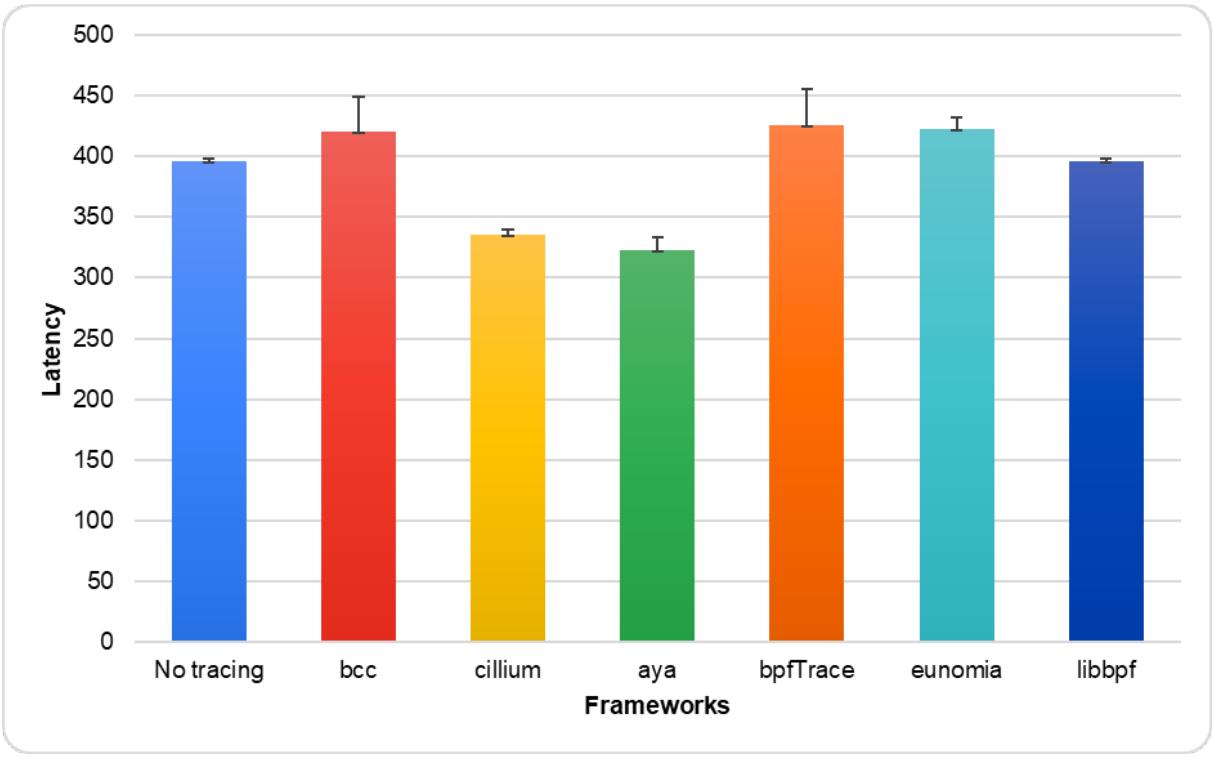

Figure 4.3: Latency of experiments by framework on a read-intensive scenario in microseconds

The following table 4.3 describes the latency measured in microseconds. Each line indicates the framework being tested and each column indicates the test number, the last 2 columns represent the mean and the standard deviation regarding the three runs performed, respectively.

Looking at the latency results, a lower latency means the system responds faster, while a higher latency indicates possible performance overhead introduced by the framework. Starting with No Tracing, the mean latency is 395 µs, with a minimal standard deviation of 2 µs. This poses as the baseline for optimal system performance without any eBPF framework overhead.

Aya exhibits the best performance among the frameworks, with a mean latency of 322.94 µs with a standard deviation of 10.63 µs. Aya introduces the least amount of overhead in the read-intensive scenario, and its tight control over variability across different runs suggests that

Framework	Run 1	Run 2	Run 3	Mean	Standard Deviation
libbpf	394	393	400	396	4
bcc	402	402	405	403	2
cillium	334	340	333	335	4
aya	310	328	329	322	10
bpfTrace	430	452	393	425	27
Eunomia	415	416	433	422	10
No tracing	398	393	397	396	2

Table 4.3: **Latency in a read-intensive scenario measured in microseconds**

it offers stable and consistent performance.

Cilium has a mean latency of 335,87 µs and a relatively low standard deviation of 3.9 µs. Although Cillium introduces more latency than Aya, its performance is still relatively strong.

libbpf introduces more overhead in this read-intensive scenario than expected, with a mean latency of 396.35 µs and a low standard deviation of 3.65 µs. While libbpf is typically known for its efficiency, its performance here suggests that it introduces more latency in read-heavy tasks compared to other frameworks like Cilium and Aya. The low standard deviation indicates consistent performance, but the higher mean latency suggests that libbpf may not be as optimized for read-intensive workloads as for write-heavy ones.

BCC exhibits similar behavior to libbpf, with a mean latency of 403.51 µs and a very low standard deviation of 1.76 µs.

Eunomia follows BCC with a mean of 422.21 µs and a standard deviation of 10.15 µs. This places Eunomia as one of the worst frameworks in terms of latency in this scenario. Eunomia introduces substantial overhead and variability.

Finally, bpftrace shows the highest latency, with a mean of 425.46 µs and a higher standard deviation of 29.73 µs. Similar to Eunomia, bpf trace is the least suitable framework for latency-sensitive, read-intensive workloads.

In the resource usage results, first, we have the percentage of user CPU usage in Table 4.4, and the percentage of system CPU usage, in Table 4.5. The next two represent the percentage of Wait I/O CPU in 4.7 and memory used in KB in 4.6.

Regarding the resource consumption metrics, in user CPU usage, the runs without tracing provide the baseline at 26.77%, which is significantly lower than any of the frameworks that introduce tracing. libbpf, BCC, Cilium, Eunomia, and Aya exhibit similar user CPU usage,

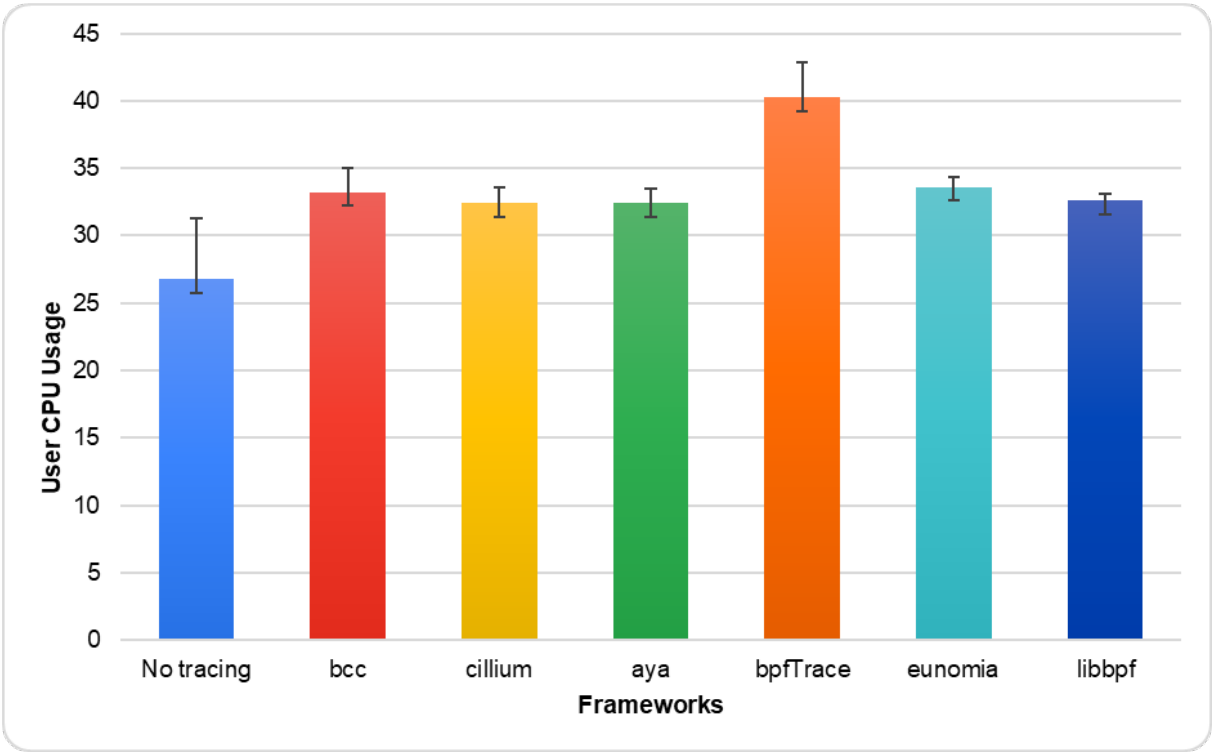

Figure 4.4: Percentage of user CPU usage by framework on a read-intensive scenario

