INSTITUTE FOR INFORMATION TECHNOLOGY

of the LUDWIG–MAXIMILIANS–UNIVERSITY MUNICH

Bachelors thesis

A Symphony of Metrics: Assessing the Advantages of eBPF over conventional Benchmarking Tools

Daniel Schneider

INSTITUTE FOR INFORMATION TECHNOLOGY

of the LUDWIG–MAXIMILIANS–UNIVERSITY MUNICH

Bachelors thesis

A Symphony of Metrics: Assessing the Advantages of eBPF over conventional Benchmarking Tools

Daniel Schneider

Supervisor:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	Maximilian Höb
Deadline:	January 31, 2024

Abstract

Supercomputers are configured to handle demanding calculations, and their architecture varies based on the workloads they run. Some workloads involve extensive data communication between instances, requiring a high grade of network traffic throughput capabilities of the cluster and an optimized architecture for intercommunication of these workloads. Other workloads may require more hardware-based resources, such as high amounts of system memory, compute performance, and disk speeds, calling for cluster architectures optimized for these resource types. In general, It can be said, that various cluster architectures have their own benefits and non-benefits, on which informed decisions have to be made.

Choosing the ideal cluster architecture for a particular workload has been a topic of extensive research. The decision-making process begins with gathering behavioral data on the workload; acting on this data is a separate field of study with various approaches. Most systems utilize a user-space program that can benchmark the workload to make its behavior more measurable.

Recent work on the extended Berkeley Packet Filter (eBPF) has made it possible to benchmark at a lower level within the kernel space. eBPF provides more control over resources and better access to operating system functions and information that are not easily accessible through user-space programs.

This thesis aims to assess eBPF's advantages in Linux systems compared to conventional benchmarking solutions. We will develop an eBPF-based benchmarking reference-implementation to collect per-process usage statistics, including CPU, Memory, and Disk utilization. Our assessment will compare this system against traditional solutions, focusing on correctness, modularity, simplicity, and scalability.

Contents

1 Introduction

The emergence of *eBPF* (extended Berkley Packet Filter) in recent years has redefined the landscape of system monitoring and networking as can be seen in [ebp24a]. eBPF offers advanced capabilities for various tasks, guaranteeing optimal performance across several environments. This technology has been accepted in many sectors, including software development, enterprise-level data science applications, and cloud-based systems, thanks to its scalability and flexibility, and thus is already fully used in large-scale production-level systems for its advanced networking capabilities such as the *Express Data Path* (XDP)[1], IP packet inspection, and system observation as outlined in [ebp24b][TR17].

For scientific institutions, particularly those engaged in scientific computing, the requirement for efficient tools to benchmark different workloads is especially important for scheduling optimization and cost reduction. As eBPF represents a relatively new technology, its usage, especially regarding system *observability*, has been relatively limited compared to conventional benchmarking systems that have been used for benchmarking for multiple decades already[2]. Its nature makes monitoring more complex and requires time and dedication to implement for these use cases. This aspect is highly important in *high-performance computing* (HPC) and *high-throughput computing* (HTC) environments, where optimal performance is critical. Efficient monitoring and understanding of task behavior in these settings can lead to major savings in energy and time, underlining the importance of eBPF's potential in these application areas as the authors have described in [ALRP15].

1.1 Contribution

The search for efficient and performant benchmarking tools is highly important in software performance analysis. eBPF has gotten significant attention among the new technologies in this domain due to its unique approach and capabilities. However, with new technology comes challenges in adoption, implementation, and comparison with already established methodologies. This thesis aims to assess the advantages of eBPF over conventional benchmarking tools, highlighting its position as a milestone technology in system observation and performance analysis. While doing so, the following challenges must be faced:

- **Complexity:** One of the primary challenges in this field is understanding the complexity and intricacies of eBPF. Unlike traditional tools, eBPF operates at a lower level of the system, offering deep insights into system performance. On the other hand, this also means that eBPF is often more complex and requires a deeper understanding of

[1]The Express Data Path is an eBPF based system that allows for highly performant sending and receiving of network packets for low-level packet inspection and mutation, allowing the user to circumvent OS-level restrictions[Jes23]

[2]The `top` tool, for example, was initially included with 4BSD Unix, which has been released around the year 1980 as can be seen in this file tree [4bs24]

system internals. This complexity impacts its usability and makes a comparison to conventional tools challenging.

- **Comparison:** The technology landscape in performance benchmarking is rapidly evolving. While conventional tools have a well-established place in performance analysis and rely on predefined interfaces, of the Linux OS[3], eBPF's introduction has shifted this approach though its unopinionated nature, requiring a fresh perspective on benchmarking applications.

- **Criteria:** Another hurdle in this comparative assessment is establishing objective and fair criteria. eBPF and traditional benchmarking tools are designed with different goals and strengths, making direct comparisons non-trivial. This thesis seeks to bridge this gap by outlining a set of criteria that fairly evaluate the capabilities of both methodologies.

- **Limitations:** eBPF poses a set of technical limitations that must be overcome, as mentioned in [Din23]. These challenges include but are not limited to program size, instruction count, and control flow. Building a usable eBPF benchmarking system will require workarounds to these challenges—more on this in the background and implementation section.

In sum, this thesis aims to assess the technical advantages of eBPF and contribute to understanding its practical implications in the landscape of performance benchmarking tools and its potential role in future performance analysis.

1.2 Outline

The *Background* chapter 2 discusses the underlying technological foundations on which a capable eBPF benchmarking system will be built by highlighting the core working principles of conventional benchmarking systems and eBPF and the possibilities this technology could bring. It will also discuss the technological feasibility and limitations of eBPF, how one can work around them, and what exactly eBPF could measure to facilitate a benchmarking system for applications.

Following that, other *Related Work* to this thesis will be reviewed in chapter 3 and will highlight the work that has already gone into this area of applications and how this thesis adds to their findings.

The following *Methodology* chapter 4 compiles a list of criteria that will be considered when building up the system design and later assessing this benchmarking system to other conventional solutions.

In the *Design* chapter 5, a clear path will be outlined to building this eBPF benchmarking system by applying common patterns in software engineering. It will first discuss the procedure for collecting measurements such as CPU utilization, memory usage, and disk I/O and how it will incorporate these different concepts into a more extensive system that allows for the programmatic consumption of these measurements.

[3]These interfaces can be found under, e.g., the `/proc` directory and other areas of the system accessible by the user

After evaluating the design of this system, a reference solution will be *Implemented* 6 utilizing standard technological foundations of eBPF development and other systems' patterns. As this thesis does not focus on implementation specifics, this chapter will only guide a rough path to the implementation, highlighting areas of interest and tough-to-retrieve information.

Having successfully implemented a reference solution, an *Assessment* 7 on how such an eBPF-based benchmarking system compares to other similar profiling systems in terms of correctness, simplicity, modularity, and scalability will be done by evaluating the extracted metrics and comparing them to the results of conventional benchmarking tools.

In the final *Conclusionary* 8 chapter, a discussion on the overall findings of the assessment can be found, which highlights the strengths and weaknesses of such an eBPF-based system and how such a system can be extrapolated to build more advanced solutions as well as lay a foundation upon which future work could be built on.

2 Background

Benchmarking is a process used to measure the performance of a system or component, typically in comparison with an established standard or other systems. It involves observing system behavior during different workloads to estimate criteria such as speed, quality, or efficiency. In computing, benchmarking is often used to compare the performance of hardware, software, or network capabilities. The aim is to clearly understand a system's performance and thus inform decisions about improvements, upgrades, or system configurations. The background chapter will first lay out the basics of benchmarking by describing how conventional benchmarking tools work 2.1, after which an explaination of the core pricinples of eBPF can be found in the section 2.2 also focusing on tracing technologies 2.3 and profiling 2.4.

2.1 Conventional benchmarking tools

Conventional user-space benchmarking tools employ methodologies to evaluate the performance of multiple components of a computer system, including CPU, memory, and I/O subsystems, from the user space without requiring deep kernel-level access. This is done by accessing predefined host Operating System interfaces such as the Linux `/proc` directory.

In the following, a non-exhaustive list of the most popular user-space benchmarking tools can be found:

1. **htop** is an interactive system monitor that provides a detailed view of ongoing processes, CPU, memory usage, and other system metrics.
2. **iostat** is used for monitoring system input/output device load by observing the time devices are active in relation to their average transfer rates.
3. **vmstat** reports information about processes, memory, paging, block IO, traps, and CPU activity.
4. **mpstat** is used to monitor CPU utilization.
5. **perf** is a performance analyzing tool in Linux for detailed performance monitoring.
6. **Sysstat** is a collection of performance monitoring tools for Linux, including sar, iostat, and mpstat.
7. **dstat** is a versatile tool for generating system resource statistics.
8. **top** displays a real-time view of a running system, including tasks the kernel manages.

Each tool specializes in different aspects of system performance, from CPU and memory usage to I/O operations. These systems can be widely used across different hardware and operating system configurations.

The advantages of these tools are their portability and Ease of Use: User-space benchmarking tools are generally easier to deploy and use, as they do not require kernel-level access or modifications other than accessing OS interfaces designed for that purpose. This makes them suitable for many users, from system administrators to application developers. They also provide some guarantees inherent to their user-space bounded execution:

- **Safety:** Operating in user space reduces the risk of system crashes or instability, as these tools are less likely to interfere with critical kernel operations.
- **Reusability:** They can be used across different operating system distributions and hardware configurations, making them ideal for comparative analysis and cross-platform performance evaluations.
- **Stability:** By relying on OS interfaces to gather their data, they are very stable across different OS kernel versions. The OS, which defines these stable interfaces, also ensures that they do not break implementations across different versions, which cannot be said about other tracing technologies.

However, there are limitations to user-space benchmarking. Due to their limited access to the system's inner workings, they may not provide as detailed or low-level insights as kernel-level tools (like eBPF). Technologies such as *perf* have enabled deeper insight into hardware-level events that are otherwise not readily available through software-defined OS interfaces, but even technologies such as *perf* reach their limitations quickly when it comes to very specific use cases that lie outside of typical usage scenarios, as well as being limited to specific processor architectures, kernel versions and its "everything goes into the kernel" philosophy (V. M. Weaver 2013, last page). [Wea13]

In summary, conventional user-space benchmarking systems are essential tools in performance analysis, offering a balance of safety, ease of use, and versatility, suitable for a broad range of performance evaluation needs in various environments. While having these guarantees, they are relatively limited in their capabilities, as they are—most of the times—built with specific use cases in mind and thus rarely offer any extensibility and easy modification.

2.2 The extended Berkley Packet Filter

The Extended Berkeley Packet Filter is an advanced version of the original Berkeley Packet Filter. Initially, BPF was designed to filter network packets efficiently in the kernel, which is the core part of an operating system. However, eBPF has expanded its capabilities far beyond just filtering packets.

It is used for a variety of important tasks in the kernel, which include:

- **Performance Monitoring:** eBPF helps closely watch and analyze how the system and workloads are performing, which is fundamental to finding and solving performance issues.
- **Security Hardening:** eBPF allows the addition of security rules directly into the kernel. This means improving the system's security without significantly changing the kernel's code. It helps set up things like firewall rules or intrusion detection mechanisms as the authors have discussed in [JZW$^+$23], as well as being able to restrict syscall availability to programs.

- **Threat Detection:** eBPF can also monitor the system to spot unusual or potentially harmful activities. By keeping an eye on what is happening in the system, like which programs are making network connections or accessing files, eBPF helps identify security threats as they happen as the autors have presented in their paper [DSM19].

eBPF programs are executed in a sandboxed environment within the kernel, thus being isolated from the rest of the system. This sandbox approach has two main benefits:

- **Safety:** eBPF programs are kept separate from the main operations of the kernel. So, if there is a problem with an eBPF program, it will not crash or corrupt the entire system.

- **Security:** Before an eBPF program is allowed to run, it is checked by the verifier. This includes ensuring it cannot run forever (which would tie up system resources) and that it is only accessing parts of the system it is supposed to, as well as other checks that the verifier does to ensure the system impact of the eBPF program is minimal.

- **Flexibility**: Developers can create and use eBPF programs tailored to their specific needs without changing the core functionality of the kernel. New features can be added and used quickly without recompiling the Linux kernel.

For a CPU/memory/IOPS benchmarking system, eBPF's strengths lie in its minimal overhead and high precision in data collection as the authors of [WYY$^+$21] have shown in figure 9 of their research paper. One can create programs that hook into various kernel events related to CPU, memory, and I/O operations, as well as many others. These programs can then collect data in real-time, providing granular insights into system behavior and resources and allowing for mutating system internals to change their inner workings. This thesis will focus its efforts on the following areas in which eBPF can be applied:

- **CPU Benchmarking:** eBPF can monitor CPU usage patterns, process execution times, and context switches. This real-time data can help identify CPU bottlenecks and optimize thread scheduling and load balancing.
- **Memory Benchmarking:** eBPF can track memory allocation, paging, and access patterns. This is crucial for detecting memory leaks, analyzing usage trends, and optimizing memory management strategies.
- **IOPS Benchmarking:** By monitoring disk I/O operations, eBPF can help assess storage systems' performance. It can track metrics like read/write operations, I/O wait times, and throughput, which are essential for evaluating storage performance and identifying I/O bottlenecks.

Furthermore, eBPF's integration with tools like the *BPF Compiler Collection* (BCC) and *bpftrace* simplify the development and deployment of these monitoring programs. This integration allows for custom benchmarking tools tailored to specific system performance aspects, making eBPF a versatile choice for building a comprehensive CPU/memory/IOPS benchmarking system.

To load a program into the kernel, it has to undergo specific validations that check for invalid routines, such as loops and other non-supported functions. A basic execution sequence of such an eBPF program can be found in figure 2.1 containing two pariticpants, the *User Space* and the *Kernel Space*, and highlights their interactions when loading an eBPF program.

1. The user-space program compiles the eBPF program to the *bytecode* representation required by the kernel by using a compiler such as clang with the eBPF build target.
2. The compiled binary is then loaded into the kernel
3. The Kernel validates the eBPF program with the eBPF *verifier* to ensure it abides by the limits posed by the eBPF spec, such as restricting control flows like loops to prevent infinite loops and verifying the program does not exceed the maximum instruction count [Tkc24]
4. If the verification has failed, the kernel will throw an exception, which can be caught and acted upon by the user-space program
5. If the verification was successful, the kernel uses its eBPF JIT compiler to compile the bytecode representation of the program into machine-specific instructions, such as x86 or ARM(RISC), that will execute at almost natively [1] compiled speeds. [eBP23]
6. The Kernel then attaches that JIT binary to the specified kernel events, such as tracepoints
7. The eBPF program can then allocate dedicated memory sections to communicate with the user-space program. These memory sections are also called maps, arrays, or other primitive data structures commonly known in computer science.