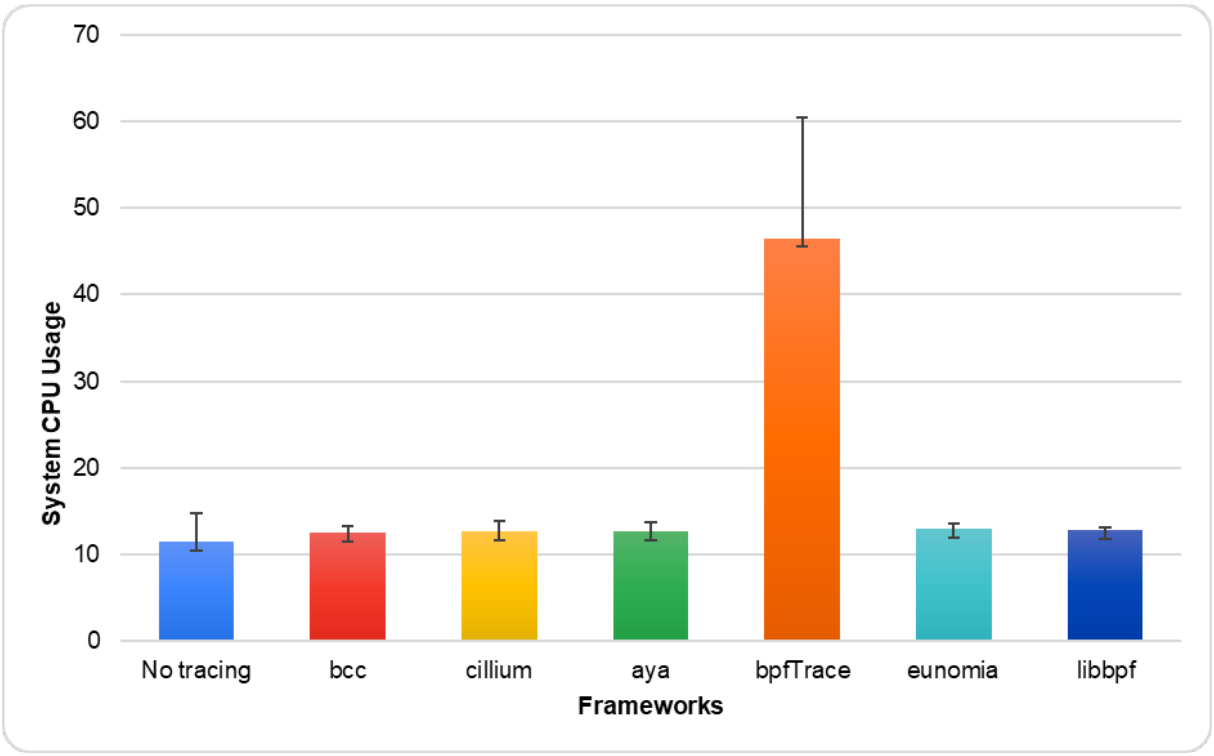

Figure 4.5: Percentage of system CPU usage by framework on a read-intensive scenario

ranging from 32.39%-33.59%, showing consistent behavior with moderate increases in user-space processing demands. These frameworks have relatively low standard deviations, indicating stable performance across runs. bpftrace has the highest user CPU usage at 40.26%, with a standard

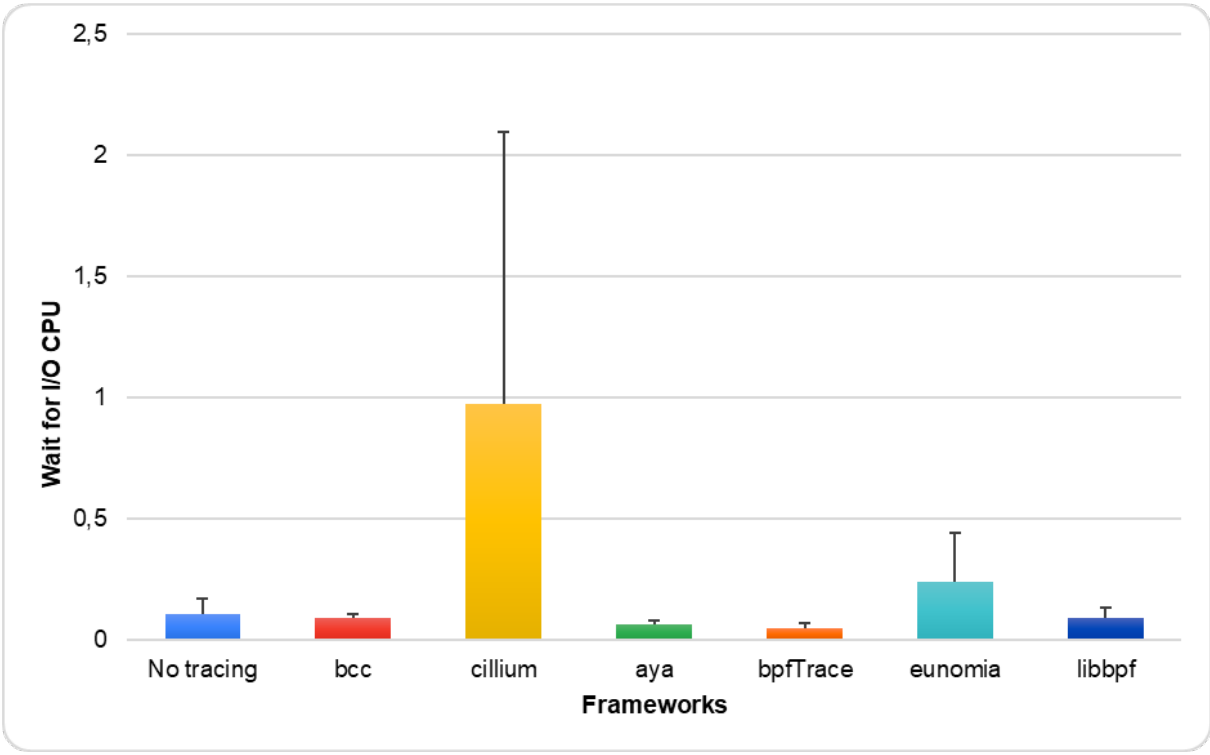

Figure 4.6: Percentage of waiting for I/O CPU by framework on a read-intensive scenario

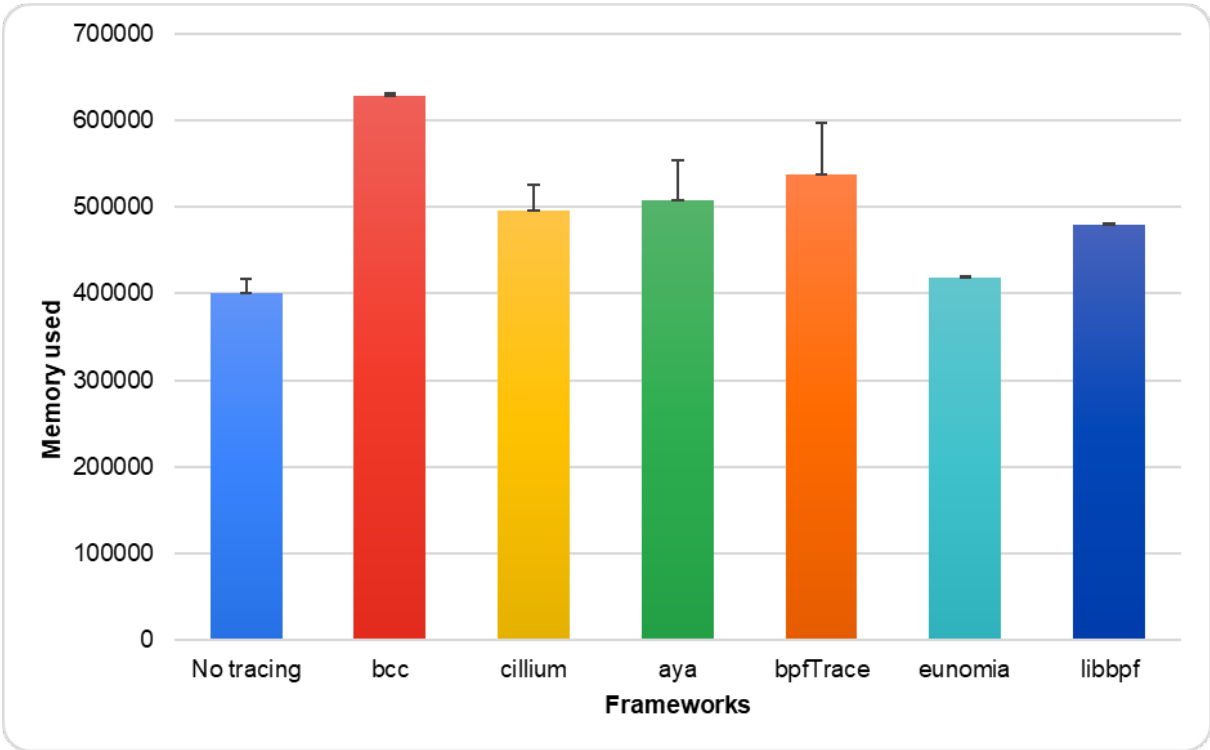

Figure 4.7: Memory used by framework on a read-intensive scenario in KB

deviation of 2.56%, showing that it requires significantly more CPU time for user-space tasks, possibly due to its higher-level abstractions and ease-of-use design.

Again, no tracing shows a baseline in system CPU usage of 11.49%, serving as the reference

Framework	Run 1	Run 2	Run 3	Mean	Standard Deviation
libbpf	38.2	32.94	32.08	32.60	0.46
bcc	32.94	35.15	31.6	33.23	1.79
cillium	33.74	32.09	31.35	32.39	1.22
aya	33.6	32.22	31.44	32.42	1.09
bpfTrace	42.71	40.47	37.61	40.26	2.56
Eunomia	33.39	32.97	34.41	33.59	0.74
No tracing	29.59	29.17	21.56	26.77	4.52

Table 4.4: **Percentage of user CPU usage in a read-intensive scenario**

Framework	Run 1	Run 2	Run 3	Mean	Standard Deviation
libbpf	12.78	13.18	12.57	12.84	0.3
bcc	12.62	11,84	13.21	12.56	0.69
cillium	13.57	13.05	11.2	12.61	1.25
aya	12,63	13.71	11.77	12.70	0.97
bpfTrace	30.98	50.9	57.61	46.50	13.85
Eunomia	12.31	12.99	13.59	12.96	0.64
No tracing	12.34	14.22	7.91	11.49	3.24

Table 4.5: **Percentage of system CPU usage in a read-intensive scenario**

for evaluating the additional overhead introduced by each framework. libbpf, BCC, Cilium, Eunomia and Aya have relatively close system CPU usage percentages ranging between 12.56%-12.96%, indicating that these frameworks introduce moderate kernel-level overhead. Their standard deviations are also low, indicating consistent behavior in kernel-space operations. bpftrace again stands out with 46.49% system CPU usage, with a large standard deviation of 13.85%, showing that it incurs a significant system overhead and exhibits substantial variability between runs.