It is important to note that the user-space program will need specific privileges such that it is even allowed to load the program into the kernel. These privileges are commonly known as ”root” privileges, but more specifically, a user-space program requires the `"CAP_BPF"` capability to load an eBPF program into the kernel and requires the `"CAP_SYS_ADMIN"` capability so that the eBPF program can influence kernel behavior and apply more invasive operations to the kernel. [eBP23]

2.2.1 Performance

eBPF delivers several performance benefits that could otherwise not be gained when using user-space profiling applications, as it enables an event-driven software architecture that can utilize OS events—such as tracepoints—to execute specific programs once such an event is invoked. eBPF programs are also unaffected by preemption, as eBPF programs are always run in one go and not subjected to OS scheduler decisions. One of the key performance benefits of eBPF lies in its ”lazy” or on-demand approach to running programs. Unlike traditional profiling tools that might continuously monitor system resources with a polling-based approach, eBPF programs are only run once their corresponding event is invoked,

[1]Note that natively compiled means that the same logic, that the eBPF program implements, is natively implemented into the kernel source-code with C and thus subjected to all the compiler optimizations that the Linux source code also undergoes

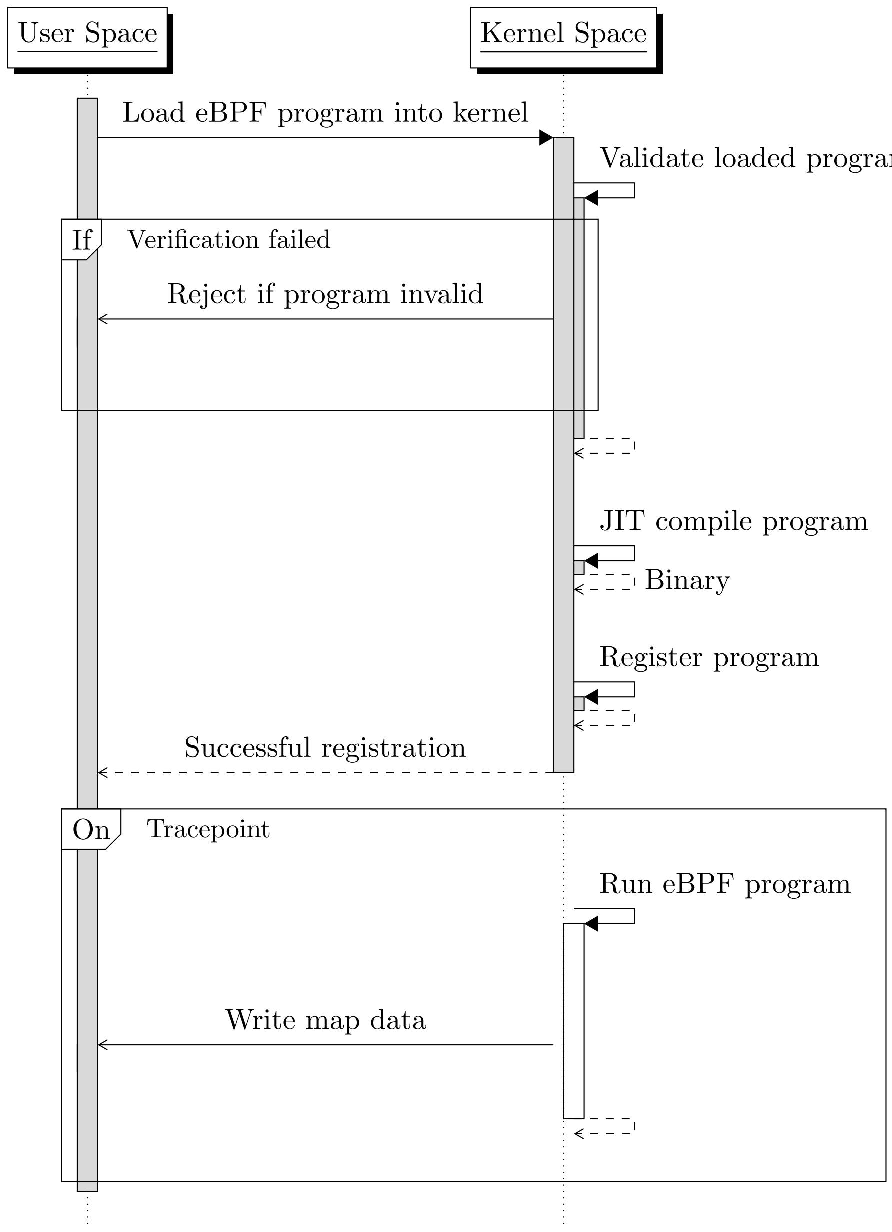


Figure 2.1: A sequence diagram outlining loading an eBPF program into the kernel and communicating with it. This and other information on the process of loading eBPF programs into the kernel can be found in the eBPF documentation [eBP23]

thus removing unnecessary executions while guaranteeing the highest invocation precision. It reduces the unnecessary consumption of CPU cycles and memory bandwidth, leading to more efficient resource utilization than user space programs, where data must be continuously passed between the kernel and user space, causing major performance overheads. The authors of paper [DSC] have shown that, especially in the area of network packet filtering, eBPF outshines its user-space counterparts by a large margin.

2.2.2 Program hierarchy

As described before, a user-space program loads eBPF programs into the kernel. This allows a single user-space program to load multiple eBPF programs into the kernel and act as a supervisor process. This is especially interesting because user-pace processing may save time and effort. This benefit in architecture will be discussed more in the assessment chapter of this thesis, highlighting the benefits of multi-eBPF single user-space program architectures over multiple user-pace program architectures.

2.2.3 Limitations

Due to the program running near the kernel, many programming basics are unavailable within the eBPF program. These limits require a more complex program architecture, often forcing the developer to implement a user-level sidecar-program next to the eBPF program—as described in the hierarchy section above—which is used for communication and more complex tasks that are otherwise not supported by the eBPF program. Here is a non-exhaustive list of the biggest limitations of eBPF, that was compiled from the verifier source-code of the Linux project found under [Tkc24]:

1. **Limited Looping Capabilities:** eBPF restricts certain types of loops, particularly infinite loops, to prevent potential deadlock scenarios in the kernel; since Linux 5.3, eBPF has gained the ability to compute bounded loops, allowing loops where the verifier can assert that they will eventually terminate.
2. **The program must be a DAG:** The eBPF program must be a *directed acyclic graph* in terms of function calls, meaning that there can not be any recursive function calls and other types of cycles that would prevent the program from terminating.
3. **No shared function calls:** It is impossible to call functions from shared libraries or other eBPF programs. Any function that needs to be called must be defined in the eBPF program itself so that the verifier can correctly analyze the program.
4. **No struct function arguments:** Passing structs as arguments is impossible; however, passing struct pointers is possible.
5. **No Direct File System Access:** Direct interactions with the file system are not permitted to prevent I/O operations from stalling kernel processes.
6. **Restricted Memory Access:** eBPF has stringent rules regarding memory access, particularly with respect to kernel memory, to maintain system security and stability.
7. **No Floating-Point Operations:** eBPF does not support floating-point arithmetic, necessitating alternative approaches for calculations that require decimal precision.

8. **Fixed Program Size:** There is a limit on the size of eBPF programs, defined by the number of instructions they can contain, to ensure quick verification and execution. As of the current Linux version 6.7, an eBPF program can only hold a maximum of 4096 instructions as can be seen in the common verifier source-code at [Tkc23]

2.3 Tracing

Tracing is observing certain programs or Operating Systems events to get a detailed overview of what happens during runtime and how these events might impact a system. In general, one can differentiate between user-space tracing, under which fall programs—written by the developer or a third party—that implement manual trace points, and kernel-space tracing, which allows privileged programs to trace events originating from the operating system. In Linux, three core tracing systems are built into the kernel itself:

2.3.1 Kprobes

Kernel Probes (Kprobes) are a debugging mechanism that allows programs to break into any routine in the kernel and collect diagnostic and debug information non-disruptively. One can place a probe at almost any location in the kernel and specify a handler routine—such as eBPF programs—to be invoked whenever the probe point is reached. Kprobes also represent a stable interface on which one can rely, although differences between kernel versions might affect or break systems relying on specific Kprobes.

All available Kprobes in a specific distribution can be found by printing the file through this path: `/proc/kallsyms`. To find a specific Kprobe, one can pipe the output through grep and thus search for a specific Kprobe in a system via this command:
`cat /proc/kallsyms | grep kmalloc`.

2.3.2 Uprobes

User Probes (Uprobes) are similar to Kprobes in that they are a native kernel feature but are instead used in user-space programs. They allow developers to set probe points in any user application during development dynamically. When running on the system, a user can then easily observe these Uprobes and, just like Kprobes, can specify a handler routine that should be invoked when the probe point is reached, such as an eBPF program.

2.3.3 Tracepoints

Tracepoints are static probe points placed in strategic locations throughout the kernel by the kernel developers. Unlike Kprobes, they are not dynamic, meaning they are predefined, and it is not possible to add or remove them at runtime, though it is possible to deactivate or activate them in the OS. However, tracepoints have a significant advantage in that they are safer and have less performance impact than Kprobes since they do not require modifying the running kernel. Attached programs to tracepoints are strictly typed and receive a struct pointer upon invocation that points to a memory location containing further information regarding this specific tracepoint. They are by far the most low-level event signals compared to Kprobes or Uprobes and allow for deep insight into the inner workings of the kernel. Combined with eBPF, tracepoints represent one of the more powerful tracing systems in

Linux. To view all available Tracepoints, one can list the directory found in Linux under `/sys/kernel/debug/tracing/events`. Each directory contains further information about the tracepoints struct format, the tracepoint ID, and more.

2.3.4 System calls

A system call is an essential mechanism in an operating system that allows user-space applications to request services or resources from the kernel, which operates at a higher privilege level. These calls are a controlled interface for executing tasks like file operations, process management, or network communication. In the context of eBPF, system calls can be used as triggering events, allowing eBPF programs to execute custom logic in response to these kernel-level operations, allowing real-time monitoring and analysis.

2.3.5 Others

Next to these core tracing systems, there also exists a myriad of other more domain-specific tracers like LTTng, Dtrace, and more statically defined tracing such as USDT, which requires recompilation of program source to attach user-defined tracepoints, which can then be measured during runtime of the program.

2.4 Profiling of processes

Measuring a process's performance can be achieved by looking at its memory utilization, disk I/O throughput, or CPU usage. All these different data points give the user a good insight into how the program behaves under load and what bottlenecks might exist. Measuring these data points is more complex than just reading out a predefined interface, as requirements and measuring methodologies can vary greatly between use cases.

2.4.1 CPU profiling

CPU profiling and benchmarking can be done in several ways, as different metrics can indicate how well a CPU performs for certain workloads. These approaches can range from measuring a CPU's time executing a workload to CPU cache efficiency to multi-threading utilization. All these metrics signify different strengths of a CPU, and it is up to the benchmarker to decide which metric is the most important.

2.4.2 Memory profiling

Memory is a limited resource in a system that stores temporary data that needs to be retrieved quickly by the processor. Measurement approaches vary from measuring the frequency of allocations and deallocations of a workload to measuring the amount of memory the workload uses. Profiling also comes in especially handy to find potential memory leaks in workloads, and thus, it is highly important to verify that a workload is safely executable.

2.4.3 Disk I/O profiling

Reading from and writing to Hard Drives, SSDs, or other storage mediums is rather expensive, as it takes a comparatively long time to do these operations. Thus, benchmarking the utilization of the disk in terms of I/O operations per second, as well as data transfer throughput and other workload behaviors, becomes essential in understanding the system's bottlenecks.

3 Related work

eBPF, a relatively recent technology[1], has been the focus of extensive research, particularly regarding its application for processing network packets through features such as the Express Data Path. Regrettably, research regarding the benefits it brings, compared to more conventional solutions, has been limited. Thus, not much scientific literature discusses how the utilization of eBPF in measuring OS resource consumption fares compared to traditional benchmarking solutions, which exist in a much more mature environment. While a lot of work has gone into building eBPF-based systems for such tasks, assessments of their benefits other than "It's just faster" have been minimal, and thus, more research has to be done in this area.

3.1 Bpftrace

Bpftrace is a command-line utility that facilitates the utilization of eBPF on a Linux machine [iov24b] by allowing for ad hoc command-line implementation of eBPF programs and therefore allows for rapid development of simple eBPF systems. With bpftrace, it is possible to attach eBPF programs to different tracepoints within the system through a single command, such as:

```
bpftrace -e 'tracepoint:raw_syscalls:sys_enter { @[comm] = count(); }'
```

Although already very mature, bpftrace is a relatively simple framework that shows its limits when it comes to more complex applications and benchmarking of large-scale deployments, as it does not cover extended capabilities such as scalability across multiple systems, easy programmatic consumption of the results, and advanced probe implementations that lie outside of the capabilities of bpftrace's proprietary[2] programming language. The user of bpftrace must build their system around bpftrace by utilizing shell scripting or other ways of automatically executing this CLI.

Utilizing bpftrace to assess how eBPF-based benchmarking systems fare to conventional solutions is not viable, as consuming the results takes time. An assessment requires control over the complete operations stack, from implementing an eBPF program in C to advertising the results in a consumable format. Bpftrace takes away the control needed to do such an in-depth assessment. Thus, while very powerful, this thesis does not utilize bpftrace to make such an assessment.

[1]While the original Berkley Packet Filter has already existed for more than a decade, the advanced functionality—which led to the renaming of this technology to *extended* Berkley Packet Filter—allowed for easier adoption and has, in its current form, only really existed since 2015 with Brendan Greggs release of the BPF Compiler Collection (BCC)[Gre15] and the release of the Express Data Path in 2016[Bla16]

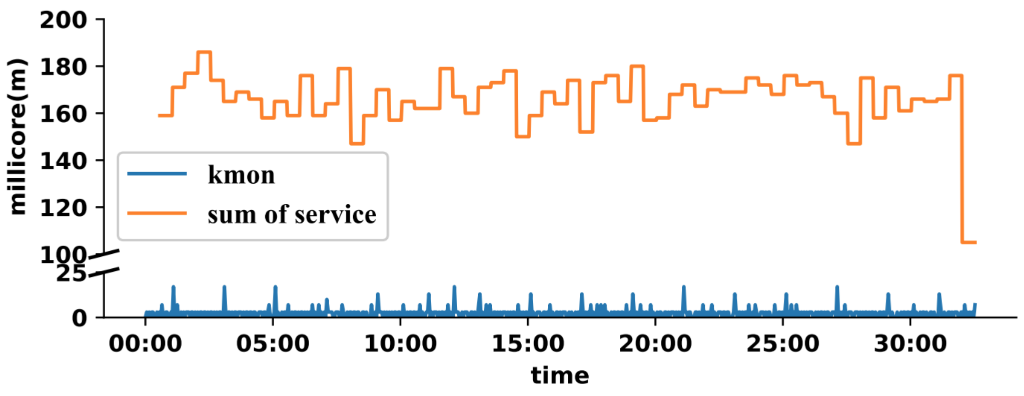
[2]Proprietary in the sense that bpftrace has its programming language to program probes in, not in the sense of corporate ownership

3.2 The Rise of eBPF for Non-intrusive Performance Monitoring

This paper [CCS], written by Cyril Cassagnes, Lucian Trestioreanu, Clement Joly, and Radu State, highlights some important points concerning how eBPF can be used to aggregate different metrics and export them in an industry-standardized format utilizing Prometheus. Their paper focuses on monitoring an inter-ledger payment system by writing an eBPF program that analyzes and monitors network packets of this respective payment system. While they have a similar architecture as will be presented in this thesis, they focus on a different use case: Network monitoring. Furthermore, the paper does not highlight the differences between their inter-ledger analysis solution and conventional analysis solutions. It focuses more on implementation specifics regarding their use-case of network observability.

3.3 Kmon: An In-kernel Transparent Monitoring System for Microservice Systems with eBPF

With their paper [WYY+21], the research team developed a microservice monitoring system based on eBPF called "Kmon," which allows for easy observability of containerized microservices running in a Kubernetes cluster. It can measure multiple metrics in performance, networking, and internal states and display its results in a flame graph, as well as export them to a set of database systems, which can be queried for further information on the collected data. Their findings show that such an eBPF-based system only has little negative influence on the OS and service response times, as can be seen in their measurements visualization in 3.3 (figure 9 on page five of their paper), which shows the CPU usage for a microservice, and kmon running alongside it, on a single host:



While they follow a similar architecture to what is proposed in this thesis, they firstly focus more on the networking aspect of monitoring microservices through eBPF, and secondly, do not assess how their solution fares to conventional monitoring solutions, which would allow for a better comparison between these two monitoring approaches. Thus, while relevant for this thesis, this thesis will focus more on such a comparative assessment.

3.4 Cloudflare ebpf_exporter

Cloudflares ebpf_exporter project [Clo23] builds an eBPF-based Prometheus exporter that advertises its measurements to be consumed by Prometheus, similarly to the architecture that will be highlighted in this thesis. Having a similar solution to our reference implementation, it needs to include a comparative analysis of eBPF over conventional benchmarking systems. To correctly assess the advantages of such an eBPF system to conventional solutions, building a reference implementation from the ground up is necessary, highlighting only the areas that are strictly necessary to do an assessment. Therefore, although it would be possible to utilize Cloudflare's solution to monitor system utilization and resources in a very similar way to how it will be implemented in this thesis, a prebuilt system such as theirs is unsuitable for an assessive analysis.

4 Methodology

This thesis aims to highlight the differences between benchmarking systems based on eBPF versus conventional benchmarking systems. To do so, a reference implementation will be created in eBPF that aggregates CPU utilization, memory utilization, and disk I/O metrics. This implementation will then be compared with other conventional tools such as **htop**, **iostat**, and the **Prometheus node-exporter**. Within this Methodology chapter, four key areas in which the assessment will take pleace will be proposed, having been deemed as the most important aspects when assessing eBPF's benefits to conventional solutions.

4.1 Correctness

Ensuring the correctness of the results is crucial, especially in complex computing environments like High Throughput Computing (HTC) and High-Performance Computing (HPC) clusters. Inaccurate benchmarking results can lead to significant financial consequences and miscalculated efforts, emphasizing the importance of reliable and precise workload benchmarking. Therefore, the focus will be on evaluating the accuracy and reliability of an eBPF-based benchmarking system compared to a traditional benchmarking approach by comparing the results to the results of industry battle-tested solutions. This assessment will examine the collected data, the precision in measuring system parameters, and the overall dependability of the results produced through this eBPF benchmarking solution. 7.1.

4.2 Modularity

Benchmarking systems often come in a large package that contains a vast array of features, of which the larger part mostly goes unused by the user. Thus, finding an adaptable system that one can selectively use without incurring memory and computing penalties that would otherwise be had is highly important. The reference implementation will be compared to conventional systems, highlighting shortcomings in their adaptability to new requirements in the assessment chapter 7.2.

4.3 Simplicity

As benchmarking is crucial for anybody running complex workloads, simplicity of usage and understandability are especially important to ensure that anybody can benchmark with basic knowledge. Therefore, when building such an eBPF benchmarking system, it must be easy to understand and to extend since any highly sufficiently complicated benchmarking system lends itself to becoming monolithic and hard to use with the years—if not maintained well—requiring deep knowledge of its inner workings and thus limiting the set of potential users. As eBPF shows much promise in several areas of system monitoring, it is important to consider the potential developers and users who might otherwise not be as versed in low-level system interactions so that they become future maintainers of these kinds of technologies. The simplicity of eBPF will be highlighted in the assessment chapter 7.3.