No tracing shows a baseline of 0.11% for wait I/O CPU. libbpf, BCC, bpftrace, and Aya all exhibit shallow I/O wait times, ranging from 0.05% to 0.09%, indicating that these frameworks do not introduce significant delays in I/O operations, even under read-heavy conditions. Their low standard deviations show that they handle I/O consistently across runs. Cilium shows

Framework	Run 1	Run 2	Run 3	Mean	Standard Deviation
libbpf	480564	477627	479343	479178	1475
bcc	624238	629277	630547	628021	3337
cillium	48124	476478	530066	495929	29659
aya	462377	505368	555229	507658	46468
bpfTrace	477127	539846	595621	537531	59280
Eunomia	417374	418055	419932	418454	1324
No tracing	414239	405791	383206	401079	16044

Table 4.6: **Memory used in a read-intensive scenario measured in kB**

Framework	Run 1	Run 2	Run 3	Mean	Standard Deviation
libbpf	0.14	0.06	0.07	0.09	0.044
bcc	0.1	0.08	0.1	0.093	0.012
cillium	0.19	2.26	0.47	0.973	1.123
aya	0.06	0.08	0.06	0.067	0.012
bpfTrace	0.05	0.07	0.03	0.05	0.02
Eunomia	0.46	0.2	0.06	0.24	0.203
No tracing	0.09	0.05	0.18	0.107	0.067

Table 4.7: **Percentage of waiting I/O CPU in a read-intensive scenario**

a higher 0.97% wait I/O CPU usage, with a higher standard deviation of 1.12%, suggesting that it introduces more variability in I/O operations and may incur additional I/O overhead in certain runs. Eunomia also shows an increased wait I/O CPU at 0.24%, but its higher standard deviation of 0.20% indicates variability, though still not as extreme as Cilium.

Finally, for memory usage, no tracing shows the baseline memory usage at 401079 KB. libbpf and Cilium show moderate memory usage at 479178 KB and 495929 KB respectively, with libbpf demonstrating the lowest standard deviation (1475 KB), suggesting very stable memory consumption. BCC stands out with the highest memory usage at 628021 KB, with a standard deviation of 3337 KB, indicating that its higher-level abstractions consume more memory, even though its performance remains consistent. Aya also shows relatively high memory consumption at 507658 KB, with a larger standard deviation of 46468 KB, indicating variability in memory

usage across runs. bpftrace uses 537531 KB, and the 59280 KB standard deviation indicates substantial variability in memory consumption, further reflecting its inconsistency. Eunomia shows the lowest memory consumption among the frameworks at 418454 KB, with a standard deviation of 1324 KB, indicating consistent performance with efficient memory use.

4.2.2 Write Intensive Scenario

In a write-intensive scenario, both time and throughput metrics provide insight into how efficiently each eBPF framework manages tasks. Write-heavy workloads typically introduce higher system demands, and the relationship between time and throughput helps us understand the performance impact of using different frameworks. This analysis considers both the mean and standard deviation for time, throughput and latency to evaluate consistency and efficiency, reflected in both 4.8, 4.9, and 4.10.

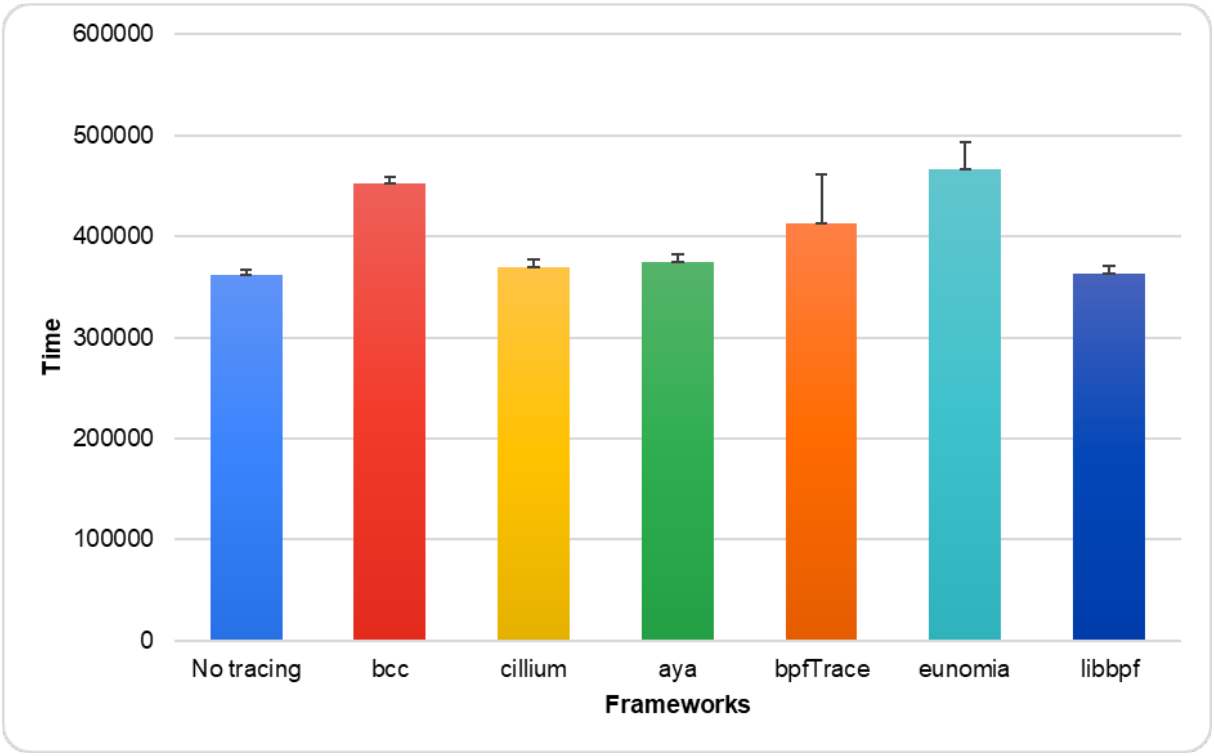

Figure 4.8: Time of experiments by framework on a write-intensive scenario in miliseconds

The following tables describe the results of the experiences performed, Table 4.8 represents the time it took to perform each experiment measured in milliseconds (ms), and Table 4.9 the throughput achieved in each experiment, measured in the number of operations per second (Ops/s).

Again, without any eBPF tracing enabled, the system performs with the lowest mean time of 361228 ms and the highest throughput of 2769 Ops/s. This reflects the maximum system efficiency, as no additional tracing overhead impacts performance. The low standard deviation

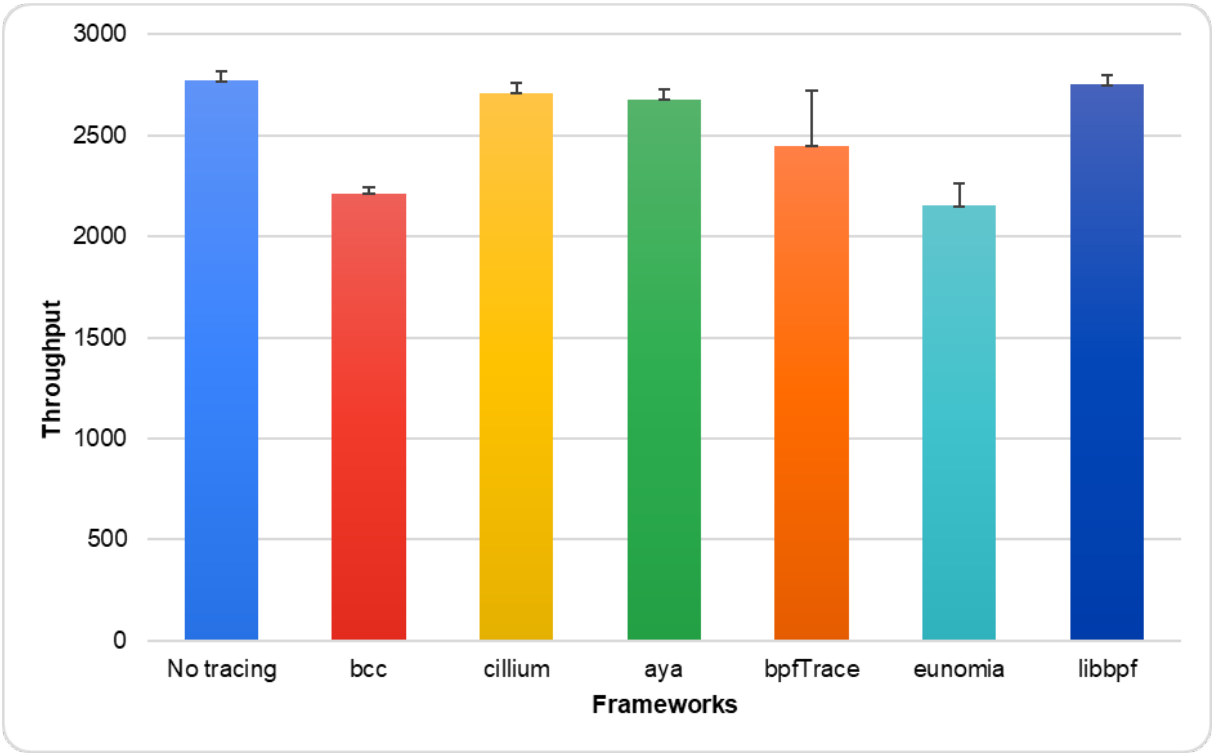

Figure 4.9: Throughput of experiments by framework on a write-intensive scenario in number of operations per second

Framework	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Mean	Standard Deviation
libbpf	352093	368836	356321	359472	367738	366571	372445	366453	363822	363750	6538
bcc	441773	456400	456328	455780	451675	445705	449582	452102	461212	452284	5986
cillium	377561	373389	374080	356352	360872	370596	373034	365512	375223	369624	7164
aya	368783	366029	366661	375630	379091	384839	384834	374435	365476	373975	7753
bpfTrace	382679	371418	362812	468599	483965	379844	396805	476317	395421	413095	48700
Eunomia	459238	452961	443029	460705	460912	454839	536012	469067	458383	466127	27135
No tracing	367077	357227	362706	360967	372582	363594	354218	359412	353272	361228	6151

Table 4.8: **Time in a write-intensive scenario measured in miliseconds**

for both time (6151 ms) and throughput (46 Ops/s) also confirms that the system behaves consistently in this baseline scenario. This sets the benchmark for comparing how each framework introduces overhead and affects performance.

libbpf offers strong performance in this write-intensive scenario, with a mean time of 363750 ms and a throughput of 2749 Ops/s, closely mirroring the baseline performance. The low standard deviation in time (6538 ms) and throughput (49 Ops/s) show that libbpf provides stable

Framework	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Mean	Standard Deviation
libbpf	2840	2711	2806	2781	2719	2727	2684	2728	2748	2749	49
bcc	2263	2191	2191	2194	2213	2243	2224	2211	2168	2211	29
cillium	2648	2678	2673	2806	2771	2698	2680	2735	2665	2706	53
aya	2711	2732	2727	2662	2637	2598	2598	2670	2736	2674	55
bpfTrace	2613	2692	2756	2134	2066	2632	2520	2099	2528	2449	272
Eunomia	2177	2207	2257	2170	2169	2198	1865	2131	2181	2151	112
No tracing	2724	2799	2757	2770	2683	2750	2823	2782	2830	2769	46

Table 4.9: **Throughput in a write-intensive scenario measured in number of operations per second**

and consistent results across multiple runs. The marginal increase in time and a corresponding slight decrease in throughput compared to the no-tracing baseline indicates that libbpf introduces minimal overhead, making it an efficient choice for write-intensive tasks.

Cilium also performs well in this scenario, with a mean time of 369624 ms and a throughput of 2706 Ops/s. The moderate standard deviation in time (7164 ms) and throughput (53 Ops/s) shows reasonable consistency.

Aya records a mean time of 373975 ms and a throughput of 2674 Ops/s, placing it slightly behind libbpf and Cilium. The moderate standard deviation in time (7753 ms) and throughput (55 Ops/s) indicates consistent performance.

Surprisingly, bpfTrace demonstrates a quite low overhead regarding the expectations, recording a mean time of 413095 ms and a throughput of 2449 Ops/s. This is unexpected given bpfTrace high-level abstractions, which typically introduce significant performance overhead, but in this case, it manages to perform more efficiently than anticipated in handling write-heavy tasks. The large standard deviations in time (48700 ms) and throughput (272 Ops/s) reflect its inconsistency, a result of its focus on ease of use and rapid development. Despite its performance in mean values, its high standard deviations confirm that bpftrace is not well-suited for high-performance, write-intensive workloads, where more efficient and consistent processing is necessary.

As expected, BCC introduces more overhead than the other frameworks analyzed thus far, with a mean time of 452284 ms and a throughput of 2175 Ops/s. This represents a significant drop in throughput compared to the no-tracing scenario, as well as increased time overhead.

The moderate standard deviation in time (10242 ms) and throughput (45 Ops/s) shows that BCC delivers consistent performance, though at a cost of higher overhead. This is likely due to BCC's higher-level abstractions, which simplify development but introduce additional processing demands.

As expected, Eunomia exhibits higher overhead and more variability in write-intensive scenarios, confirming its limitations in such demanding environments.

Eunomia, with a mean time of 466127 ms and a throughput of 2151 Ops/s, shows the most significant overhead. The high standard deviation in time (27135 ms) and throughput (112 Ops/s) further highlights the variability across different runs. Eunomia's focus on simplicity and ease of deployment contributes to this performance trade-off, making it less ideal for high-throughput, write-heavy tasks, though it may still be suitable for less critical environments.

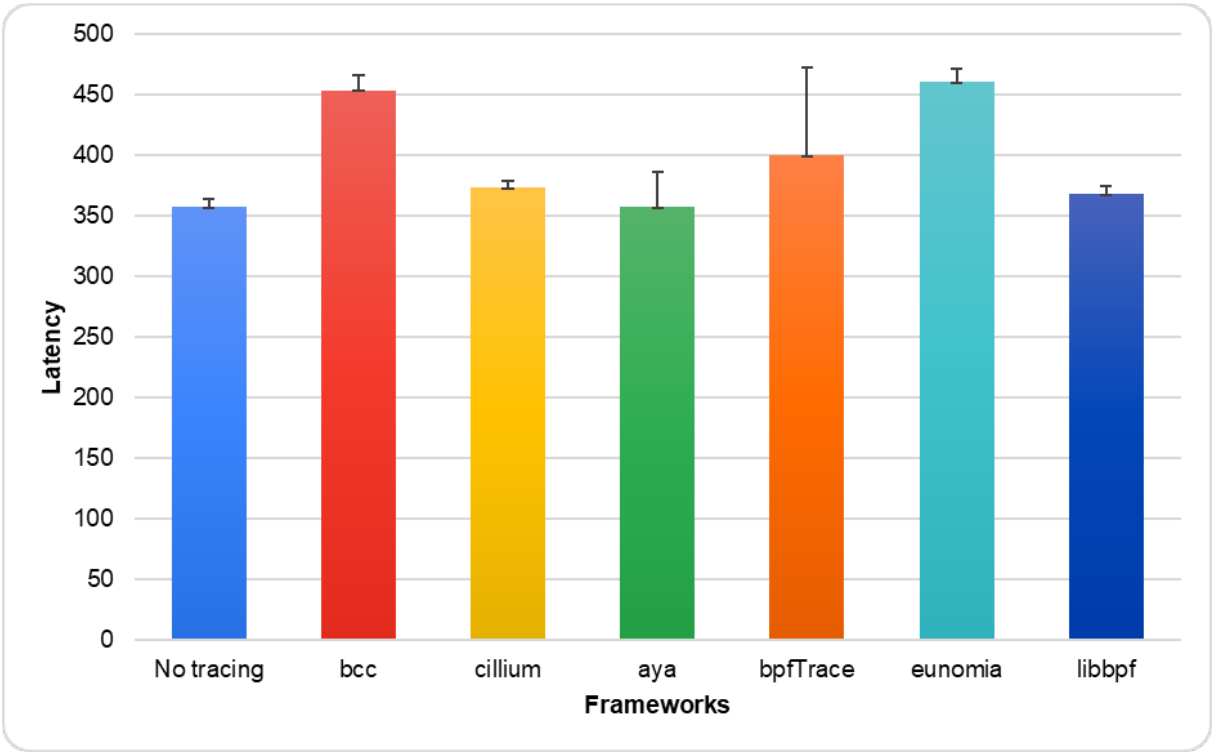

Figure 4.10: Latency of experiments by framework on a write-intensive scenario in microseconds

The following table 4.10 describes the latency measured in microseconds. Each line indicates the framework being tested and each column indicates the test number, and the last 2 columns represent the mean and the standard deviation regarding the three runs performed, respectively.

Looking at the latency results, and starting with No Tracing, which serves as the baseline for system performance, we observe a mean latency of 357.55 µs, with a low standard deviation of 6.04 µs. This represents the system's optimal performance without the overhead introduced by any eBPF framework. It sets the reference for how much overhead each framework adds when applied in this write-intensive environment.

Framework	Run 1	Run 2	Run 3	Mean	Standard Deviation
libbpf	374	367	360	367	7
bcc	439	462	458	453	12
cillium	373	367	377	372	5
aya	326	380	365	357	28
bpfTrace	475	329	393	399	73
Eunomia	448	468	463	460	10
No tracing	358	362	351	357	6

Table 4.10: **Latency in a write-intensive scenario measured in microseconds**

Aya, with a mean latency of 357.55 µs, matches the no-tracing scenario. The fact that Aya introduces almost no additional latency in this workload confirms its efficient design. Its standard deviation, at 27.78 µs, is slightly higher, indicating some variability in performance across different runs. However, this variability remains relatively small, showing that Aya maintains consistent low-latency performance even in write-heavy environments.

libbpf also performs well, with a mean latency of 367.69 µs. Its standard deviation of 6.69 µs is very low, indicating that libbpf delivers stable and predictable performance across multiple runs.

Cilium performs almost equally to libbpf, with a mean latency of 372.90 µs. Its standard deviation of 5.26 µs is the lowest among all frameworks, showing that Cilium delivers highly consistent performance with minimal variation across runs. This low variability is a significant strength, as it ensures that Cilium provides predictable latency.

Moving to bpftrace, we see a mean latency of 399.37 µs, higher than the previously discussed frameworks. The standard deviation of 72.90 µs is also significantly larger, indicating considerable variability in its performance across runs. The large variation in latency makes bpftrace less reliable in performance-sensitive environments, as it cannot consistently maintain low-latency execution.

BCC follows with a mean latency of 453.26 µs, the increase in latency is expected given BCC's higher-level abstractions. The standard deviation of 12.33 µs indicates that BCC's performance is relatively stable, though the mean latency is considerably higher than the more optimized frameworks.

Finally, Eunomia shows the highest latency, with a mean of 460.19 µs, this suggests that Eunomia introduces overhead in write-intensive environments, making it the least efficient frame-

work for such tasks. The standard deviation of 10.45 µs indicates low variability, showing that while Eunomia's performance is somewhat stable, its higher latency consistently places it at the bottom regarding efficiency.