4.4 Scalability

Benchmarking programs, especially in large distributed systems, bring a new set of requirements that need to be fulfilled: easy scalability across several different compute-cluster nodes. Thus, highlighting the differences and advantages between an eBPF-based system versus a conventional benchmarking system allows for better judgment on which system to utilize in which use case. Especially large-scale computing lends itself to rigorous workload benchmarking and thus requires a good overview over large amounts of cluster nodes simultaneously. The scalability of eBPF systems will be highlighted in the assessment chapter 7.4.

Overall, this methodology enables an assessment on a technical level by ensuring that eBPF does indeed measure comparable results to conventional solutions, as well as more subjective aspects such as simplicity and scalability that also need to be considered when discussing the development of such systems. The human component is a crucial part that needs to be considered for any human-facing system and, thus, should not be ignored.

To realize this methodology, a reference-implementation design will be discussed in chapter 5 and the design implementation in chapter 6. Subsequently, an assessment will be done in chapter 7 comparing the reference implementation to conventional solutions in the four key areas mentioned above.

The following chapter will dive into the design of the reference-implemenation.

5 Design

This chapter outlines the design of a reference implementation of an eBPF-based benchmarking system, focusing on scalability, extensibility, and integrability. Firstly, the design objectives are outlined, specifying the criteria considered in the design decisions 5.1. Following this, the design of the probes is discussed, highlighting the challenges, possible solutions, and their final design 5.2. After describing the probe design, the user-space part of the benchmarking system is detailed in 5.3, focusing on problems that need to be solved, available tools for user-space program implementation, its architecture and operation sequence, as well as its limitations.

5.1 Objectives

To design such a system, clarity about the objectives it is supposed to fulfill will first be needed.

- **Simplicity:** As this system is compared to similar conventional benchmarking systems, it is important to match them in terms of ease of use and extensibility. While the solution highlighted in this thesis intends to represent more of a framework around eBPF benchmarking, it should still be as easily usable possible.

- **Extensibility:** Allow for easy extension of the system's core capabilities for more fine-grained utilization.

- **Parallelization:** The system should support the parallel measurement of multiple different metric types, such as measuring the CPU utilization while at the same time measuring memory consumption of the same process as well as disk I/O utilization. Allowing for a broad spectrum of parallel measurements gives the user much better insight into their program behavior. It allows for quicker analysis than if it were necessary to measure these metrics sequentially.

- **Performance:** As this system is supposed to be comparable to conventional benchmarking tools that have been diligently performance-optimized over the years, a focus lies in building a *performant* eBPF-based benchmarking suite. The benefits eBPF provides already offset several performance penalties that would be incurred with even an unoptimized, eBPF based implementation.

- **Precision:** With certain workloads, high-precision measurements are immensely important. Major memory consumption spikes might cause the OS to terminate a process. At the same time, a common benchmarking system might not even be able to measure the exact time such a large memory allocation took place. With the eBPF event-based approach, measuring precision becomes a commodity one may seek to exploit.

5.2 Probes

Beginning with the core building blocks of the eBPF benchmarking tool, specific metrics will be extracted from the system by these core elements, referred to as Probes. These Probes aim to extract only the required information, adhering to the UNIX philosophy of "Write programs that do one thing and do it well."

Three foundation probes, which may serve as a reference for further research and implementations, will be included in the system. A discussion of the technical challenges, the implementation of a solution, and the measurement of their specific metric by these base probes will be started. The CPU utilization probe, being the most important one, will be the starting point.

5.2.1 CPU Probe

Measuring the CPU utilization of a specific process is more challenging than it appears. Initially, a definition of what CPU utilization means is needed. Generally, it expresses the extent to which a process uses the CPU. However, upon a deeper examination of operating systems, it becomes clear that "A percentage value that defines the utilization of the CPU" does not exist as traditionally understood since a CPU normally always executes programs with as much processing power as it can afford [1].

As an operating system needs to execute multiple tasks simultaneously, task executions are sliced into discrete intervals between which the CPU switches, creating an illusion of parallel processing. In reality, processes are executed sequentially if running on the same core; this approach is called preemption. Therefore, a metric that provides the exact CPU utilization value spent on a specific task must be found.

Challenges

A way needs to be developed to measure the impact of CPU usage on a specific program within the system. For that, several methods could indicate the utilization, such as

- **Measuring the frequency that the CPU runs at:** Since systems are built in that can dynamically scale the CPU frequency at which the processor should run [ker23], these can be used as an indirect indicator of what the current utilization is. The problem with this technique is that it does not allow for fine-grained measurements of specific processes but only for the overall measurement of the whole CPU. It is also affected by thermal throttling behavior and other external factors and, as such, is not a reliable measurement technique.
- **Cache hits and misses:** These could indirectly indicate CPU utilization, allowing for the measurement of specific processes. As with higher cache hit/miss rates, it can be said that the CPU is doing more work, be it switching its context more often or just doing more operations that require loading memory pages. While somewhat viable, this approach does not provide an absolute value that can be used for further comparison with other benchmarking solutions.

[1] Modern CPUs are capable of dynamic frequency scaling to optimize power usage efficiency during low-demand tasks. However, for the sake of simplicity, it is assumed that a CPU will always run at a fixed frequency

- **Context switching rates:** Rates at which the CPU switches its context could indicate that the scheduler is more switch-happy and thus has more processes to switch between. This would allow for the measurement of the frequency at which the scheduler switches to the process currently being measured, thus providing good insight into the demand of that specific process on the system relative to other processes. However, this does not provide the total utilization, as it only represents a relative value to other processes. Thus, it cannot be used as a reliable method for measuring the overall effort expended by the CPU on that specific process. This technique also depends on the scheduling strategy of the OS and thus could be different between different system configurations, such as real-time Linux OSs vs. traditional Linux OSs, and makes this metric difficult to compare.
- **Overall time spent on a specific process:** Measuring the total start and end times would allow for fine-grained measurements in specific processes and provide a rough estimate of the CPU time utilized. However, this technique breaks down when considering that measuring the overall time spent on a process means a continuously running program, such as a web server, would outweigh any short-lived processes. This is despite the possibility that these short-lived processes might consume significantly more CPU time. Conversely, a web server with a low request volume, often running in an idle state, would have a negligible impact on overall CPU utilization and yet show up as the most demanding task in the system. Thus, simply measuring the overall time spent by the OS on that process does not provide a good insight into CPU utilization.
- **Measuring the ALU [2] operations:** This could give A good insight into especially maths-heavy computations. If the exact workloads to be measured are known and their biggest bottleneck lies in the speed of mathematical computations, measurements could be focused on the ALU, and it could be measured which process takes up most ALU operations. Although this approach is a viable way of measuring very maths-heavy workloads, it breaks down in environments where the workload is unknown.

Limitations that make these solutions impractical for precisely measuring the CPU utilization of a specific process are present in all of them.

Solution

The time spent on a process is measured by the measurement technique utilized by the CPU probe. However, the time the CPU spends running that process is the only time measured, thus excluding any timeframes in which the scheduler switched to a different process instead of measuring the overall time spent from the program's start to the end.

An eBPF program will be attached to the kernel schedule switch event, invoked whenever the scheduler switches from one process to another. This allows the measurement of the time the scheduler switches from one process to another and enables the addition of a total per-process counter that holds the amount of time the CPU spent on each specific process.

Furthermore, a map that holds all process PIDs and their processing time is held within the eBPF program, and every time the scheduler switches away from that PID, it will be added by the eBPF probe. This map can be read out by a parent user-space program, as described later.

[2] Arithmetic logic unit

A flowchart that outlines the basic working principle of the CPU probe can be found in figure 5.1.

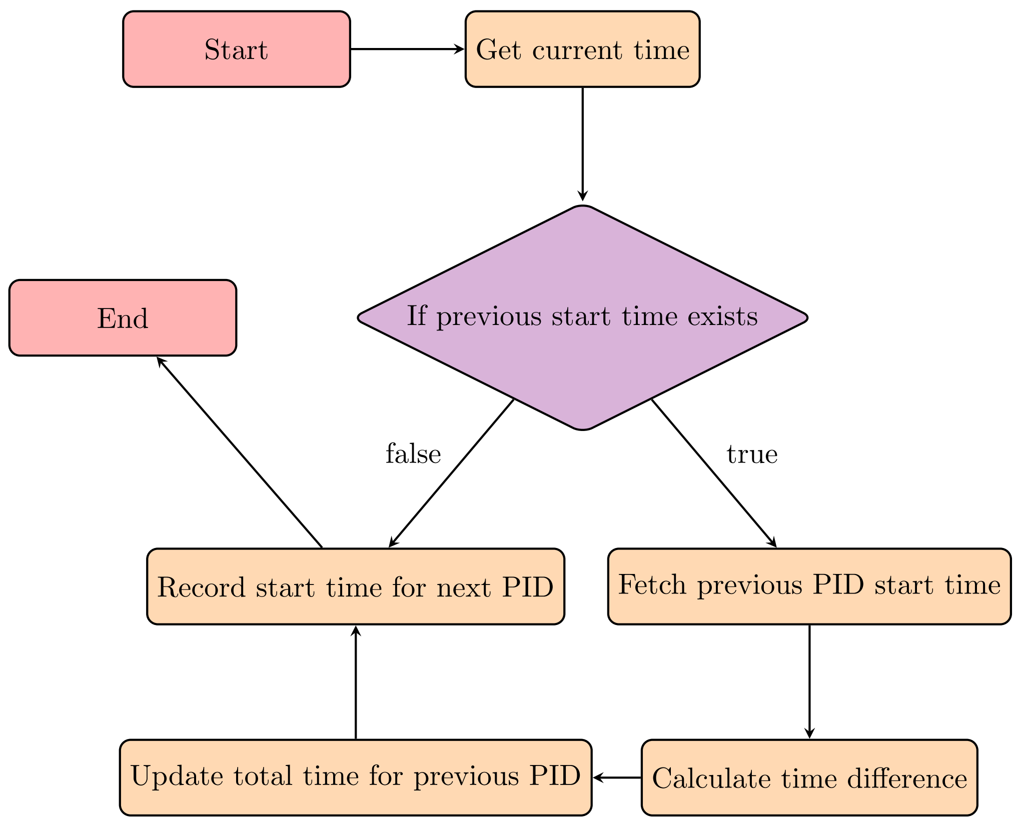

Start
Get current time
If previous start time exists
End
false
true
Record start time for next PID
Fetch previous PID start time
Update total time for previous PID
Calculate time difference

Figure 5.1: A flowchart diagram, showing the order of operations of the eBPF CPU probe

The flowchart describes the steps of operation done by the eBPF program to easily store all the CPU timings of the different processes in the hashmap, which the user-space program can then read out. The order of operations is as follows:

1. The current timestamp of that time with the highest possible precision, nanoseconds in this case, is first fetched once the probe is invoked.
2. If the hashmap does not hold a start time for the process from which the switch is being made, the current execution for that process is ignored, and a direct jump to step 5 is made.
3. Otherwise, the start time of the previous process, stored by a past eBPF program invocation in step 5, is looked up, and the time difference from the timestamp fetched in step 1 is calculated.
4. The total time spent on the previous PID is then updated by adding the calculated difference to the already existing time-spend-on-process value, which is continuously incremented until it is decided to reset it by the user-space program.
5. The start time for the next process is then recorded, which will later be used in step 3.

Following this probe design, the time the CPU spent on a specific PID can be measured. Thus, by comparing it to the overall measurement interval of the eBPF program, a relative percentage value and, consequently, the CPU utilization can be calculated.

One important thing to note is that this approach still needs to account for multi-processor systems. Running this probe on a multi-processor system against a program that does some multi-threading might lead to utilization values above 100%. This is due to the program utilizing two processor cores at the same time.

So, suppose the PID timings are summed up, and the measurement interval is not adjusted by multiplying it by the number of available cores. In that case, programs might be seen using up to $n * 100\%$ utilization, where n is the number of processor cores the system has.

For simplicity's sake and to avoid further confusion, the decision has been made not to normalize the CPU utilization by the number of available cores, even though it is acknowledged that a utilization percentage can exceed 100%.

In summary, with this procedure, the CPU probe is expected to correctly measure the CPU utilization of a specific process. However, it is important to note that this approach is still quite primitive. Time spent on the CPU does not account for specific syscalls within a process that could be slow but do not have much computational impact on the CPU. Other operations, such as I/O operations, could cause the scheduler to remain in a process even though the I/O operation might be bottlenecked by other limiting factors such as memory-, network-, or disk speeds. To measure CPU utilization more granularly, accounting for the exact syscalls to compensate for other operations, like mathematical operations computed by the ALU that also pose a limiting factor to CPU utilization, is necessary.

5.2.2 Memory probe

The memory usage behavior of a program is measured to evaluate if the program is especially demanding on memory operations, leading developers to find better memory management solutions to keep their memory usage in check.

Measuring memory usage is more complex than it sounds. It is necessary first to decide how the memory utilization of a process is to be measured and what memory utilization even means.

Challenges

The measurement of memory utilization of a program in Linux must be approached differently than the measurement of CPU utilization. Unlike CPU utilization measurement, which can be started at any time without needing any knowledge before registering the eBPF program and can be easily measured through tracepoints and eBPF programs, measuring memory utilization poses some hurdles, especially regarding the intricacies of OS-level memory management.

Several methods exist for measuring the memory impact a program has; a few examples will be seen in the following:

- **Measuring actual memory usage:** If insight into how much memory is dedicated to a program is desired, this becomes especially interesting. It can aid in scheduling optimization tasks, providing information about how many programs could run on a single machine. This represents one of the more approachable forms of memory utilization measurement and allows for a good basic insight into the program's behavior.

It is important to note that several operating systems have features such as memory swapping, where parts of the memory are offloaded to the disk as it runs out of space. Thus, measuring the RAM and swap usage is important for a clear overview of memory utilization.

- **Measuring the number of allocations and deallocations:** Allocating memory is a costly procedure on the Operating System side, as the operating system needs to keep track of which memory segments are dedicated to which processes. As such, allocations alone can already significantly negatively impact system performance. Some programs could run on low memory space requirements while overwhelming the operating system with large amounts of allocation calls. Thus, the OS speed could be negatively impacted while the system still has much free memory that could be used. Of course, not only the number of allocations and deallocations but also the size of their memory allocation and deallocation will need to be taken into account, as especially small allocations can lead to fragmented memory, thus not only negatively impacting the OS memory management performance but also memory storage efficiency.
- **Memory Access Patterns:** Analyzing how a program accesses memory, such as read/write operations and access to different memory regions (stack vs heap), can provide insights into its efficiency and potential optimization areas.
- **Page Fault Frequency:** Monitoring the rate of page faults (minor and major) can indicate how effectively a program uses memory and how often it requires additional memory resources from the system.
- **Cache Usage:** Investigating cache hits and misses, especially in systems with multiple levels of caches, can help understand the efficiency of memory usage. eBPF tools can be used to track cache access patterns.
- **Memory Bandwidth Utilization:** Measuring the bandwidth a program uses for memory operations. High memory bandwidth usage can indicate intensive memory operations, which might be a bottleneck in system performance.

When measuring memory utilization, there are two main events that one should account for:

- **Memory allocations:** Any time a program requires increased memory, it will try to allocate that amount of memory in the heap of the program memory. This will naturally increase the amount of memory utilization of the program.
- **Memory deallocations:** Freeing up memory to be used by the OS for other parts of the system will lead to decreased memory utilization of the program to be measured

It is important to note that a program's memory consists of the stack and the heap. The stack size, comprised of variables, functions, and general program instructions, stays static and is already mainly known once the program is loaded into memory. In contrast, the heap can grow and shrink dynamically during program execution. Therefore, a decision needs to be made about whether to include the stack memory consumption of the program or to measure only the dynamic memory part, where good insight into the memory behavior is much more crucial.

Additionally, measuring allocations and deallocations alone only allows the memory behavior to be measured from when the eBPF program was loaded into the kernel. This means that if the goal is to measure the memory utilization of a specific process already running at the time the eBPF program is loaded into the kernel, the correct amount of total memory utilization of that program will not be captured, as it would most likely have already allocated memory before the measurements began.

To work around this issue, it is necessary to initially query the OS to determine how much memory is allocated to a specific process. This value represents the starting point, to which the size of all allocation and deallocation operations is then added or subtracted to calculate the final amount of memory utilization.

Since eBPF cannot directly perform I/O or any other operations that cause long waiting times, the initial program memory consumption from the OS cannot be fetched within the eBPF program. Therefore, a user-space sidecar program is required to act as an initializer, fetching these initial values and preparing a map containing these prefetched values for the eBPF program to operate on.

To measure memory allocations and deallocations, Linux provides several tracepoints and Kprobes. Due to the different ways a user-space program can request the OS to allocate memory, like using the "brk" and "mmap" syscalls, which differ in their use-case scenarios, it is necessary to find a simple method of measuring allocations and deallocations without covering all possible system events. Linux offers a useful Kprobe event called kmalloc, which is called every time a program allocates memory, allowing the attachment of an eBPF program to this event.

However, listening for deallocations becomes more complicated. The corresponding 'kfree' probe does not provide the memory size freed up. Additionally, using helper functions that the kernel would provide, such as 'ksize', is not feasible. These functions represent an unstable interface to eBPF and would access restricted memory areas of the OS; thus, their utilization is impossible.

Solution

As already discussed, monitoring the memory utilization with eBPF is more complex than monitoring CPU utilization. This complexity arises from the numerous ways memory utilization can be measured and eBPF's reliance on stable OS APIs, which imposes certain restrictions.

A probe design that can measure a PID's overall memory consumption and behavior is needed, enabling later comparison with other conventional benchmarking tools. A probe design is opted for to measure memory utilization reliably, where the initial user-space program first aggregates the current memory utilization of every PID in the system. These aggregations represent the base for the runtime memory allocation analysis on which allocations are added and deallocations are subtracted, thus resulting in the actual memory utilization of each specific program.

In figure 5.2, the core sequence of operations for the memory probe initialization is outlined:

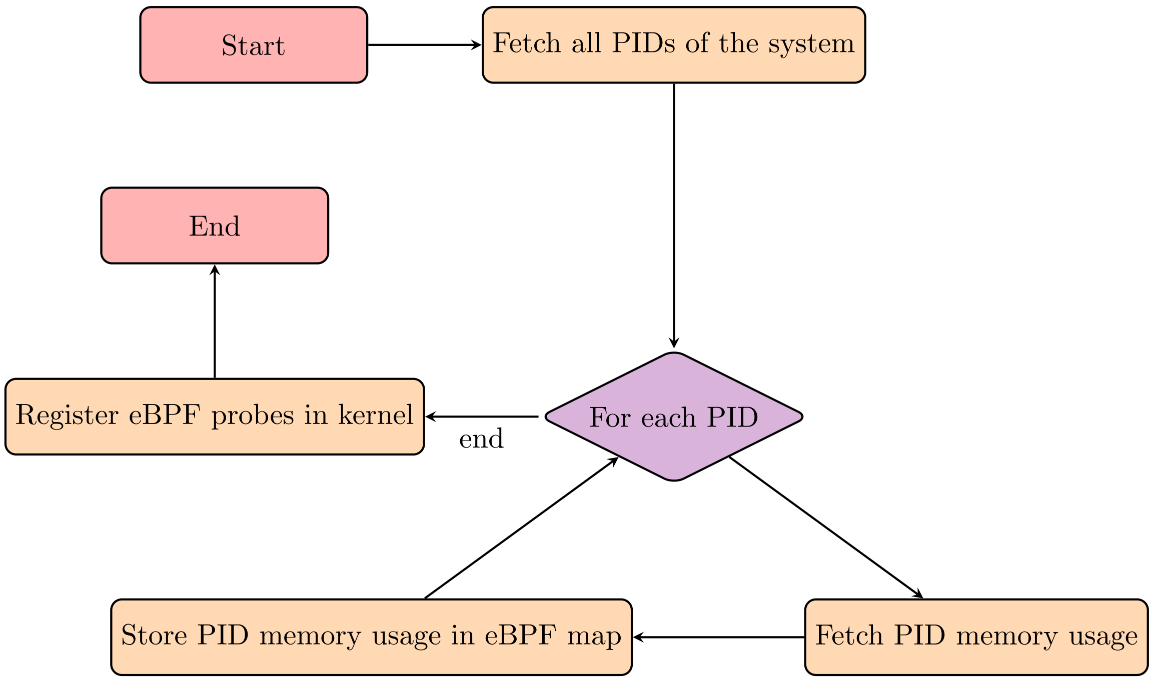

Start
Fetch all PIDs of the system
End
Register eBPF probes in kernel
end
For each PID
Store PID memory usage in eBPF map
Fetch PID memory usage

Figure 5.2: A flowchart diagram, showing the order of operations of the eBPF memory probe on the user-space program

The flowchart outlines the steps that need to be taken by the eBPF program to correctly prefetch the current memory consumption of each process in the host OS. The sequence of operations is as follows:

1. The user-space program will first list all available PIDs in the system.
2. For each PID, the OS's `/proc/PID/status` will be queried, returning the different types of memory used by this process. Since the total memory consumption is of interest, the total memory usage under the "VmSize" field within that file will be taken. The VmSize field returns the virtual memory size reserved by that process.
3. After extracting all PID memory usage, the memory usage eBPF probes will be registered. These probes consist of two functions that are called independently by different events. A flowchart diagram of these can be viewed in figures 5.3 and 5.4.

Following that, a sequence of operations when a memory allocation takes place can be found in figure 5.3, highlighting the following steps:

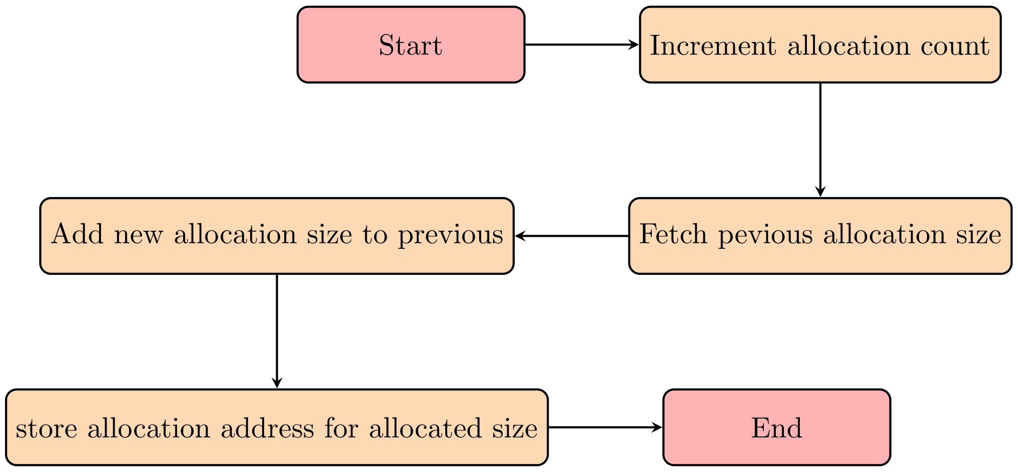

Start
Increment allocation count
Fetch pevious allocation size
Add new allocation size to previous
store allocation address for allocated size
End

Figure 5.3: A flowchart diagram, showing the order of operations of the eBPF memory allocation probe

The flowchart describes the sequential operations needed to gather all information correctly when a process allocates new memory. The following steps can be observed:

1. The eBPF probe is invoked through an allocation event from the OS.
2. The allocation counter is incremented by one.
3. The previous allocation size, prepared by the user-space initialization or a previous run of the eBPF probe, is fetched.
4. The newly allocated size is added to the previous allocation size and stored in the allocation size hashmap.
5. A mapping for the memory address to allocation size is also stored, which will be required by the deallocation probe below.

The order of operations for the deallocation probe, as depicted in figure 5.4, is as follows:

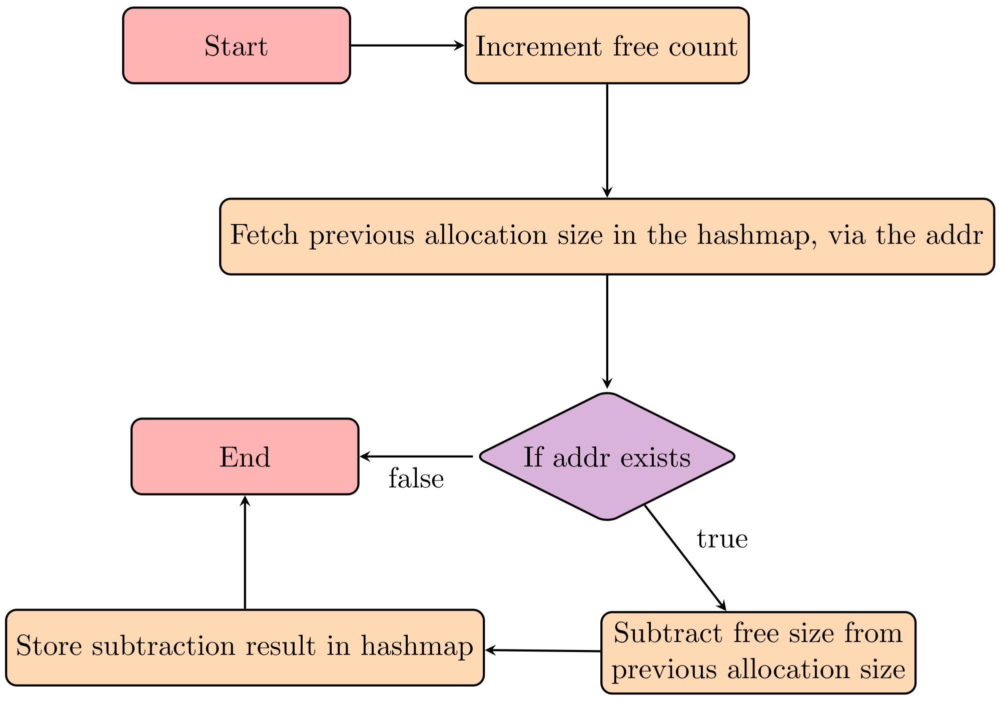

Start
Increment free count
Fetch previous allocation size in the hashmap, via the addr
If addr exists
false
End
true
Subtract free size from previous allocation size
Store subtraction result in hashmap

Figure 5.4: A flowchart diagram, showing the order of operations of the eBPF memory free/deallocation probe

The flowchart describes the operations that need to be run in order to correctly account for deallocations/frees of a process. The following steps may be observed:

1. The eBPF probe is invoked through a deallocation event from the OS.
2. The deallocation counter is incremented by one.
3. The memory size is fetched through the address provided by the OS.
4. If that address exists in the map, the size is subtracted from the total memory allocation size of the process.
5. If it does not exist, the eBPF function call is exited directly, as the amount of memory freed cannot be known if a mapping does not exist for that address.

5.2.3 Disk utilization probe

Measuring disk utilization brings its intricacies with it. There are several different types of harddisks on the market that all have very different behavioral patterns. They can majorly differ in access speeds, readout speeds, write speeds, and other areas that might not be as apparent, such as resistance against vibration and thermal limitations.

Challenges

There are several ways of measuring the disk utilization of the system, such as:

- **I/O Operations Per Second (IOPS):** This metric measures the number of read and write operations on the disk per second. High IOPS indicates heavy disk usage, which can impact system performance. It is possible to trigger the eBPF program on system calls related to disk I/O, such as read(), write(), pread(), and pwrite().
- **Throughput:** Measuring the amount of data read from or written to the disk per second. This metric is crucial for understanding the data processing capability of the disk. Triggers would be similar to IOPS, focusing on the volume of data transferred in I/O operations.
- **Latency:** The time taken for an I/O operation to complete. It is a critical metric for performance-sensitive applications. Triggers could be the Start and completion of each I/O operation, allowing the eBPF program to measure the time elapsed for these operations.
- **Disk Queue Length:** The number of I/O requests waiting to be processed. A long queue can indicate a bottleneck. Triggering can be done on the enqueue and dequeue events in the disk's request queue.
- **Error Rates:** The frequency of read/write errors. High error rates can indicate hardware issues or filesystem corruption. Triggers would be system calls that return errors during disk operations.
- **Disk Utilization Percentage:** The proportion of time the disk is actively processing I/O requests. It helps in understanding how busy the disk is. Triggering on all disk-related activities and time measurements can provide this data.

Facing several fundamentally different types of hardware with a common API for user-space programs, such as SSDs, HDDs, NVME technology, or network block storage, requires measuring their utilization independently of the specific technology used. This approach allows for creating a comparable set of metrics that can be reliably compared.

Solution

To effectively measure disk utilization, the choice will be made to measure IOPS [3] and disk throughput. This approach allows for easy insight into the disk's behavior. While measurements like Latency, disk queue length, or even error rates can be interesting in niche

[3] I/O operations Per Second

use cases, they do not provide good insights into the general process of disk utilization behavior.

Thus, the eBPF probe will measure the following metrics:

- **IOPS:** Measurement of IOPS is rather straightforward. The eBPF program will increment a counter any time there is disk access. Disk accesses differ by what is done on the disk, in either writing or reading data. In general, read operations are faster than write operations, and thus, the maximum amount of read IOPS a disk can handle will normally be higher than the maximum write IOPS. Note that this is not always the case, especially when it comes to newer technologies such as network block storage devices, which are often used in large-scale computing clusters and are more often than not limited by their inter-connectivity speed and computing speed of the storage nodes, as well as replication and mirroring procedures or soft limits set by system administrators to limit the throughput of the storage system, as to not negatively impact other systems that are also utilizing a network block storage device. (Noisy Neighbors)

- **Throughput:** Disks can store vast data. The throughput defines the size of the data being stored or retrieved from the disk. Every single throughput write/read request will be recorded, thus providing a precise insight into the disk behavior.

These two metrics are often negatively correlated, where if a process has a very high amount of IOPS, the effective throughput to and from that disk is reduced, as large amounts of IOPS can cause a slowdown in read and write operations. Especially regarding HDDs, read/write head movements and logical disk sector lookups are very expensive regarding the time taken to do the operation. With SSDs, this has become less of an issue since no moving parts within the disk are physically limited in their speeds. [Pur23]

IOPS and Throughput will be split into their respective read and write values. This split results in four distinct metrics that will be stored in the eBPF map and can be read out by the user-space program.

The design for the disk I/O probe will result in a sequence of operations as depicted in figure 5.5.

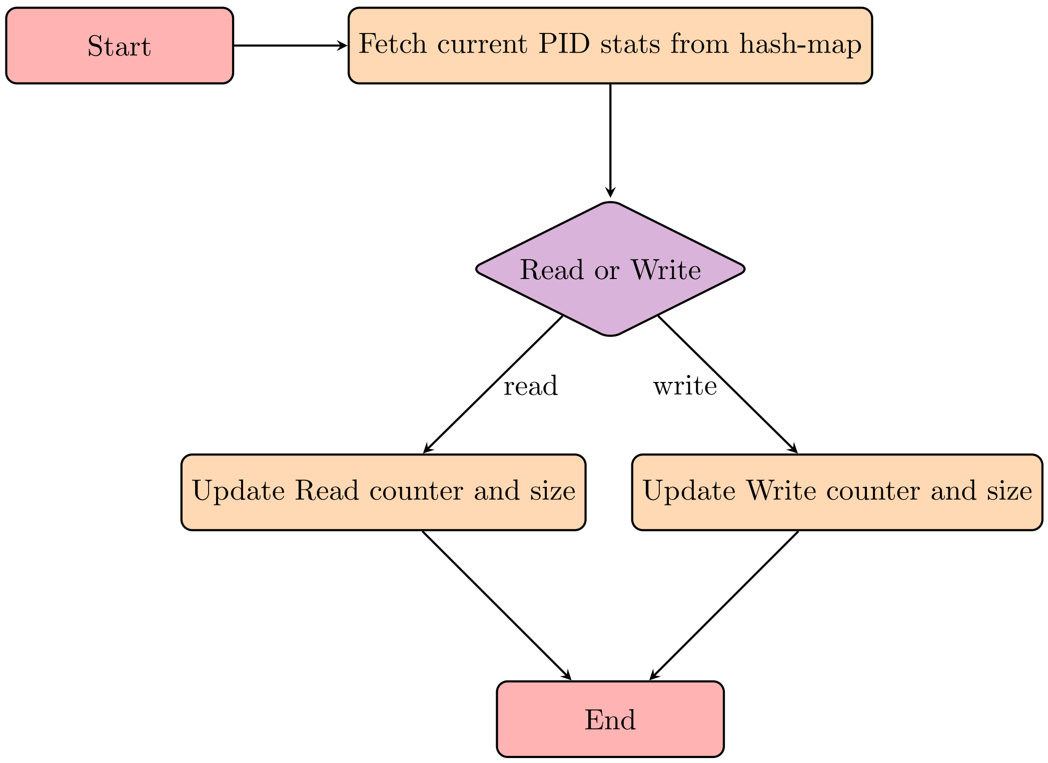

Start
Fetch current PID stats from hash-map
Read or Write
read
write
Update Read counter and size
Update Write counter and size
End

Figure 5.5: A flowchart diagram, showing the order of operations of the eBPF disk I/O probe

The flowchart diagram represents the operation sequence that must be executed to aggregate disk data correctly writes and reads, including the size of that read or write operation. The following steps can be observed to achieve such an aggregation:

1. The probe is invoked through the OS event configured.
2. The probe checks if the current operation is a read or a write operation.
3. If it is a read operation, the read IOPS counter will incremented by one, and the read byte size will be added to the total read size in the hash map.
4. If it is a write operation, the write IOPS counter will incremented by one, and the written byte size will be added to the total write size in the hash map.

5.3 Main user-space program

Having developed a sound design for the core building blocks of the eBPF benchmarking system, the focus can shift up one level. While specific metrics covering a vast area in the system utilization spectrum have been considered, it is recognized that these building blocks can achieve only so much by themselves. For the execution of these probes, a main user-space program is required. This program will manage the compilation pipeline of the eBPF probes, load them into the kernel, and handle the communication, consumption, and exposure of the data from these probes for further analysis.

5.3.1 Problems to solve

The user-space program will need to solve some fundamental problems that can not be solved or are very cumbersome to solve by the eBPF programs themselves.

1. **Probe data aggregation:** As discussed in the probe design section, some probes measure interval-dependent metrics. Furthermore, while eBPF probes are normally always invoked on an event—meaning they are even-driven—the actual calculation to aggregate the data gathered by a probe might still be time-dependent. While CPU utilization can be measured any n seconds, where the average utilization that a process demands from a CPU will be calculated, other metrics such as the disk IOPS are clearly defined as I/O Operations per Second and thus must be measured in 1-second intervals. This problem should be solved generically, as many time-dependent metrics can be measured within the system, either by existing probes, such as the disk I/O probe, or probes that might be implemented later, such as network throughput per second. Therefore, the architecture of the user-space program requires a precise interval aggregation capability. This will allow for the free definition of which probe is evaluated and how often within a specific time interval.