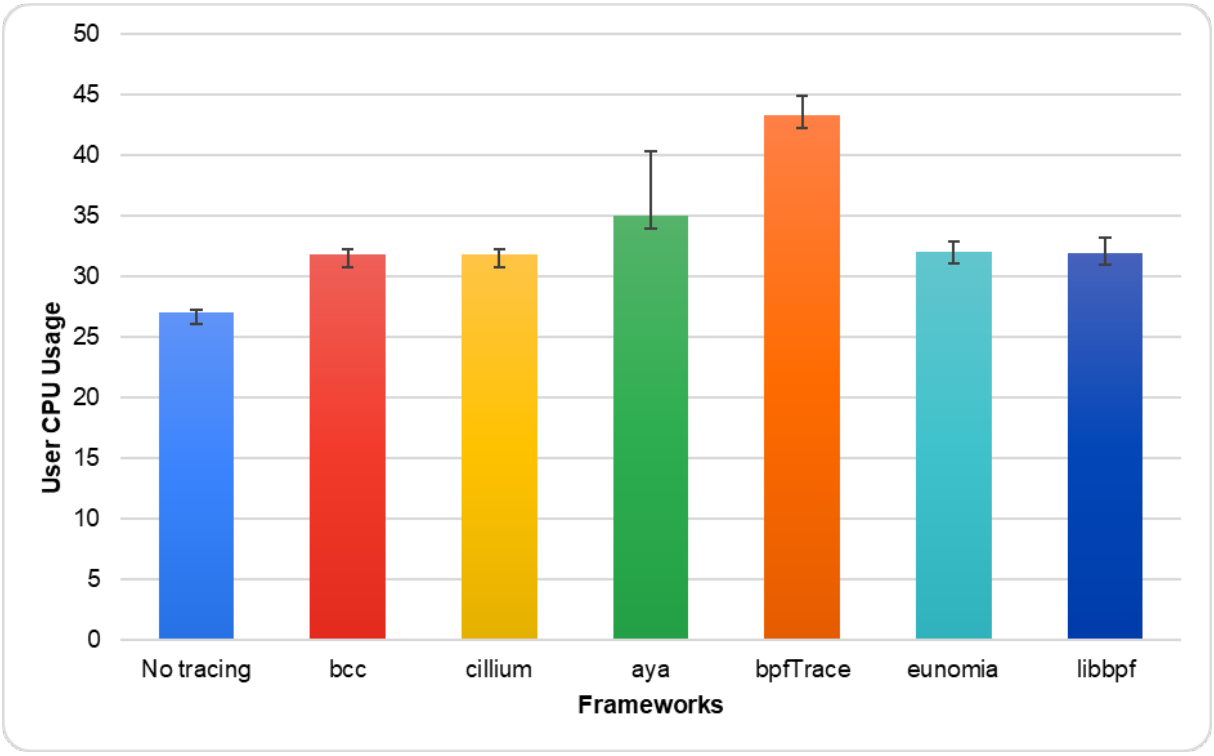

Figure 4.11: Percentage of user CPU usage by framework on a write-intensive scenario

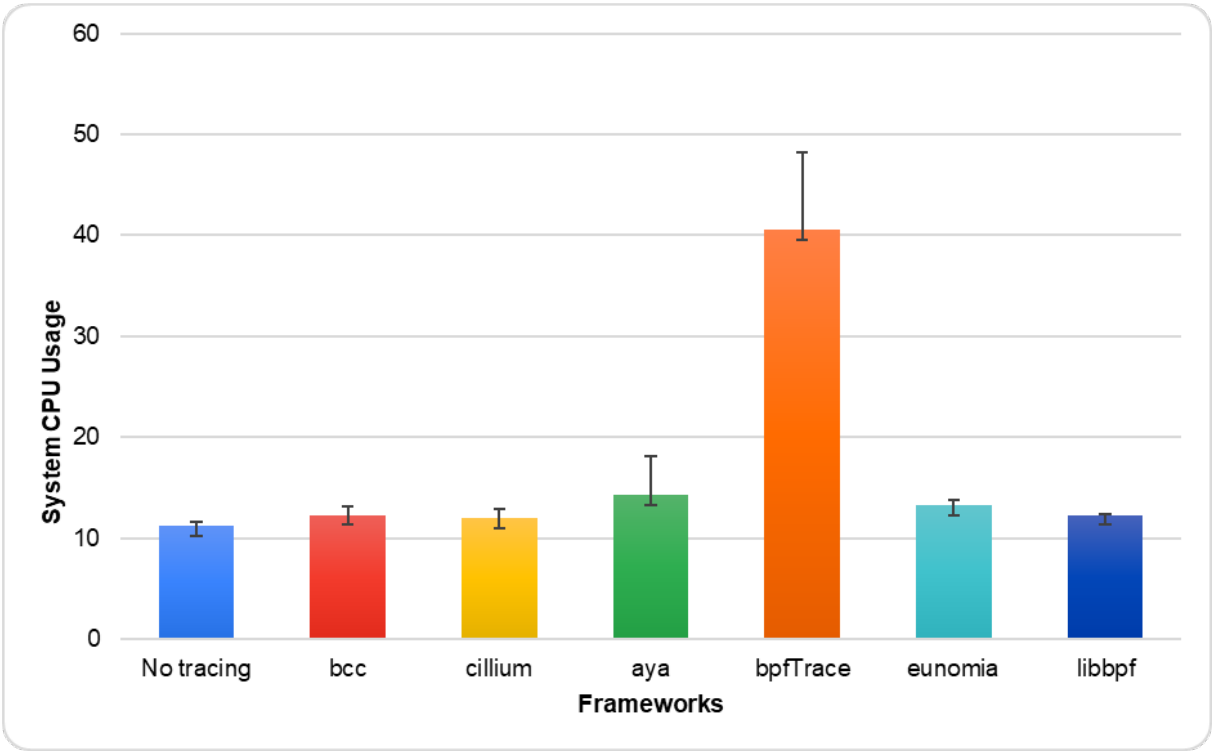

Figure 4.12: Percentage of system CPU usage by framework on a write-intensive scenario

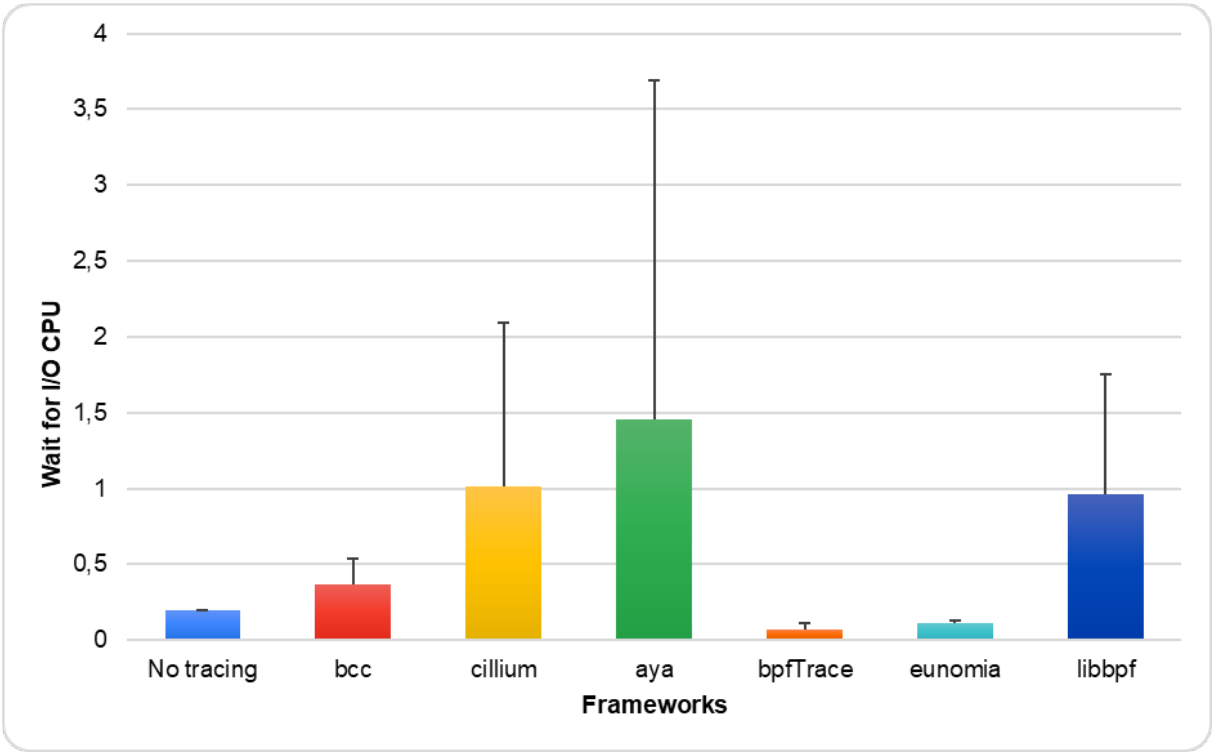

Figure 4.13: Percentage of waiting for I/O CPU by framework on a write-intensive scenario

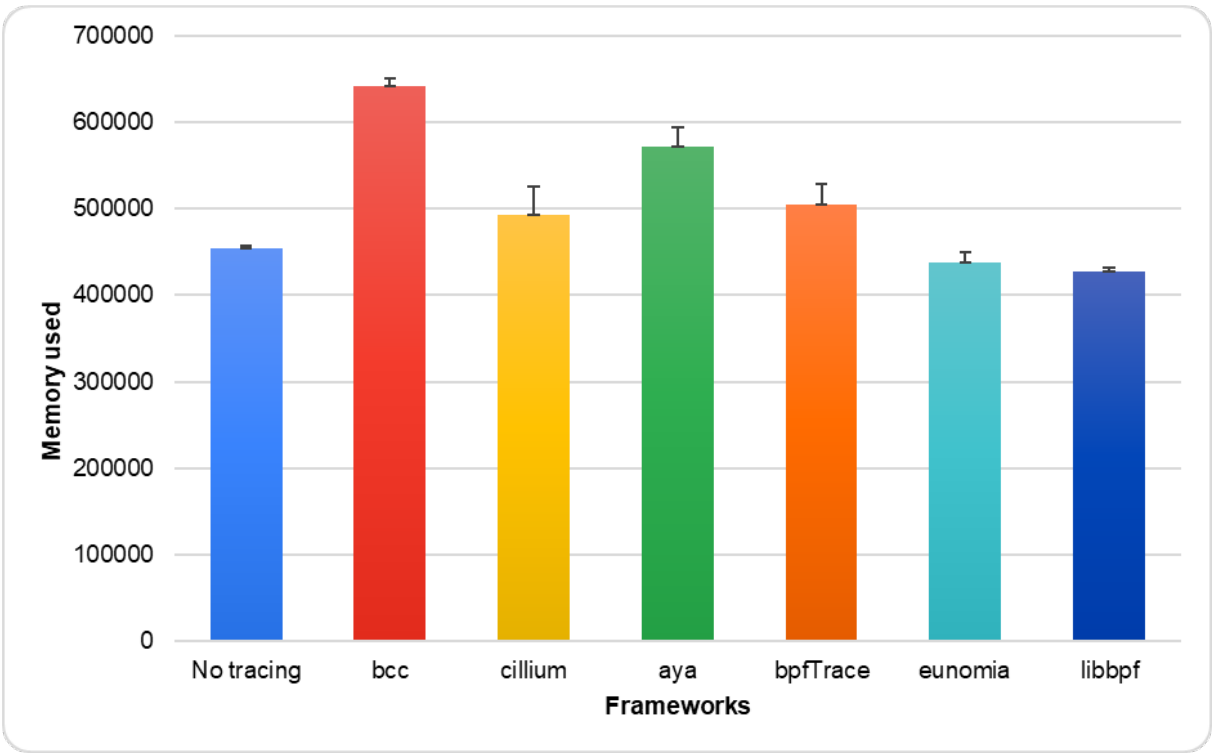

Figure 4.14: Memory used by framework on a write-intensive scenario in KB

In the resource usage results, first, we have the percentage of user CPU usage in Table 4.11, and the percentage of system CPU usage, in Table 4.12. The next two represent the percentage of Wait I/O CPU in 4.14 and memory used in KB in 4.13.

Framework	Run 1	Run 2	Run 3	Mean	Standard Deviation
libbpf	31.96	30.61	33.09	31.89	1.24
bcc	31.8	32.14	31.26	31.74	0.45
cillium	31.89	32.09	31.35	31.78	0.38
aya	41.08	32.45	31.23	34.92	5.37
bpfTrace	44.91	43.16	41.65	43.24	1.63
Eunomia	31.89	31.31	32.93	32.04	0.82
No tracing	27.07	27.14	26.87	27.03	0.14

Table 4.11: **Percentage of system CPU usage in a write-intensive scenario**

Framework	Run 1	Run 2	Run 3	Mean	Standard Deviation
libbpf	12.26	12.21	12.46	12.31	0.13
bcc	11.93	11.79	13.23	12.32	0.79
cillium	11.61	13.05	11.2	11.95	0.97
aya	18.72	12.13	12.14	14.33	3.80
bpfTrace	33	40.39	48.27	40.55	7.64
Eunomia	13	13.06	13.91	13.32	0.51
No tracing	11.56	11.33	10.84	11.24	0.37

Table 4.12: **Percentage of system CPU usage in a write-intensive scenario**

Still in the write-intensive scenario, regarding the resource consumption metrics, the baseline with no tracing has the lowest user CPU usage at 27.03% with very low variability, serving as a reference for the additional overhead introduced by the eBPF frameworks. libbpf, BCC, Cilium, and Eunomia perform similarly, showing user CPU usage around 31.74%-32.04%, with minimal variability across runs. This indicates a moderate increase in user CPU usage compared to the no-tracing baseline, which is expected due to the tracing overhead. Aya shows a higher user CPU usage at 34.92%, with a higher standard deviation of 5.37%, indicating some inconsistency across runs and slightly more resource demand in user-space tasks. bpftrace stands out with the highest user CPU usage at 43.24%, which significantly exceeds the other frameworks, indicating that it demands more CPU resources for user-space processing.

In the metric system CPU usage, again, no tracing provides the baseline with 11.24% sys-

Framework	Run 1	Run 2	Run 3	Mean	Standard Deviation
libbpf	422170	430647	428459	427092	4400
bcc	632067	642214	650472	641584	9218
cillium	473283	476478	530066	493276	31901
aya	595527	567707	549861	571032	23014
bpfTrace	481496	502194	529081	504257	23859
Eunomia	431797	430600	451731	438043	11869
No tracing	450291	454222	456353	453622	3075

Table 4.13: **Memory used in a write-intensive scenario measured in kB**

Framework	Run 1	Run 2	Run 3	Mean	Standard Deviation
libbpf	1.01	0.14	1.73	0.96	0.80
bcc	0.56	0.26	0.27	0.36	0.17
cillium	0.3	2.26	0.47	1.01	1.09
aya	4.04	0.22	0.1	1.45	2.24
bpfTrace	0.03	0.09	0.1	0.07	0.038
Eunomia	0.09	0.12	0.12	0.11	0.02
No tracing	0.19	0.2	0.19	0.19	0.01

Table 4.14: **Percentage of waiting I/O CPU in a write-intensive scenario**

tem CPU usage, representing the optimal case with no eBPF overhead. Also again, libbpf, BCC, and Cilium exhibit similar performance, with system CPU usage around 11.95%-12.32%, indicating minimal overhead for kernel-level operations. This suggests that these frameworks efficiently manage system resources. Aya and Eunomia show slightly higher system CPU usage at 13.32%-14.33%, with Aya showing a higher standard deviation of 3.80%, reflecting more variability. bpftrace shows significantly higher system CPU usage at 40.55%, with a large standard deviation of 7.64%, indicating that it demands far more kernel-level processing time and exhibits substantial variability in performance across different runs.

No tracing again serves as the baseline with 0.19% CPU waiting for I/O. libbpf, BCC, and Cilium show relatively low values, with 0.36%-1.01%, indicating that these frameworks introduce minimal waiting time for I/O operations, suggesting they handle I/O efficiently even

under write-heavy loads. Aya has the highest wait I/O CPU at 1.45%, with a high standard deviation of 2.24%, indicating more inconsistency and potentially longer waits for I/O during certain runs, which could impact overall performance. bpftrace shows very low wait I/O CPU usage at 0.07%, though its significantly higher user and system CPU usage suggests that this framework introduces overhead elsewhere, compensating for minimal I/O delays. Eunomia also has low wait I/O usage at 0.11%, with minimal variability, indicating consistent performance in terms of I/O wait times.

Finally, the runs without tracing result in 453,622 KB of memory used, representing the baseline memory consumption. libbpf, Cilium, and Eunomia show moderate memory usage between 427,092 KB and 493,276 KB, with Cilium exhibiting higher variability (standard deviation of 31,901 KB). These frameworks demonstrate efficient memory management without significant memory overhead. bpftrace uses 504,257 KB of memory, showing that despite its higher CPU usage, its memory footprint remains relatively moderate. BCC stands out with the highest memory usage at 641,584 KB, suggesting that its higher-level abstractions result in more memory consumption. Aya also consumes a considerable amount of memory at 571,032 KB, with a standard deviation of 23,014 KB.

4.2.3 Discussion

One of the notable inconsistencies observed in the analysis is the performance of libbpf in read-intensive scenarios. Contrary to expectations, libbpf performs similarly to BCC in this case, with both frameworks showing a significant amount of overhead compared to Aya and Cilium. Given that libbpf is known for its low-level control and efficiency, it was expected to perform better in read-intensive tasks, but the results indicate that it introduces more overhead than anticipated in this context. It is important to note that libbpf's latency in the read scenario was higher than expected, this might indicate that libbpf struggles with higher read frequencies although it still maintains relatively low-latency processing in these operations. In the write-intensive scenario, libbpf performs much better, offering near-baseline performance and standing out as one of the top frameworks alongside Cilium. This inconsistency suggests that libbpf's efficiency may be more workload-dependent, excelling in write-heavy tasks but not so much in read-heavy environments. It could also be due to several other factors that remain unclear.

Another unexpected result was bpftrace's higher performance in the write-intensive scenario than in the read-intensive tasks. Given bpftrace's design, which prioritizes ease of use and high-level scripting over raw performance, it was expected to introduce significant overhead in both scenarios. However, in the write-intensive workload, bpftrace performed better than

anticipated, showing reduced overhead and higher throughput compared to the read-intensive scenario. Nevertheless, bpftrace introduces a significant amount of overhead in system CPU usage.

The resource consumption analysis reveals that while BCC and bpftrace are powerful frameworks for eBPF development, their performance is significantly hindered by resource inefficiencies. BCC's high memory usage across both read and write-intensive scenarios is one of its key drawbacks. Despite its overall consistent CPU performance, the substantial memory overhead indicates that BCC's higher-level abstractions come at a cost. This higher memory consumption reduces its suitability for high-throughput environments, particularly in resource-constrained systems where memory efficiency is critical. Similarly, bpftrace suffers from high CPU usage, particularly in the system space. In both scenarios, it demonstrated the highest system CPU usage, which, combined with significant variability across runs, highlights its inefficiency in managing CPU resources. This high CPU overhead is particularly problematic in performance-sensitive environments where minimal system interference is required.

The last unexpected result was that some frameworks exhibited lower latency than the baseline. However, this behavior can be explained by operation batching. The overhead introduced by eBPF leads to batches being flushed with fewer operations compared to the baseline. As a result, the mean latency appears lower because each batch is processed more quickly due to its smaller size. However, this comes at the cost of reduced throughput, as fewer operations are completed in each batch cycle, thus lowering the overall system efficiency despite the improved latency. This trade-off between latency and throughput highlights another impact of eBPFs on system performance.

Regarding the expected results, given that Rust's performance is generally on par with C in terms of system-level programming, it was expected that Aya, by being written entirely in Rust, would be one of the most performant eBPF frameworks. This expectation is confirmed across read and write-intensive scenarios, where Aya consistently performs near the top. In both workloads, Aya demonstrated minimal overhead, maintaining high throughput and low latency, reinforcing its position as a high-performance framework. This shows that Aya's Rust-based implementation does not introduce significant performance penalties, and it benefits from Rust's safety features without compromising on speed, making it a strong choice for performance-critical environments.

Another expected result was Cilium's strong performance across both read and write-intensive scenarios, it was anticipated to handle workloads efficiently. This expectation was confirmed, as Cilium consistently performed well, demonstrating low overhead, latency, and high throughput

in both scenarios. Its design, which leverages eBPF for optimized networking tasks, proved effective, further validating Cilium as a reliable choice for environments where networking performance and scalability are critical. The performance of libbpf in the write-intensive scenario also met expectations, demonstrating its efficiency and low overhead in handling high-volume operations. Due to its close integration with the kernel and minimal abstraction layers, libbpf managed the write-heavy workload with minimal performance degradation.