2. **Data Advertising:** Merely aggregating data within the user-space program has limited use cases. Ideally, this data should be advertised so that it can be consumed by other systems, which may then build decision-making processes or observability on top of it. The advertising of this aggregated data should follow a standardized protocol, enabling the program to integrate into existing infrastructure with minimal breaking changes. As the data is advertised, relying on an event-driven approach to stream data to an external source is not feasible. Implementing such an approach would necessitate specific technologies, like Remote Procedure Calls or Websocket Streams, which are beyond the scope of this thesis. Thus, a consuming system will query the advertised data at specific intervals. It is important to note that the consume interval is independent of the probe data aggregation interval. For example, while CPU utilization is measured every second, a consumer can only fetch the current utilization every minute. Users can adjust the final configuration of data aggregation intervals versus consumption intervals based on their specific needs or preferences.

3. **Extensibility:** Adding new probes to the user-space program should be non-invasive. There should be no need to alter the existing program when adding a new probe.

5.3.2 Available tools

To compile the eBPF programs to the bytecode representation discussed in the Background section, a compiler suite is needed that aids with compilation and subsequently loading the bytecode into the kernel. The BCC toolkit by IOVisor offers a vast set of functionalities and integration frontends for Python and Lua [Con24b].

BCC offers a straightforward front end for Python so that the user-space program implementation will be written in Python. This choice was made because Python is a simple programming language, allowing a wider audience to understand its concepts more easily. Although it is possible to also build user-space programs in other languages such as Rust with the AYA toolchain [Con24a] as well as Golang, where Cilium has created an eBPF compilation suite [eC24].

5.3.3 Architecture

A general architecture must be outlined to build the user-space program, including how it functions and bootstraps the eBPF probes. Figure 5.6 represents such an architecture:

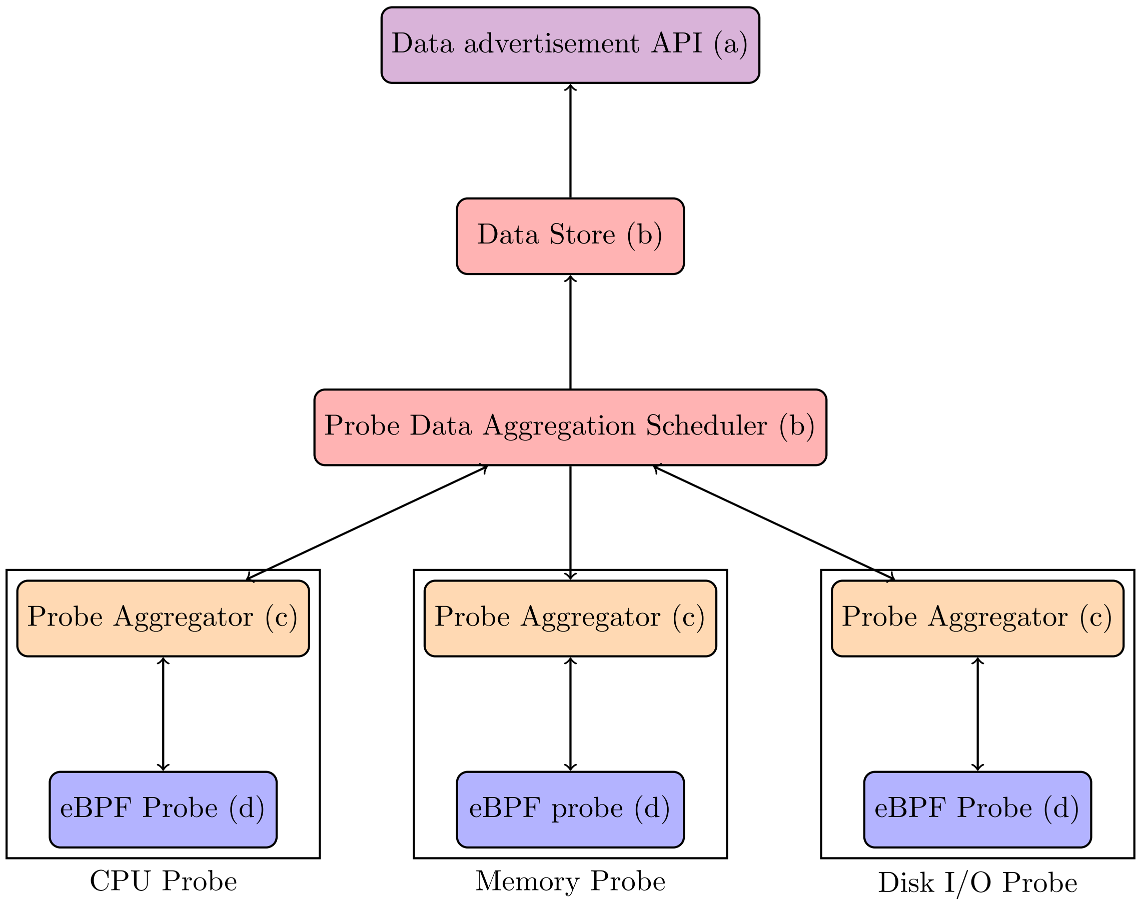


Figure 5.6: A color-coded architectural overview of the whole system. Colors are accompanied by letters for visually impaired readers

This flowchart diagram, also representing an architectural diagram, highlights the core working principles of the reference implementation. The colors of each node represent their area of concern:

- **Violet (a):** The violet node at the top symbolizes the part of the user-space program responsible for communication with external systems. Though theoretically optional, choosing not to publish metric aggregation results is senseless.
- **Red (b):** These sections represent the internal systems of the user-space program. As previously discussed, they are written in Python and have a specific implementation.
- **Orange (c):** The orange segments indicate implementations specific to the corresponding eBPF probe. These implementations are crafted in Python, the same language as the main part of the user-space program.
- **Blue (d):** The blue sections denote the eBPF probe implementations necessary for measuring specific metrics. These eBPF implementations and their user-space counterparts are closely linked and should always be considered cohesive.

Additionally, a closer description for each section of figure 5.6 can be found in the following list of items:

- **Data advertisement API:** As mentioned, this component is responsible for external communication. A webserver-based approach allows almost any consumer to extract necessary data. The chosen format is the Prometheus data export format, offering a standardized way of exporting data. This choice also potentially extends the functionality of the user-space program for a future push-based data publication approach using Prometheus' PushGateway. [Aut24]
- **Datastore:** This area stores all measurement results and is the source from which the advertisement API fetches data. As data is continuously fetched from the probes, there might be a need to retain it for extended periods or, conversely, to discard old entries. Targeting Prometheus as a consumer, which performs its data retention, means the data store only needs to retain data for a short period. However, retaining data for at least as long as the consumer's polling interval is crucial to ensure data is not deleted before consumption.
- **Probe Data Aggregation Scheduler:** This system triggers data aggregation and loads the required probes into the user-space program. It enables selective loading of probes, which is beneficial when managing many probes that could otherwise impact the system if all were loaded unnecessarily.
- **Probe Aggregator & eBPF Probe:** This component represents the specific metric or metrics to be measured by a probe. Each probe can measure one or multiple metrics within a similar domain.

This architecture covers all objectives as mentioned at the beginning of the Design chapter, that is:

- **Simplicity:** Although subjective, the architecture's simplicity is apparent in its ease of understanding due to the limited number of features that need to be covered.

- **Extensibility:** This aspect is facilitated by the ability to dynamically load numerous probes into the user-space program without modifying the program's core functionality. Probes function as modules or plugins to the main system.
- **Parallelization:** Enabled by the Aggregation Scheduler and the inherent nature of eBPF, this feature allows for responding to system events as they occur and aggregating probe data in various ways, independent of other concurrently running probes.
- **Performance:** The core working principle of eBPF ensures performance. The ability to freely configure the intervals at which probe data is aggregated can lead to significant performance optimization without sacrificing precision.
- **Precision:** The event-driven nature of eBPF guarantees precision, allowing for the measurement of **every** traceable event without resorting to sampling methods. The probe aggregator and Aggregation Scheduler determine the precision with which metrics are measured, averaged, and analyzed.

5.3.4 Operation sequence

To further highlight how this user-space program will operate once it is installed on a system and is being bootstrapped, the reader can refer to figure 5.7. This figure shows a sequence diagram containing four participants:

- **System:** The host OS and other processes running within that host OS that are not part of the eBPF benchmarking system.
- **Aggregator:** The layer of the benchmarking system that schedules the probes and ensures measurement data retention.
- **Data store:** The part of the benchmarking system that holds the data gathered by the probes.
- **Probe:** Entities in the benchmarking system that are dedicated to retrieving a certain metric. This participant includes the eBPF probe, as well as its user-space counterpart

5.3.5 Limitiations

Even though this system does cover a good set of basic requirements, there are still some technical limitations that could be iterated upon, but it is chosen not to do so, as it is not the primary scope of this thesis. One such limitation would be the reliance on a JIT-compiled user-space program written in Python, which requires the Python runtime to be installed on the system running the benchmarking suite as well as libraries/packages that need to be installed. Ideally, the user-space program would be written in a compiled language such as Golang or even Rust, which would allow for higher performance, lower system impact, and less dependency requirements of the program. Compiling with statical linking would even allow shipping a single binary without **any** external dependencies such as *glibc*.

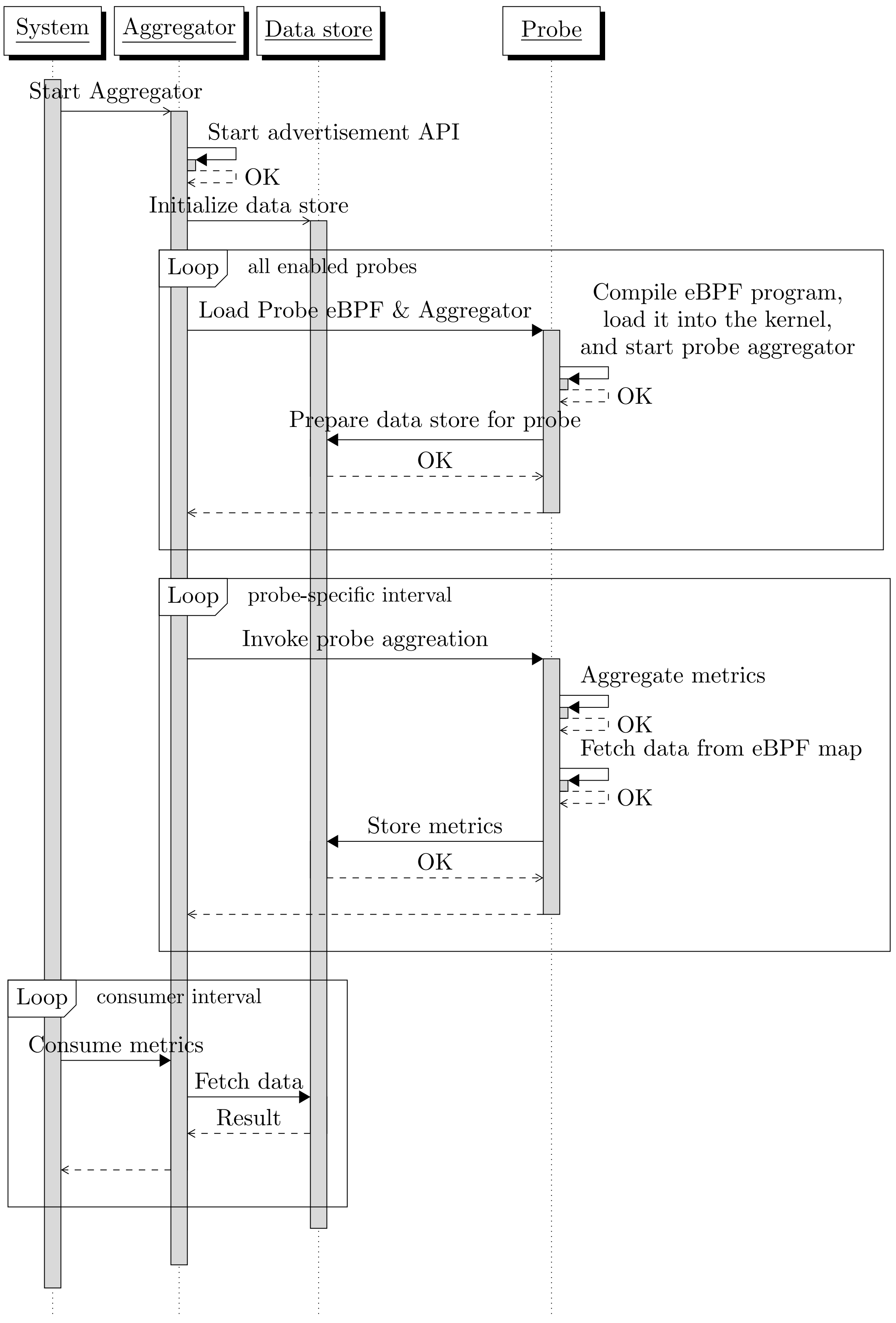


Figure 5.7: A sequence of operations of the aggregator user-space program, as described in the design section 5.3

6 Implementation

As the system's architecture has been outlined, the focus shifts to its implementation. This chapter aims to highlight particularly interesting details of the system's implementation; it does not provide a step-by-step walkthrough of the code and, therefore, is not intended as an implementation guide.

6.1 Toolchain

To work efficiently with eBPF, a simple pipeline is required to load an eBPF program into the kernel and initiate communication. This involves creating a Python script that loads the eBPF program—written in C—from a file. This program is then passed to the kernel for verification, compilation, and attachment to the defined tracepoints.

Additionally, it's crucial to ensure that the operating system meets all the requirements for loading eBPF code into the kernel. This includes verifying that the host kernel was compiled with the appropriate flags and features. The bcc-tools library and the necessary Linux headers used by the eBPF program during compilation must also be installed. Detailed installation instructions are available in the IOVisor installation readme. [iov24a]

6.2 Probes

This section delves into the implementation specifics of the reference eBPF benchmarking system. It does not contain a full implementation but only references points of interest, such as specific implementation details and potentially tough-to-retrieve information.

6.2.1 CPU utilization probe

EBPF Part

As previously discussed, the CPU probe extracts each process's start and end times upon the OS schedule switch event.

All of these timings are added by the eBPF program, providing a proportional overview of the time spent on a specific process compared to the time spent on all other processes and the idle state.

The tracepoint named `sched_switch` has been chosen for attaching the eBPF program to the kernel scheduling switch event. The available tracepoints enabled for the current OS must first be listed to locate this tracepoint. In Linux, this can be achieved by viewing the `/sys/kernel/debug/tracing` directory containing a directory of potential tracepoints. Each directory within the tracing directory is tasked with a different part of the system. Since interest lies in scheduling events, the `sched` directory is inspected to find all scheduling tracepoints available. A file named `format` is also found within this directory, containing information on all available values that can be utilized in the eBPF program specific to that tracepoint. Additionally, ensuring that the tracepoint is enabled in the OS is crucial, which can be done by introspecting the `enabled` file.

With the identification of the desired event to attach to, the development of the eBPF program, written in C, can proceed. The choice to write the eBPF program in C is due to the more mature landscape of eBPF tools for C compared to other languages. Theoretically, eBPF programs can be written in any language sufficiently suited to and supported by LLVM[1], provided LLVM can target the eBPF ELF file format for compilation in that language. However, further development is required to allow the development of eBPF programs in languages other than C. This necessity arises because support for Linux header files must also be extended to that language.

The required Linux headers for the eBPF program have been installed, providing all necessary components for developing the eBPF probe program. The development will begin with the initialization of a C file and the import of these headers:

```
#include <uapi/linux/ptrace.h>
#include <linux/sched.h>
```

In developing eBPF programs, a function must be defined called when the observed event occurs. This function receives a set of arguments specifically defined by the type of event it addresses. In the scenario where the function is attached to the `sched_switch` tracepoint, it takes a pointer to a struct as its argument. This struct's definition aligns with the information the `sched_switch` event provides. The struct, imported from the installed kernel headers, is named `tracepoint__sched__sched_switch`. It contains important information about the tracepoint which may be of interest for further implementation:

```
int onswitch(struct tracepoint__sched__sched_switch *args)
{
    //function implementation
}
```

Note: An arbitrary name can be assigned to the function in eBPF programs. In this instance, the chosen name is *onswitch*. Although the name is arbitrary, it is still necessary to specify this function name later to inform eBPF about the function which it should attach to the event of interest.

With a solid basic structure already established for the eBPF program, the implementation of the specific functionality can continue. In this case, as the focus is solely on aggregating CPU times, the task involves querying the Process ID (PID) that the scheduler is transitioning from and the PID it is switching to. This information can be extracted from the previously mentioned struct in the following manner:

```
int onswitch(struct tracepoint__sched__sched_switch *args)
{
    u32 prev_pid = args->prev_pid;
    u32 next_pid = args->next_pid;
}
```

[1]LLVM is not an acronym. LLVM is a compiler infrastructure consisting of toolchains that allow for rapid development of new programming languages.[LLV24]

The timings are measured by retrieving the current time in nanoseconds and calculating the time difference since the scheduler last switched to the `prev_pid` PID. This difference is added to the hashmap, providing the required timing data.

To communicate these timings to the external environment, eBPF maps are utilized. These maps can be viewed as a shared memory location between the eBPF program and its user-space counterpart, allowing data transfer.

The definition of these maps in the eBPF program is possible through the use of predefined C macros specifically designed for eBPF:

```
BPF_HASH(cpu_time, u32, u64);
```

The first argument here is the variable name of that map that can be referenced and interacted with by the program. In contrast, the second and third arguments represent the key and value types of the map, respectively.

User-space Part

Having developed the eBPF program, it must be attached to the outlined tracepoint, a process that is executed from the operating system's user space. In this scenario, the BCC compiler toolset is utilized for this purpose.

This can be achieved by importing the Python frontend library of the BCC toolset. Utilizing this library, the eBPF program can be attached in the following manner:

```python
from bcc import BPF

b = BPF(src_file="./ebpf.c")
b.attach_tracepoint(tp="sched:sched_switch", fn_name="onswitch")
```

A new BPF instance is initialized in this step, and the eBPF program is attached by loading it from the local filesystem. It is possible to have multiple functions defined within a single file, in this case `ebpf.c`, which can be attached to the same or different tracepoints, as well as to Kprobes or other available tracing systems.

Note: The `onswitch` function name, previously defined in the eBPF program, is referenced in this program snippet.

6.2.2 Memory utilization probe

EBPF Part

Like the CPU probe implementation, the Memory utilization probe will be written in C and attached to an OS event. However, instead of using tracepoints, this probe will be attached to Kprobes, as in this case, they offer a more direct approach for reading memory allocations than tracepoints.

As outlined in the design section, measuring two events is necessary to evaluate the memory utilization of a specific program containing both allocations and deallocations. For this purpose, the `kmalloc` and `kfree` Kprobes will be utilized. Additionally, an eBPF function will be attached to a third Kprobe, `kretalloc`, to query the actual storage allocated in the `alloc` call and to record its memory address. This information is important for the `dealloc` function to determine the memory size deallocated for a given address, as the `kfree` Kprobe does not provide this size data.

To identify such Kprobes, one can query the OS by listing all available Kprobes within the file `/proc/kallsyms`. It's important to note that Kprobes are more dynamic and can vary across different Linux distributions. They may also be blacklisted due to specific Linux security features. Therefore, Kprobes can differ between systems, and it is important to verify the availability of the required Kprobes, especially when encountering issues in attaching eBPF programs to them.

Implementing the eBPF probe for memory utilization shares similarities with the CPU probe but differs slightly in querying event-specific information for Kprobes. In the following, a sample structure of the
`alloc` probe can be viewed:

```c
int alloc(struct pt_regs *ctx, size_t size, gfp_t gfp) {
    u32 pid = bpf_get_current_pid_tgid() >> 32;

    //function implementation
    return 0;
}
```

Note: It is interesting to mention that the function receives the allocation request size as an argument and not as a struct pointer containing further information anymore. This way of passing data to the function fundamentally differs from tracepoints and should be kept in mind during development.

User-space Part

In the user-space part of the memory utilization probe, it can be observed that its operation is similar to the CPU utilization probe. However, an additional step is required to aggregate memory utilization across all PIDs accurately. This requirement is further highlighted in the design section 5.2.2. This step exists due to the eBPF program only being able to measure memory allocations and deallocations occurring after the initiation of the eBPF program are measured. Thus, allocations made before this point should be accounted for. Therefore, it is necessary to initially fetch the memory already allocated for each PID until the eBPF probe is registered. Following this, the size of new allocations is added, and the size of deallocations is subtracted to obtain the final memory utilization.

$$initial\_memory\_usage + allocation\_size - deallocation\_size = current\_memory\_usage$$

Predefined OS interfaces can be used to fetch the required information about the initial memory utilization. Especially, the `/proc` directory, containing process-related information in the OS, is a valuable source. The focus lies in querying the `/proc/PID/status` file for the task at hand. This file contains multiple lines detailing the memory utilization of a specific PID. The line of interest is `VmSize:`, which defines the exact amount of memory, in **kilobytes**, utilized by the process. This information is extracted for every newly detected PID for the eBPF program through the user-space system. The inability to perform I/O operations within eBPF requires this approach, as reading out this data directly within the eBPF program is impossible.

In addition to memory utilization data, the storage of allocation and deallocation counts in separate eBPF maps is also implemented. These maps maintain a simple integer count, incrementing each time an allocation or a free operation is executed. Implementing these increment maps is straightforward, as it works very similarly to the maps described in the CPU probe implementation and thus does not warrant further explanation.

6.2.3 Disk I/O utilization probe

EBPF Part

The nature of the disk I/O probe is very straightforward. It should be mentioned that an eBPF map is being stored, where the values of this map hold structs, with each struct containing the following four parameters:

- **Reads:** Measures the I/O read operations per second for that process.
- **Writes:** Measures the I/O write operations per second for that process.
- **Read bytes:** Measures the amount of bytes read from the disk.
- **Written bytes:** Measures the amount of bytes written to the disk.

The values within this struct will be continuously incremented such that the user-space part can correctly calculate the IOPS, written bytes per second, and read bytes per second.

It is important to differentiate between write and read operations by reading out the operation from the tracepoint struct `tracepoint__block__block_rq_complete`, which contains information about the number of sectors affected, as well as the operation such as Read, Write, Sync, Flush, and more.

User-space Part

The user-space implementation is similar to the other probes, just that it has to deal with this new map structure, which contains structs instead of primitive types. It attaches the eBPF probe to the `block_rq_complete` tracepoint, which is called every time a block request operation is completed.

The implementation details of the probes are thus concluded. It is important to note that the map of the eBPF program is reset back to its defaults every time it is read out by the user-space program, ensuring that the process of storing data starts from a fresh point after each readout. Without resetting them, continuously incrementing values in the map may lead to integer overflows or similar issues. Additionally, aggregating that data in the user-space counterpart would be made harder by continuously incrementing the values.

6.3 Aggregator

6.3.1 Dynamic module loading

The Aggregator is written in Python and needs to take care of a few steps, such as loading the probes into the kernel correctly. Following the objective of Extensibility and Simplicity, the Aggregator will dynamically load probes from a directory. Any probe within this directory will be utilized, and each probe comes in a package containing the eBPF C program, the user-space counterpart, and a `setup.toml` file with the necessary probe-specific configuration.

The relevant program section in the Aggregator allows for this dynamic loading of probes and can be introspected in the program snipped below.

```python
def update_probe(module, config, probe_name):
    while True:
        result = module.update()
        with store_lock:
            shared_store[probe_name] = result
        time.sleep(int(config["aggregation_interval"]) / 1000)

for folder in os.listdir("./probes"):
    if not os.path.isdir(os.path.join("./probes", folder)):
        continue
    try:
        with open(os.path.join("./probes", folder, "setup.toml")) as f:
            config = toml.load(f)
            print(
                f"Loaded config for probe {folder}: {config}")

        module = importlib.import_module(f"probes.{folder}.probe")
        module.init(config)

        thread = threading.Thread(target=update_probe, args=(
            module, config, folder))
        thread.daemon = True
        thread.start()
        threads.append(thread)
```

It can be seen that the probe's `setup.toml` is first loaded, and then the Python module, which is the user-space counterpart to the eBPF program, is dynamically loaded. In this module, the `init` function is called. Taking the configuration setup, this init function can perform rudimentary data preparation, such as pre-fetching memory utilization in the case of the memory probe.

After the conclusion of the `init` function, the module update function is registered through the `update_probe` function, which runs in a separate thread. In the `update_probe` function, the update function of the probe is continuously called at the specified interval [2] of that probe, allowing for the parallel execution of the eBPF probes.

[2]Provided in milliseconds in the setup.toml

Further down, the main aggregator program attempts to `join` all these threads, which prevents the main program from terminating until all threads have finished. This ensures continuous execution, as the threads never terminate under normal operation.

6.3.2 Data advertisement API

Advertising the aggregated data is done via the Prometheus Python client. This library allows for storage of the results of the probes in several different ways, which are then published through an HTTP web server endpoint. This allows an aggregation system such as Prometheus to consume the aggregated metrics and base further calculations on them.

Here is an example of how easy it is to publish data:

```
g = Gauge("ebpf_cpu_utilization", "CPU utilization", ["PID", "name"])
g.labels(PID="<some-PID>", name="<some-name>").set(100)
```

Several custom labels can be defined for these metrics, which can then be used to find the correct data points in Prometheus using Prometheus' built-in query language, Promql.

Once the system is launched, waiting for Prometheus to consume the advertised data becomes the next step, allowing for subsequent querying. Grafana[3] will be used to visualize the query results from Prometheus.

A look at how the data is displayed in Grafana in figure 6.1 is an example of possible visualizations. Further analysis of these results will be conducted in the following assessment chapter.

[3]An open source dashboard for data visualization

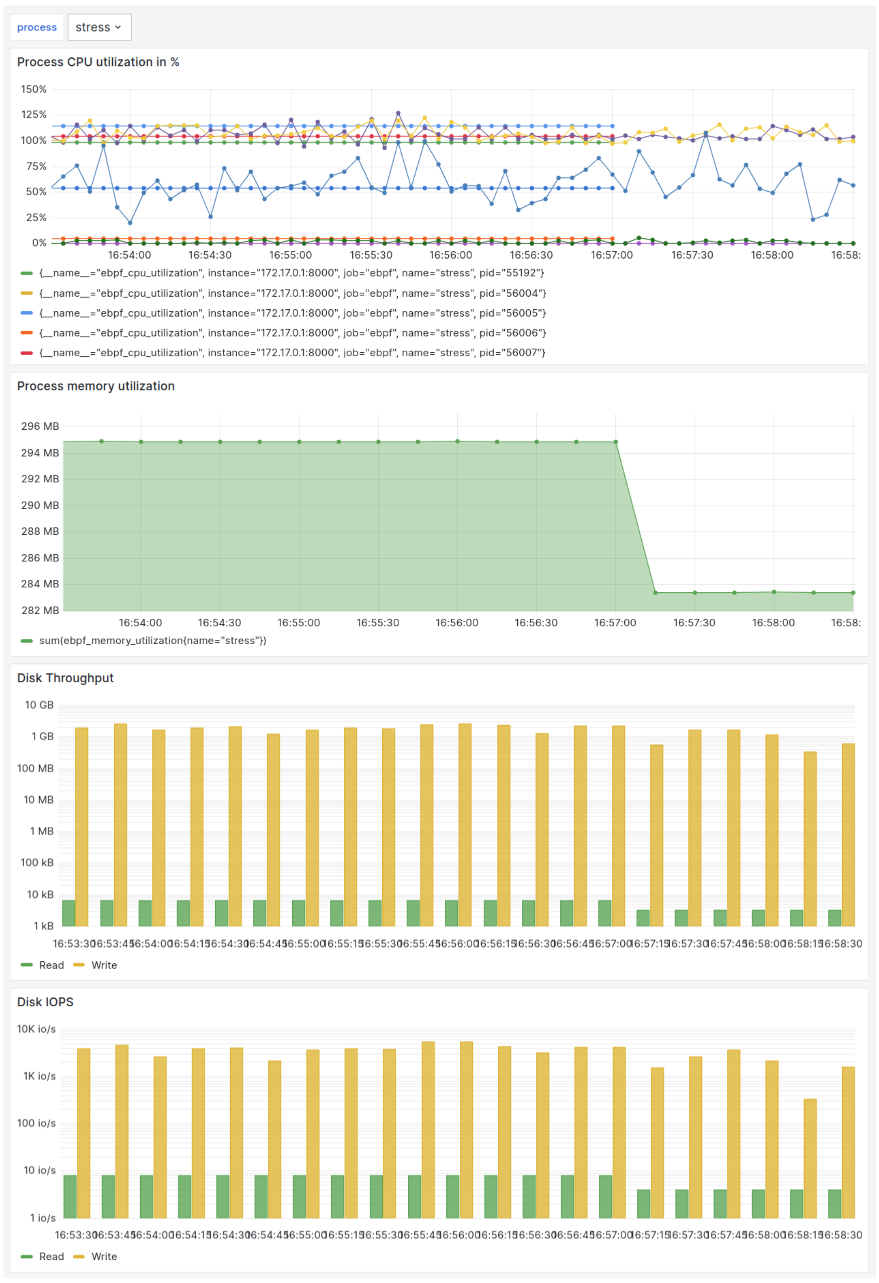

process
stress
Process CPU utilization in %
{__name__="ebpf_cpu_utilization", instance="172.17.0.1:8000", job="ebpf", name="stress", pid="55192"}
{__name__="ebpf_cpu_utilization", instance="172.17.0.1:8000", job="ebpf", name="stress", pid="56004"}
{__name__="ebpf_cpu_utilization", instance="172.17.0.1:8000", job="ebpf", name="stress", pid="56005"}
{__name__="ebpf_cpu_utilization", instance="172.17.0.1:8000", job="ebpf", name="stress", pid="56006"}
{__name__="ebpf_cpu_utilization", instance="172.17.0.1:8000", job="ebpf", name="stress", pid="56007"}
Process memory utilization
sum(ebpf_memory_utilization{name="stress"})
Disk Throughput
Read
Write
Disk IOPS
Read
Write

Figure 6.1: A Grafana dashboard visualizing the metrics for the `stress` process, which is a tool used to stress-test systems

7 Assessment

Having built the eBPF benchmarking system and extracted the necessary information, the next step is to assess its advantages over conventional benchmarking systems. As specified in the methodology chapter 4 of this thesis, eBPF will be evaluated in four key areas and compared to similar, more conventional tools. The areas of comparison are:

- **Correctness:** The correctness of the homebrew benchmarking system will be evaluated against well-established conventional systems to verify its capability of meeting this core requirement. This will involve verifying results against other systems and highlighting key advantages of eBPF.
- **Modularity:** The modularity of the eBPF benchmarking system, and eBPF in general, will be compared to other systems such as bpftrace, perf, and top.
- **Simplicity:** The system's simplicity will be compared by examining its general complexity versus industry-established conventional systems.
- **Scalability:** By building a proof of concept, an exploration into how the system scales with large-scale deployments versus other conventional systems will be conducted.

7.1 Correctness

To measure the system's correctness, a stable testing environment will be defined, running tests in a way that minimizes outside interference. A cloud-based virtual machine with limited memory, disk, and CPU capacity from a common cloud provider will be used. To reduce the risk of noisy neighbors in the virtual machine deployment, dedicated[1] CPU cores will be secured.

The chosen testing environment is a machine with four dedicated CPU cores, 16 gigabytes of RAM, and 160 GB of built-in disk space. An Ubuntu Linux deployment will be bootstrapped on this virtual machine, providing the capabilities needed to load eBPF programs into the kernel and execute them without limitations.

7.1.1 Procedure

The virtual machine will use a reference workload to evaluate the correctness of the eBPF benchmarking system and other conventional systems.

The `stress` cli utility has been chosen, allowing for an easy definition of how the system should be stressed. It enables the configuration of command line arguments to specify the number of stress test workers for each system component:

[1]CPU cores are often shared across multiple customers, especially in cloud environments. It must be ensured that the CPU cores are solely dedicated to this use case.

- **CPU workers:** Each worker will fully consume one core's cycles by calculating square roots, primarily loading the CPU ALU.
- **I/O workers:** I/O workers will stress disk I/O by repeatedly calling the `sync()` operation, prompting the OS to flush temporary memory data to disk.
- **Virtual memory workers:** These workers will frequently call `malloc()` and `free()`, managing allocations and deallocations of memory in the OS.

This approach allows for a configurable workload impacting the system in various ways. This workload will be benchmarked by the eBPF and conventional systems, enabling comparison of results.

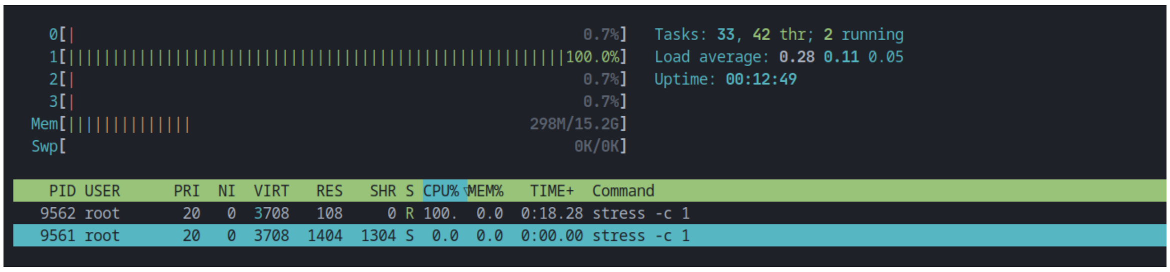
The discussion will then focus on eBPF's advantages regarding correctness, which other systems might lack.