The resource consumption metrics also highlight Aya's and Cilium's strong performance. Despite slightly higher memory usage compared to other frameworks, Aya shows minimal CPU usage increases. This demonstrates Aya's efficiency, balancing higher memory consumption with minimal CPU overhead, thus maintaining stable performance across different workloads. Similarly, Cilium continues to demonstrate strong efficiency, confirming its ability to handle complex networking tasks without heavily burdening system resources. The same happens for libbpf, confirming its strong overall performance. With moderate user CPU usage, low system CPU usage, minimal I/O wait times, and stable memory usage, libbpf demonstrates an efficient balance of resource consumption. These metrics reinforce its suitability for performance-critical environments, as it consistently handles both read and write-intensive workloads with minimal overhead and variability.

Despite bpftrace's surprising improvement in the write-intensive scenario, it still exhibited significant inconsistency across both scenarios, with high variability in performance. On top of that, bpftrace also shows other drawbacks such as its high CPU usage and its limitation to attach to the TC hook. Eunomia also showed substantial overhead and variability like bpftrace, these frameworks were, as expected, the worst performers overall. Both frameworks prioritize ease of use and deployment over raw performance, which aligns with their lower throughput and higher time overhead compared to more optimized frameworks like libbpf, Aya, and Cilium.

In both write and read-intensive scenarios, libbpf, Aya, and Cilium stand out for their efficiency and consistency, while BCC, Eunomia, and bpftrace exhibit higher overhead and variability, making them less suitable for performance-critical tasks.

On one side, in the write-intensive scenario, libbpf leads with minimal overhead and high throughput, making it ideal for demanding, write-heavy workloads. Cilium is close behind, with strong performance and moderate variability, especially in network-heavy environments. Aya performs well but introduces slightly more overhead. BCC shows more overhead, reducing its suitability for high-throughput tasks, though it remains consistent. Eunomia and bpftrace struggle the most, with high overhead and variability, making them inefficient for write-intensive tasks.

On the other, aya excels in read-heavy tasks, providing high throughput and low latency. Cilium performs well but with slightly more variability. libbpf and BCC offer moderate performance, with BCC suffering from high memory usage, and libbpf showing better consistency. Eunomia balances simplicity and performance but suffers from variability, while bpftrace is the least efficient, with high latency and poor throughput.

4.3 Conclusion

This practical analysis allows us to evaluate eBPF frameworks in a key aspect, performance. It focuses on real-world performance, measuring the overhead in actual scenarios like read and write-intensive operations. By benchmarking the frameworks, we can validate the theoretical expectations and understand how they perform in practice, revealing potential deployment complexities or limitations.

It provides a comprehensive evaluation, ensuring that our conclusions help achieve the thesis's objective of determining the most effective eBPF framework for specific use cases. Through this rigorous testing, we can draw reliable insights that guide both the theoretical understanding and practical application of eBPF technologies, ensuring optimal performance in diverse real-world environments.

Chapter 5

Conclusions

In distributed systems, the challenge of achieving comprehensive observability is paramount. The axiom "You can't improve what you can't measure" encapsulates the essence of this challenge.

The state of the art still falls short in two key areas: full system observability and the lack of a clear, structured guide on how to effectively work with eBPF to achieve specific system goals, current research does not sufficiently address how to best leverage these tools.

Our work is focused on this second gap, providing a detailed, theoretical, and practical guide on how to utilize eBPF to meet various system requirements. By examining both theoretical and practical aspects of it, we aim to offer insights into optimizing its use in real-world scenarios.

5.1 Achievements

In this work, we have successfully addressed the gap in the state of the art by providing a comprehensive theoretical and practical analysis of eBPF frameworks. Our research covers various key areas such as tracing, networking, security, storage, and scheduling, offering insights into the strengths and limitations of popular frameworks like libbpf, BCC, Cilium, and others.

Through this analysis, we have created a clear guide that helps understand how to select and utilize eBPF frameworks effectively based on their system goals. By balancing theoretical knowledge with real-world testing and use cases, our work offers practical solutions for implementing eBPF in performance-critical environments, cloud-native applications, and security-sensitive systems. This guide serves as a valuable resource for anyone looking to harness the full potential of eBPF in achieving optimized system observability, performance, and security.

5.2 Future Work

As future work, several aspects of eBPF implementation remain to be explored in greater detail. One key area is the implementation of user and kernel space counters to reliably trace lost events, which would improve the accuracy of system monitoring and help identify performance bottlenecks. We could also explore analyzing each eBPF program independently rather than running all programs simultaneously. This approach would allow for a more granular evaluation of the performance impact and overhead of each specific eBPF program. By isolating their effects, we could gain insights into the individual contributions and potential bottlenecks that each program introduces. This more detailed analysis would enhance our understanding of how each program performs under different workloads, helping to fine-tune deployment strategies for specific scenarios.

Additionally, a more thorough analysis of certain performance costs is needed. This includes studying the cost of in-kernel computation and the cost of communication with user space, such as the overhead involved with using ring buffers, maps, and other mechanisms for data transfer between kernel and user-space programs.

Finally, there is the potential to build a kernel profiling tool that can assess the in-kernel execution paths of applications. Such a tool would provide deeper insights into the performance and behavior of applications within the kernel, helping to optimize their efficiency and resource usage. These future directions will further enhance the understanding and utilization of eBPF in system optimization.

Bibliography

[1] Systemtap. https://sourceware.org/systemtap/.

[2] Daniel Borkmann Alexei Starovoitov. Extended berkeley packet filter. https://ebpf.io/.

[3] 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, Renton, WA, July 2019. USENIX Association.

[4] Adam Leventhal Bryan Cantrill, Mike Shapiro. Dtrace. https://dtrace.org/.

[5] Inspektor Gadget Contributors. Inspektor gadget. https://www.inspektor-gadget.io/. Contributors: https://github.com/inspektor-gadget/inspektor-gadget/graphs/contributors. Accessed: 2023-12-31.

[6] Jörg Domaschka, Simon Volpert, Kevin Maier, Georg Eisenhart, and Daniel Seybold. Using ebpf for database workload tracing: An explorative study. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*, ICPE '23 Companion, page 311–317, New York, NY, USA, 2023. Association for Computing Machinery.

[7] Tânia Esteves, Francisco Neves, Rui Oliveira, and João Paulo. Cat: Content-aware tracing and analysis for distributed systems. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, page 223–235, New York, NY, USA, 2021. Association for Computing Machinery.

[8] Tânia Esteves, Ricardo Macedo, Rui Oliveira, and João Paulo. Toward a practical and timely diagnosis of application's i/o behavior. *IEEE Access*, 11:110184–110207, 2023.

[9] William Findlay, Anil Somayaji, and David Barrera. bpfbox: Simple precise process confinement with ebpf. In *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*, CCSW'20, page 91–103, New York, NY, USA, 2020. Association for Computing Machinery.

[10] Niclas Hedam, Morten Tychsen Clausen, Philippe Bonnet, Sangjin Lee, and Ken Friis Larsen. Delilah: ebpf-offload on computational storage. In *Proceedings of the 19th International Workshop on Data Management on New Hardware*, DaMoN '23, page 70–76, New York, NY, USA, 2023. Association for Computing Machinery.

[11] Yasukata Kenichi, Tazaki Hajime, and Aublin Pierre-Louis. zpoline: a system call hook mechanism based on binary rewriting. *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, July 2023.

[12] Chunghan Lee, Reina Yoshitani, and Toshio Hirotsu. Enhancing packet tracing of microservices in container overlay networks using ebpf. In *Proceedings of the 17th Asian Internet Engineering Conference*, AINTEC '22, page 53–61, New York, NY, USA, 2022. Association for Computing Machinery.

[13] Chang Liu, Zhengong Cai, Bingshen Wang, Zhimin Tang, and Jiaxu Liu. A protocol-independent container network observability analysis system based on ebpf. In *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 697–702, 2020.

[14] Alessandro Rivitti, Roberto Bifulco, Angelo Tulumello, Marco Bonola, and Salvatore Pontarelli. ehdl: Turning ebpf/xdp programs into hardware designs for the nic. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 208–223, New York, NY, USA, 2023. Association for Computing Machinery.

[15] Mohsen Salehi and Karthik Pattabiraman. Poster autopatch: Automatic hotpatching of real-time embedded devices. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 3451–3453, New York, NY, USA, 2022. Association for Computing Machinery.

[16] Wanqi Yang, Pengfei Chen, Kai Liu, and Huxing Zhang. Nahida: In-band distributed tracing with ebpf, 2023.

[17] Zhe Yang, Youyou Lu, Xiaojian Liao, Youmin Chen, Junru Li, Siyu He, and Jiwu Shu. u-IO: A unified IO stack for computational storage. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 347–362, Santa Clara, CA, February 2023. USENIX Association.

[18] Jesson Yo and Achmad Imam Kistijantoro. Analyzing fair share fairness of tasks in the linux

completely fair scheduler using ebpf. In *2023 10th International Conference on Advanced Informatics: Concept, Theory and Application (ICAICTA)*, pages 1–6, 2023.

[19] Sepehr Abbasi Zadeh, Ali Munir, Mahmoud Mohamed Bahnasy, Shiva Ketabi, and Yashar Ganjali. On augmenting tcp/ip stack via ebpf. In *Proceedings of the 1st Workshop on EBPF and Kernel Extensions*, eBPF '23, page 15–20, New York, NY, USA, 2023. Association for Computing Machinery.

[20] Long Zhang, Brice Morin, Benoit Baudry, and Martin Monperrus. Maximizing error injection realism for chaos engineering with system calls. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2695–2708, 2022.

[21] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. XRP: In-Kernel storage functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, Carlsbad, CA, July 2022. USENIX Association.

[22] Hao Zhou, Shuohan Wu, Xiapu Luo, Ting Wang, Yajin Zhou, Chao Zhang, and Haipeng Cai. Ncscope: hardware-assisted analyzer for native code in android apps. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 629–641, New York, NY, USA, 2022. Association for Computing Machinery.