7.1.2 CPU probe

When the stress command `stress -c 1` is executed on the virtual machine, it spawns one CPU worker, fully consuming one CPU core. Therefore, the benchmarking systems should show at least one CPU core at 100%

Htop

Htop, a utility tool often used by system administrators, provides a human-readable output of system utilization. Figure 7.1 shows the output from `htop` for the `stress` command. It displays a bar chart diagram at the top indicating each CPU core's utilization and a list of processes with details like the process ID, executing user, virtual memory usage, CPU utilization, and more. While running the stress command, the output indicates that one CPU core is at around 100% utilization.



```
    0[|                                                        0.7%]   Tasks: 33, 42 thr; 2 running
    1[||||||||||||||||||||||||||||||||||||||||||||||||||||||||100.0%]   Load average: 0.28 0.11 0.05
    2[|                                                        0.7%]   Uptime: 00:12:49
    3[|                                                        0.7%]
  Mem[|||||||||||||                                     298M/15.2G]
  Swp[                                                      0K/0K]

    PID USER       PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
   9562 root        20   0  3708   108     0 R 100.  0.0  0:18.28 stress -c 1
   9561 root        20   0  3708  1404  1304 S  0.0  0.0  0:00.00 stress -c 1
```

Figure 7.1: A filtered output of the htop command, displaying on the stress process

Prometheus

With the necessary tools and environment in place, the system can be evaluated against other benchmarking systems, with the help of Prometheus.

The scraped metrics results fetched by Prometheus from the built-in node-exporter extension for the virtual machine will be inspected in the following paragraphs. The node-exporter utilizes Linux OS interfaces, such as the `/proc` file system, to extract relevant metrics about processes and the overall system. Additionally, it can extract metrics through `perf`, enabling the measurement of advanced OS hardware metrics.

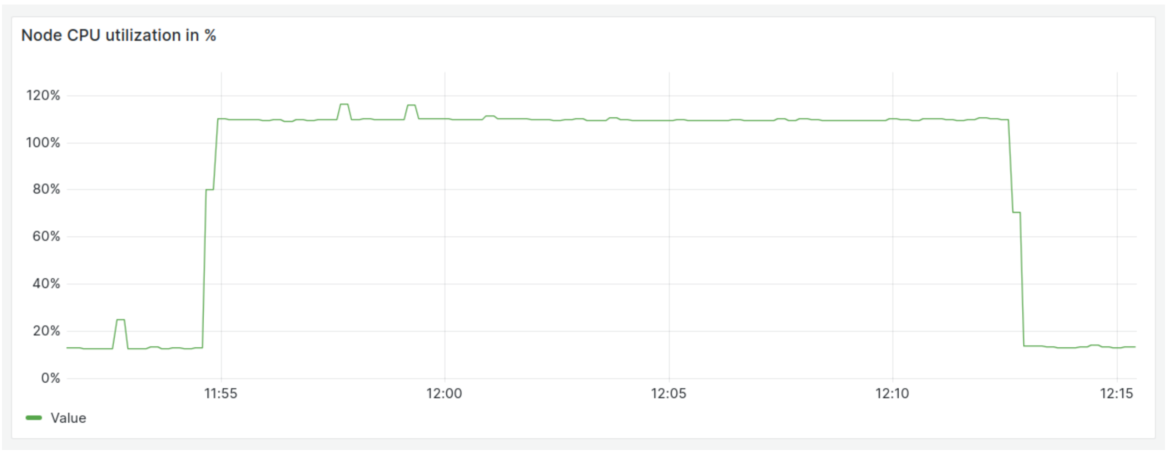


Figure 7.2: The node CPU utilization, provided by the Prometheus node-exporter, during stress testing

As shown in figure 7.2, the server was under high stress during benchmarking. The line within this chart represents the total CPU usage of the system. 100% without the chart represents the full utilization of one core within the system, and as it has 4-cores, the total maximum utilization that could be reached here is 400%. This is also why an overall system utilization of around 110% can be observed, as the stress command takes up 100%, and the remaining 10% are caused by all the other processes running in the OS. A utilization statistic can be extracted by using the following PromQL query `sum(1 - rate(node_cpu_seconds_totaljob="node", mode="idle"[30s]))`. This PromQL query first calculates the rate of the `node_cpu_seconds_total` metric, as this metric, by default, is an ever-incrementing value. The `rate` aggregation takes the derivative of that ever-incrementing value to provide better insight into how many CPU seconds are spent at any time. This is normalized by subtracting the value from 1 since the "idle" task is being measured, meaning the time the CPU spent doing nothing is being measured. Therefore, $1 - idle$ represents the CPU seconds the CPU spent processing relevant tasks. This would provide the utilization of each CPU core; however, as the interest lies in the overall system utilization, the utilization of all cores will be summed up to a single value, indicating the overall system utilization.

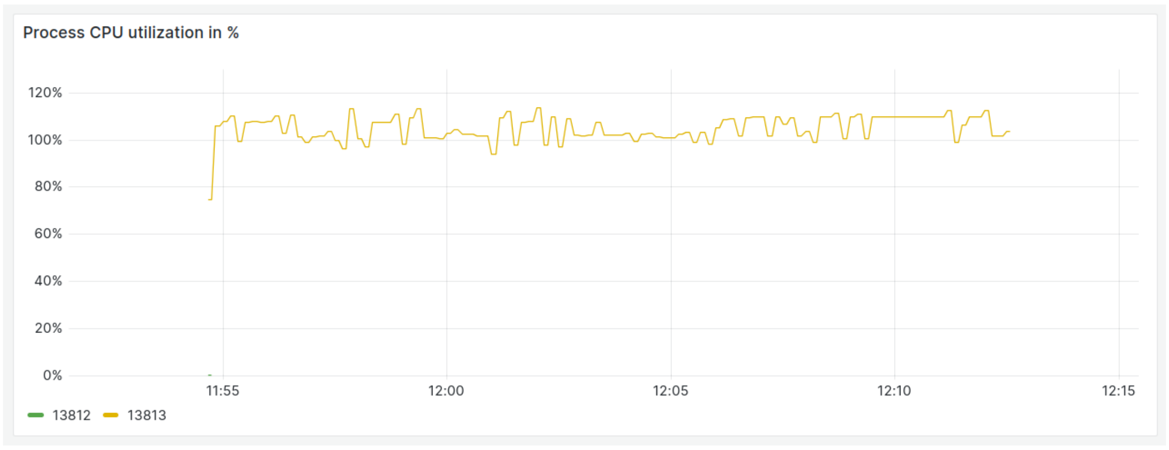


Figure 7.3: The process utilization during stress testing

The result from figure 7.2 can now be compared with the measurements of the eBPF CPU utilization probe, found in figure 7.3. In this chart, the CPU utilization of the stress command is presented. It can be seen that the eBPF probe measures realistic utilization values, all hovering around 100%. Some volatility in the measurements is observed. This indicates that the new eBPF CPU utilization probe measures the stress command correctly and can be used to base real-world decisions on its measurements.

Volatility

In figure 7.3, some imprecision in the eBPF CPU measurements is observed, as these sometimes exceed 100% utilization. Theoretically, this should not be possible due to the calculation defined in the design section, which adds to the CPU's discrete-time processing of a single task. The sum of these timings should not exceed the benchmark duration of the process, as this would imply the process ran longer than tested for. This phenomenon can be explained by understanding how multiprocessing systems operate.

In such systems, for load balancing or core efficiency, the scheduler may schedule the process onto a different CPU core during execution. When the scheduler starts scheduling the process onto the new CPU core, the eBPF program is called for that CPU core as well, while the previous instance of the process might still be in the teardown phase in the previous CPU core, and thus the eBPF program has not yet been executed at that point in time. This results in overlapping intervals in the measurement process, leading to minor overshooting of CPU utilization. Compensating for this could involve tracking all processes' exact start and end times across CPU cores and subtracting half the sum of all overlapping intervals from each process's CPU times. However, for simplicity, this thesis will not implement this extra safeguard. The extensibility of eBPF, though, makes such an implementation trivial.

The chart also helps explain why CPU utilization rarely dips below 100%. Regardless of how the scheduler optimizes process scheduling, one process is always fully utilizing the CPU. The difference between the spike and 100% can be considered ”wasted” time due to rescheduling the process. This insight could be used to measure the efficiency of the OS scheduler, suggesting a potential area for further research.

Similar phenomena are observed with other tools like `htop`, which often record process utilization values above 100%, even when only a single CPU core is used by the process.

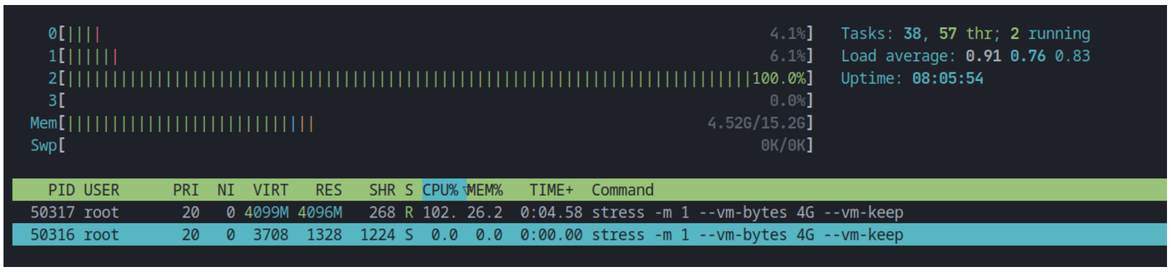
Several advantages of eBPF can be made out, including the high precision of its measurements and the extensive flexibility it offers. This flexibility enables the implementation of additional logic into the CPU utilization probe to compensate for measurement imprecision, a task that would be, by a large margin, more challenging in a user-space program.

7.1.3 Memory probe

In the following paragraphs, the correctness of the memory probe will verified against conventional benchmarking systems. Initially, htop will be used to confirm that the stress command is indeed stressing the system's memory. The stress command will be started with the total memory workers set to 1 and the allowed memory usage to 4 gigabytes, using the command `stress -m 1 --vm-bytes 4G --vm-keep`.

Htop

Running `htop`, a relatively high memory utilization of the stress command, around 4 gigabytes or 4099 megabytes, is observed, as shown in figure 7.4. It is noted that although only memory workers have been activated with the stress command, these workers still exert a significant load on the CPU, resulting in CPU core three being under full load.



```
    0[||||                                                                   4.1%]   Tasks: 38, 57 thr; 2 running
    1[||||||                                                                 6.1%]   Load average: 0.91 0.76 0.83
    2[|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||100.0%]   Uptime: 08:05:54
    3[                                                                       0.0%]
  Mem[|||||||||||||||||||||||||||                                   4.52G/15.2G]
  Swp[                                                                     0K/0K]

    PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
  50317 root       20   0 4099M 4096M   268 R 102. 26.2  0:04.58 stress -m 1 --vm-bytes 4G --vm-keep
  50316 root       20   0  3708  1328  1224 S  0.0  0.0  0:00.00 stress -m 1 --vm-bytes 4G --vm-keep
```

Figure 7.4: Htop output after running the stress command and spinning up the memory worker

Prometheus

The memory utilization measured by Prometheus' node-exporter will be examined by aggregating the currently used memory in the system with the PromQL query:
`node_memory_MemTotal_bytes - node_memory_MemAvailable_bytes`. In this query, the total available system memory is subtracted from the still available memory to calculate the used memory. This used memory encompasses the memory consumed by processes running in the system and the operating system itself. Node-exporter gathers these metrics from the `/proc` directory.

When this PromQL query is rendered in Grafana, the output can be seen in figure 7.5.

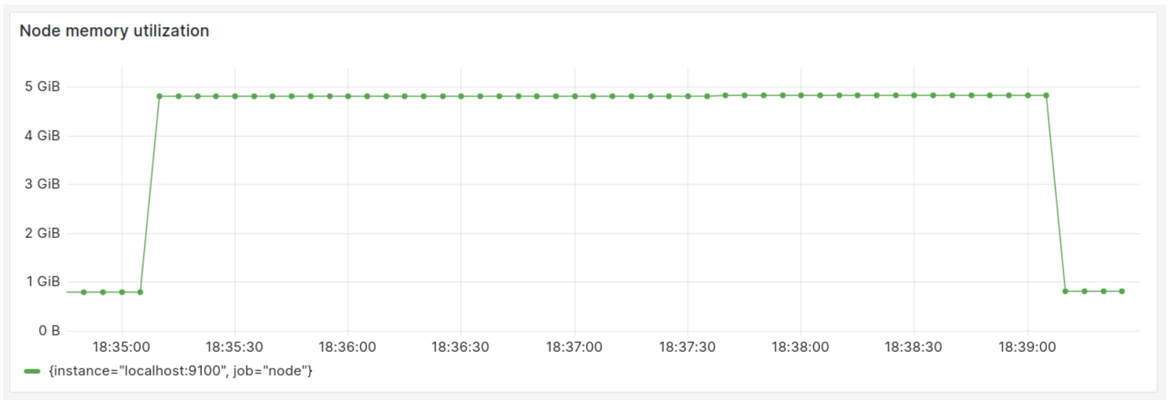


Figure 7.5: Node memory utilization during the execution of the stress command

Comparing figure 7.5 to the results that the eBPF system measured, it can be identified that it indeed displays the correct memory amount being used 7.6.

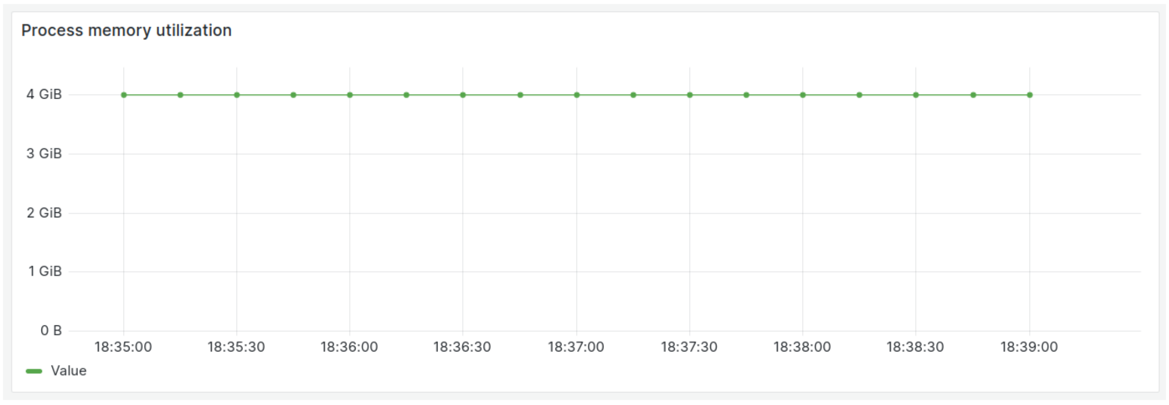


Figure 7.6: The memory measurement of the eBPF memory probe

With this eBPF-based system, it is also possible to measure further intricacies into memory behavior that are not available through the `/proc` directory and, as such, are also not available to conventional benchmarking tools, highlighting eBPF's advantage over conventional systems in terms of flexibility and capabilities. eBPF will also be able to measure every single allocation and deallocation in a workload, allowing for memory spike detection. At the same time, a sampling-based system such as the node-exporter will only be able to measure rough insights into process memory usage, therefore making memory usage not very traceable.

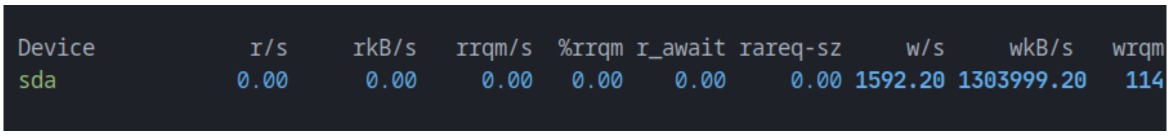
7.1.4 Disk I/O probe

A look at the disk I/O statistics will be taken by running the `stress --hdd 10 --io 1` command, which spawns ten disk workers and one I/O worker. This command repeatedly calls the `write()` and `sync()` C methods, writing large amounts of data to the system disk. To measure this, the stress command will first be tested with metrics provided by `iostat`, a utility that offers a good overview of disk utilization.

Iostat

To measure disk utilization with iostat, the `sysstat` package must be installed on the OS. Then, the `iostat` program can be run with the command `iostat -dx /dev/sda`, which monitors the `/dev/sda` disk, the mount point of the primary hard disk in the testing environment.

Upon running the program, an output is received, as shown in figure 7.7.



```
Device            r/s     rkB/s   rrqm/s  %rrqm r_await rareq-sz     w/s     wkB/s   wrqm
sda              0.00      0.00     0.00   0.00    0.00     0.00 1592.20 1303999.20   114
```

Figure 7.7: Disk utilization output from iostat

This program offers a useful set of interesting metrics. The `w/s (writes per second)` is particularly noteworthy in this case, as it represents all write operations per second on that specific device. In this case, the `stress` command causes a load of 1592.2 writes per second.

Prometheus

The same metrics as those provided by the program are also advertised by the Prometheus node-exporter through the `node_disk_writes_completed_total` metric, as observable in figure 7.8. Interestingly, unlike the relatively stable CPU utilization, the IOPS here vary quite significantly, notably within a range of 20%. This variation can be attributed to many underlying reasons, such as other priorities of the OS that might cause the write IOPS to fluctuate to this extent.