[23] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. Electrode: Accelerating distributed protocols with eBPF. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1391–1407, Boston, MA, April 2023. USENIX Association.

[24] Yang Zhou, Xingyu Xiang, Matthew Kiley, Sowmya Dharanipragada, and Minlan Yu. DINT: Fast In-Kernel distributed transactions with eBPF. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 401–417, Santa Clara, CA, April 2024. USENIX Association.

Appendix A

eBPF programs

In this appendix, we present each eBPF program implemented, using the examples from libbpf, the first program is the eBPF program that attaches to the open system call.

```
#include "../../.github/actions/build-selftests/vmlinux.h"
#include <bpf/bpf_helpers.h>

struct pid_open_info {
    u32 pid;
    u64 open_count;
    u64 call_count;
} __attribute__((packed));

struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __type(key, u32);
    __type(value, struct pid_open_info);
    __uint(max_entries, 64);
} open_calls SEC(".maps");

struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __type(key, u32);
    __type(value, u32);
    __uint(max_entries, 1);
} target_pid SEC(".maps");
```

```
SEC("tracepoint/syscalls/sys_enter_openat")
int trace_open(const struct pt_regs *ctx)
{
    u32 pid = bpf_get_current_pid_tgid() >> 32;
    u32 key = 0;
    u32 *target_pid_val = bpf_map_lookup_elem(&target_pid, &key);

    // Return if no target PID is set or current PID does not match targeta PID
    if (!target_pid_val || *target_pid_val != pid) {
        return 0;
    }

    struct pid_open_info *value, zero = {0};

    value = bpf_map_lookup_elem(&open_calls, &pid);
    if (value) {
        value->call_count++;
        if (value->call_count % 5 == 0) {
            value->pid = pid;
            value->open_count++;
            value->call_count = 0;
        }
    } else {
        struct pid_open_info new_value = {.pid = pid, .open_count = 1, .call_count = 1};
        bpf_map_update_elem(&open_calls, &pid, &new_value, BPF_ANY);
    }

    return 0;
}

char LICENSE[] SEC("license") = "GPL";
```

The second program is the eBPF program that attaches to the system call that creates processes.

```
#include "../../.github/actions/build-selftests/vmlinux.h"
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_core_read.h>

#define TASK_COMM_LEN 16

struct event {
    int pid;
    int ppid;
    int uid;
    int retval;
    bool is_exit;
    char comm[TASK_COMM_LEN];
};

struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __type(key, u32);
    __type(value, u32);
    __uint(max_entries, 1);
} target_pid SEC(".maps");

struct {
    __uint(type, BPF_MAP_TYPE_LRU_HASH);
    __type(key, u32);
    __type(value, struct event);
    __uint(max_entries, 256);
} events SEC(".maps");

SEC("tracepoint/syscalls/sys_enter_execve")
int tracepoint_syscalls_sys_enter_execve(struct trace_event_raw_sys_enter* ctx) {
    struct event event = {0};
    u64 id;
    pid_t tgid;
```

```
    char comm[TASK_COMM_LEN];
    struct task_struct *task = (struct task_struct *)bpf_get_current_task();
    id = bpf_get_current_pid_tgid();
    tgid = id >> 32;

    u32 *target_pid_val = bpf_map_lookup_elem(&target_pid, 0);

    if (!target_pid_val || *target_pid_val != tgid) {
        return 0;
    }

    event.pid = tgid;
    event.uid = bpf_get_current_uid_gid();

    struct task_struct *parent_task;
    bpf_probe_read(&parent_task, sizeof(parent_task), &task->real_parent);

    bpf_probe_read(&event.ppid, sizeof(event.ppid), &parent_task->tgid);

    if (bpf_probe_read_str(&event.comm, sizeof(event.comm), (void *)ctx->args[0]) == 0) {
        // Store the event data into a BPF map
        bpf_map_update_elem(&events, &tgid, &event, BPF_ANY);
    }

    return 0;
}

char LICENSE[] SEC("license") = "GPL";
```

The third program is the eBPF program which counts cache hits and misses.

```
#include <linux/types.h>
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

typedef unsigned int u32;
```

```
struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __type(key, u32);
    __type(value, u32);
    __uint(max_entries, 4);
} counters SEC(".maps");

struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __type(key, u32);
    __type(value, u32);
    __uint(max_entries, 1);
} target_pid SEC(".maps");

#define CACHE_HIT 0
#define CACHE_MISS 1
#define CACHE_DIRTY 2

SEC("kprobe/mark_page_accessed")
int count_hit(struct pt_regs *ctx) {
    u32 pid = bpf_get_current_pid_tgid() >> 32;
    u32 key_map = 0;
    u32 *target_pid_val = bpf_map_lookup_elem(&target_pid, &key_map);

    if (!target_pid_val || *target_pid_val != pid) {
        return 0;
    }

    u32 key = CACHE_HIT;
    u32 init_val = 1;
    u32 *value;

    value = bpf_map_lookup_elem(&counters, &key);
```

```
    if (value)
        *value += 1;
    else
        bpf_map_update_elem(&counters, &key, &init_val, BPF_ANY);

    return 0;
}

SEC("kprobe/add_to_page_cache_lru")
int count_miss(struct pt_regs *ctx) {
    u32 pid = bpf_get_current_pid_tgid() >> 32;
    u32 key_map = 0;
    u32 *target_pid_val = bpf_map_lookup_elem(&target_pid, &key_map);

    if (!target_pid_val || *target_pid_val != pid) {
        return 0;
    }

    u32 key = CACHE_MISS;
    u32 init_val = 1;
    u32 *value;

    value = bpf_map_lookup_elem(&counters, &key);
    if (value)
        *value += 1;
    else
        bpf_map_update_elem(&counters, &key, &init_val, BPF_ANY);

    return 0;
}

char LICENSE[] SEC("license") = "Dual BSD/GPL";
```

The fourth program is the eBPF program which filters network traffic at the TC level.

```
#include <linux/types.h>
```

```
#include <bpf/bpf_helpers.h>
#include <linux/ip.h>
#include <linux/bpf.h>
#include <linux/pkt_cls.h>
#include <linux/if_ether.h>
#include <linux/tcp.h>
#include <bpf/bpf_endian.h>

#ifndef IPPROTO_TCP
#define IPPROTO_TCP 6
#endif

typedef unsigned int u32;

struct packet_data {
    u32 src_ip;
    u32 dst_ip;
    char payload[64];
};

struct pid_data {
    struct packet_data data[1024];
    u32 counter;
};

struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __type(key, u32);
    __type(value, struct pid_data);
    __uint(max_entries, 1);
} last_packet SEC(".maps");

SEC("classifier")
int http_filter(struct __sk_buff *skb) {
```

```
u32 key = 0;
u32 proto;

bpf_skb_load_bytes(skb, offsetof(struct ethhdr, h_proto), &proto, sizeof(proto));
if (proto != bpf_htons(ETH_P_IP))
    return TC_ACT_OK;

struct iphdr iph;
bpf_skb_load_bytes(skb, ETH_HLEN, &iph, sizeof(iph));
bpf_printk("packet, Proto: %u, %u", iph.protocol, IPPROTO_TCP);
if (iph.protocol != IPPROTO_TCP)  // Ensure it's TCP
    return TC_ACT_OK;

struct tcphdr tcph;
int tcp_header_size = ETH_HLEN + (iph.ihl * 4);
bpf_skb_load_bytes(skb, tcp_header_size, &tcph, sizeof(tcph));

struct packet_data pkt_data = {};
pkt_data.src_ip = iph.saddr;
pkt_data.dst_ip = iph.daddr;

int payload_offset = tcp_header_size + (tcph.doff * 4);

struct pid_data *pid_entry;

pid_entry = bpf_map_lookup_elem(&last_packet, &key);
if (!pid_entry)
    return TC_ACT_OK;
if (pid_entry->counter == 1024) {
    pid_entry->counter = 0;
}
struct packet_data *pkt_data = &pid_entry->data[pid_entry->counter];

if (bpf_skb_load_bytes(skb, payload_offset, &pkt_data->payload, 64) == 0) {
```

```
        pkt_data->src_ip = iph.saddr;
        pkt_data->dst_ip = iph.daddr;

        // Increment the counter for the next packet
        pid_entry->counter += 1;
        bpf_map_update_elem(&last_packet, &key, pid_entry, BPF_ANY);
    }
    return TC_ACT_OK;
}

char _license[] SEC("license") = "GPL";
```

The fifth and last program is the eBPF program that attaches to the read-and-write system calls.

```
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>
#include <linux/if_ether.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <linux/udp.h>

typedef unsigned int u32;

struct data_t {
    u32 pid;
    u32 tid;
    char comm[16];
    char type;
};

struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __type(key, u32);
    __type(value, u32);
    __uint(max_entries, 1);
```

```
} target_pid SEC(".maps");

struct {
    __uint(type, BPF_MAP_TYPE_RINGBUF);
    __uint(max_entries, 1 << 24);
} events SEC(".maps");

int trace_rw(char type) {
    u32 pid = bpf_get_current_pid_tgid() >> 32;
    u32 key = 0;
    u32 *target_pid_val = bpf_map_lookup_elem(&target_pid, &key);

    if (!target_pid_val || *target_pid_val != pid) {
        return 0;
    }

    struct data_t *data = bpf_ringbuf_reserve(&events, sizeof(struct data_t), 0);
    if (!data)
        return 0;

    data->pid = bpf_get_current_pid_tgid() >> 32;
    data->tid = bpf_get_current_pid_tgid() & 0xFFFFFFFF;
    bpf_get_current_comm(&data->comm, sizeof(data->comm));
    data->type = type;

    bpf_ringbuf_submit(data, 0);
    return 0;
}

SEC("tracepoint/syscalls/sys_enter_write")
int trace_write(struct pt_regs *ctx) {
    return trace_rw('W');
}
```

```
SEC("tracepoint/syscalls/sys_enter_read")
int trace_read(struct pt_regs *ctx) {
    return trace_rw('R');
}

char _license[] SEC("license") = "GPL";
```