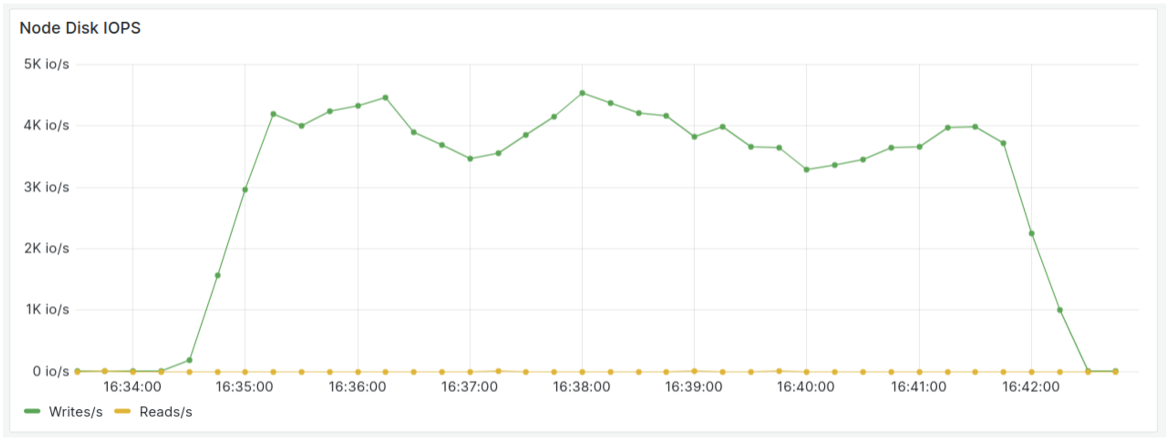


Figure 7.8: Disk IOPS output of the Prometheus node-exporter

Inspecting the output of the eBPF disk I/O utilization probe in figure 7.9, which also measures the IOPS of the system, a strong positive correlation is observed between the graph from the Prometheus node-exporter and the disk I/O utilization probe. Both peak at roughly the same places and measure relatively similar values. However, it is noted that these graphs do not represent an exact overlap, as the Prometheus and the `iostat` output appear to be slightly higher than the results from the eBPF probe. This discrepancy can be attributed to the differences in measurement approaches. The eBPF probe measures only the impact of the stress process on the disk, whereas the `iostat` and node-exporter aggregate their values on a disk level, including other processes and the OS operating on that disk. The Prometheus/Grafana installation stack also contributes to the disk load, as storing these metrics utilizes a significant amount of the available disk throughput and IOPS. This process-level measurement capability of the eBPF probe is an advantage that would be more cumbersome to achieve via a conventional user-space benchmarking system.

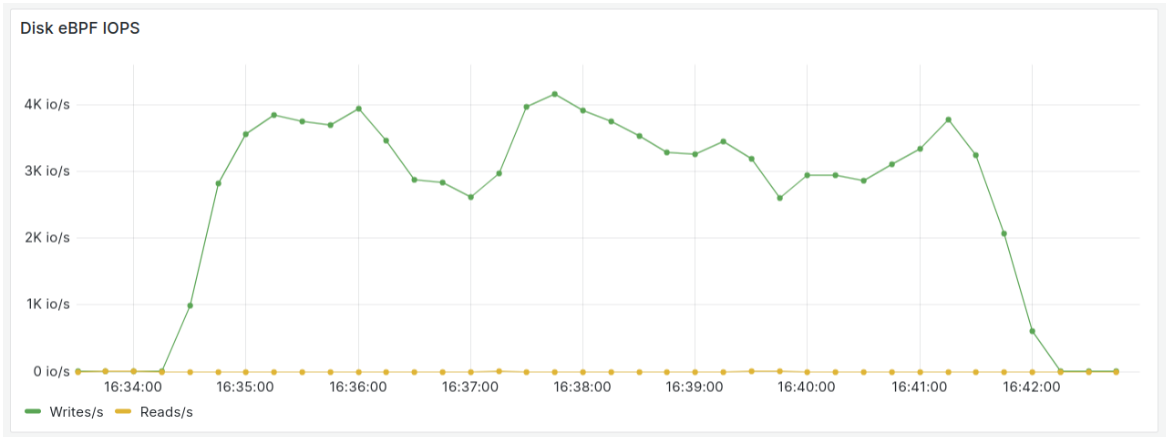


Figure 7.9: eBPF probe output for the IOPS measurement

Having compared the metric measurements against two other standardized benchmarking systems, it can be confidently concluded that the measurements of these specific metrics are accurate. While there may be small differences in the results, these can primarily be attributed to the granularity of the eBPF programs, which measure at the process level, and other minor differences that were not accounted for.

7.2 Modularity

This section will assess how the eBPF-based system compares to conventional benchmarking systems in modularity. Modularity can be defined in various ways; in this context, modularity refers to the ease of extending new probes and measurement approaches, as well as the ability to remove any measurements that are not necessary.

7.2.1 Cli tools

The iostat and the htop command rely on the `/proc` directory to fetch their metrics. In terms of modularity, these do not allow for easy extensibility. The different command line arguments that can be provided to the client are capable of limiting the user to which data they want to see, but in terms of extendability, iostat, as well as htop, provide no way of extending their functionality except for manually forking the repository and recompiling them with added features, which would require deep knowledge of their code.

7.2.2 Node Exporter

Like most other benchmarking systems and iostat, the Prometheus node-exporter relies on the `/proc` file system to fetch its metrics. It does allow for advanced configuration, as can be viewed in their documentation [Pro24], but it does still limit the user to the predefined toolset it provides. Regarding granularity, the node-exporter is especially limited concerning process-level metrics, mainly aggregating metrics on an OS or device level. Therefore, while very interesting, especially for multi-node benchmarking, it only helps a little with specific workload benchmarking during runtime.

7.2.3 eBPF

The eBPF approach has been observed to allow for extremely high modularity and extensibility, particularly in measurement implementation details. Despite some limitations of eBPF, such as program complexity restrictions and the exclusion of I/O operations, it enables the construction of powerful measurement systems. eBPF probes can measure metrics with high precision, and it is more likely that users would opt to reduce this precision to mitigate system impact rather than due to eBPF reaching its limits.

Attaching eBPF programs to various probes within the system, including tracepoints, kprobes, perf events, and others, provides vast capabilities for gaining insights into the OS with minimal unnecessary system impact. Its capacity to measure network metrics and high-level events aids in building advanced benchmarking systems. However, this extended complexity requires deeper knowledge of system internals, of which documentation is often hard to come by other than reading the Linux source code.

7.3 Simplicity

7.3.1 Cli tools

In terms of simplicity, command line tools offer the user the easiest way to read out system utilization. There are no dependencies [2] and thus no further systems that need setting up. Many CLI tools even offer an advanced human-readable mode often enabled by the `-h` flag, and thus make it even easier for the user to read out their measurements quickly.

7.3.2 Node exporter

The Prometheus node exporter and similar tools require a more extensive setup within the system. To begin reading measurements, Prometheus must first be set up on the same or another node within the network. Following this, the node-exporter must be installed on the node to be measured. Both Prometheus and the node-exporter may require additional configuration to work together correctly. Only after these steps is it possible to access the metrics. Thus, while the Prometheus node exporter and similar systems are powerful, they do not excel in simplicity, but this is not due to the core nature of user-space benchmarking solutions but rather due to the implementation specifics of the Prometheus & node-exporter stack.

7.3.3 eBPF

In this scenario, a similar architecture to the Prometheus node-exporter was followed by hosting aggregated metrics on an HTTP server, which is then scraped by a Prometheus instance. This system causes added complexity, especially when developing custom eBPF probes, which demands careful consideration of whether such a benchmarking system is appropriate. Aside from this approach, there are alternative methods to utilize eBPF for monitoring and benchmarking systems. An example is bpftrace [iov24b], a command-line interface that facilitates the rapid development of tracing systems. However, using bpftrace also involves learning its tracing language, which could be a hurdle for those seeking to measure a workload quickly without significant learning overhead.

It can be said that while common benchmarking systems such as top, iostat, and other systems that rely on reading out the `/proc` file system or rely on `perf` events are very limited in their capabilities, their ease-of-use outshines an eBPF based system, at least when implemented from scratch, and might be a sufficient solution most of the times. Therefore, eBPF does not hold an advantage over conventional benchmarking systems regarding ease of use and simplicity.

[2]Except for some OS level libraries such as glibc

7.4 Scalability

Scalability describes the viability of applying the benchmarking system to larger deployments, more complex system architectures, and advanced aggregations across different nodes.

7.4.1 Cli tools

CLI tools like htop and iostat do not offer data aggregation across multiple nodes. Their sole purpose is to measure the system utilization of the node they are running on. Thus, it can not be extrapolated to larger multi-node deployments where parallel measurements of these nodes and the metrics within are crucial.

7.4.2 Node exporter

Due to the approach that Prometheus takes, the node-exporter can be seen as extremely scalable. As long as a Prometheus deployment runs on the same machine or a different machine in the same network, multiple nodes can be scraped and aggregated by that Prometheus instance. Scaling is as easy as deploying a new data provider, such as the node-exporter on a new node, and configuring Prometheus to scrape that node. The Limits start when Prometheus can no longer handle the amount of incoming metrics. However, even in that case, there are ways to scale out the Prometheus deployment vertically or horizontally to guarantee better availability.

7.4.3 eBPF

When talking about scalability in the case of eBPF, it completely depends on the implementation one has opted for. In this case, similar scalability freedom to the Prometheus node-exporter is achievable, given a similar architectural approach. Other eBPF benchmarking solutions, such as bpftrace, require further solutions and systems for higher scalability. As eBPF represents a technology or toolset more than a full benchmarking system, implementation details can vary immensely, and thus, so can the usability in terms of scaling up to multi-node deployments. Thus, eBPF can be very scalable when implemented correctly.

7.5 Assessment summary

Summarizing these findings, eBPF generally holds up very well and excels in correctness, modularity, and scalability compared to conventional benchmarking systems. Simplicity could be more straightforward as eBPF requires deep knowledge of the internal workings of the host OS, core technologies, knowledge of C and Python, and theoretical knowledge of the inner workings of an OS. Thus, in three of the proposed four points, eBPF shows to have advantages over conventional benchmarking solutions.

8 Conclusion

In this exploration, the advanced capabilities of eBPF technology have been discovered, particularly in providing precise measurements of operating system resources such as CPU, memory, and disk I/O utilization. Nonetheless, with great power comes great responsibility. The initial challenge when developing eBPF-based benchmarking systems lies in grasping the advanced concepts of eBPF and the workflow for developing eBPF programs. This requires a deeper understanding of low-level systems programming and of the C programming language, as well as interactions with the Linux kernel, where documentation and other helpful aids on eBPF are sparse and require deeper research into the Linux source code to be able to understand and develop these capable systems.

Understanding measurements of specific metrics is essential for developing an effective benchmarking tool, and this demands advanced knowledge of the Linux OS, including its interfaces, kernel behaviors, and other system intricacies.

Once these obstacles are tackled, the true potential of eBPF can be uncovered. Its immense capabilities allow developers to create extremely capable programs, especially in system observability and other areas, such as networking. Due to its low-level nature and independence from user-space scheduling constraints, eBPF provides reliable insights into the kernel, allowing developers to access and measure metrics that are beyond the capabilities of conventional benchmarking systems, such as htop, iostat or anything that relies on the `/proc` directory as a data source.

Implementing a correct, easy-to-understand, extensible, and scalable benchmarking system is possible. Thus, eBPF is a viable replacement, with several advantages, for benchmarking systems that are otherwise restricted by user-space execution limitations.

8.1 Final Results and Recommendations

Compared to standard benchmarking systems, eBPF offers several distinct advantages, as described in the assessment summary 7.5. While it allows for building systems with a high degree of correctness, modularity, and scalability, it does not have any advantages over conventional benchmarking solutions in terms of simplicity. Therefore, it is advisable to implement eBPF-based benchmarking systems only after defining the use case, validating if utilizing eBPF represents a sufficient benefit over conventional solutions, and considering the potential challenges. These challenges include a higher complexity in extracting operating system information from eBPF programs and the inherent limitations of eBPF in supporting advanced programs. Systems that primarily leverage eBPF may become complex and difficult for new developers and users to reason.

8.2 Implications for Further Research

This thesis has laid the groundwork by addressing the basic operational principles of implementing an eBPF-based benchmarking system that scales well, is easy to understand, and assesses how it fares compared to conventional benchmarking solutions. However, there remains a great amount of unexplored areas regarding the full extent of eBPF's capabilities. Future research should delve into advanced applications of eBPF, exploring its potential in more complex system environments regarding system utilization monitoring and workload benchmarking and its integration with other technologies. Such studies could provide valuable insights into optimizing system performance and expanding the limits of current benchmarking methodologies. Exploring these advanced approaches can pave the way for innovative applications in systems programming and performance optimization. Further research would be especially interesting in containerized applications and how well eBPF fares in terms of benchmarking these compared to conventional container benchmarking solutions. Evaluating how much an eBPF-based benchmarking solution might impact OS resources compared to conventional solutions might also be of interest for further research, as eBPF does not adhere to scheduling limitations other processes are bound to, and thus might also bring its own set of problems that require further analysis and exploration.

List of Figures

Bibliography

[4bs24] 4bsd file tree. https://minnie.tuhs.org/cgi-bin/utree.pl?file=4BSD, 2024. Accessed: 2024-01-31.

[ALRP15] Ehab Alkhanak, Sai Lee, Reza Rezaei, and Reza Parizi. Cost optimization approaches for scientific workflow scheduling in cloud and grid computing: A review, classifications, and open issues. *Journal of Systems and Software*, 113, 12 2015.

[Aut24] Prometheus Authors. Instrumentation. https://prometheus.io/docs/practices/instrumentation/, 2024. Accessed: 2024-01-08.

[Bla16] Brenden Blanco. [PATCH v10 00/12] Add driver bpf hook for early packet drop and forwarding. Netdev Mailing List, 2016. Accessed: 2024-01-17.

[CCS] Clement Joly Cyril Cassagnes, Lucian Trestioreanu and Radu State. The rise of ebpf for non-intrusive performance monitoring.

[Clo23] Cloudflare. ebpf_exporter. https://github.com/cloudflare/ebpf_exporter, 2023. Accessed: 2024-01-14.

[Con24a] The Aya Contributors. Aya - a rust ebpf library. https://github.com/aya-rs/aya/blob/main/README.md, 2024. Accessed: 2024-01-08.

[Con24b] The BCC Contributors. Bcc - tools for bpf-based linux io analysis, networking, monitoring, and more. https://github.com/iovisor/bcc/blob/master/README.md, 2024. Accessed: 2024-01-08.

[Din23] Artem Dinaburg. Pitfalls of relying on ebpf for security monitoring and some solutions. https://blog.trailofbits.com/2023/09/25/pitfalls-of-relying-on-ebpf-for-security-monitoring-and-some-solutions/, Sep 2023. Accessed: 2024-01-14.

[DSC] Paul Emmerich Alexander Kurtz Krzysztof Lesiak Dominik Scholz, Daniel Raumer and Georg Carle. Performance implications of packet filtering with linux ebpf. page 9.

[DSM19] Luca Deri, Samuele Sabella, and Simone Mainardi. Combining system visibility and security using ebpf. In *Italian Conference on Cybersecurity*, 2019.

[eBP23] eBPF.io. ebpf documentation, 2023. Last accessed: December 20, 2023.

[ebp24a] ebpf applications landscape. https://ebpf.io/applications/, 2024. Accessed: 2024-01-14.

[ebp24b] ebpf core infrastructure landscape. https://ebpf.io/infrastructure/, 2024. Accessed: 2024-01-14.

[eC24] The Cilium eBPF Contributors. Cilium ebpf - ebpf library for go. https://github.com/cilium/ebpf/blob/main/README.md, 2024. Accessed: 2024-01-08.

[Gre15] Brendan Gregg. bcc: Taming linux 4.3+ tracing superpowers, 2015. Accessed: 2024-01-17.

[iov24a] iovisor. INSTALL.md for bcc. https://github.com/iovisor/bcc/blob/master/INSTALL.md, 2024. Accessed: 2024-01-10.

[iov24b] iovisor Contributors. bpftrace: High-level tracing language for linux ebpf, 2024. GitHub repository.

[Jes23] Jesper Dangaard Brouer. Xdp - express data path, 2023. Accessed: 2024-01-17.

[JZW$^+$23] Jinghao Jia, YiFei Zhu, Dan Williams, Andrea Arcangeli, Claudio Canella, Hubertus Franke, Tobin Feldman-Fitzthum, Dimitrios Skarlatos, Daniel Gruss, and Tianyin Xu. Programmable system call security with ebpf, 2023.

[ker23] kernel.org. intel_pstate cpu performance scaling driver, 2023. Last accessed: January 1, 2023.

[LLV24] LLVM Project. The llvm compiler infrastructure, 2024. Accessed: 2024-01-28.

[Pro24] Prometheus Community. Node exporter, 2024. GitHub repository.

[Pur23] Pure Storage. Storage iops vs. throughput: Analysis of size modalities. Pure Storage Blog, 2023. Accessed: 2024-01-06.

[Tkc23] Linus Torvalds and Linux kernel contributors. bpf_common.h in linux kernel source. https://github.com/torvalds/linux/blob/0dd3ee31125508cd67f7e7172247f05b7fd1753a/include/uapi/linux/bpf_common.h, 2023. Accessed: 2024-01-14.

[Tkc24] Linus Torvalds and Linux kernel contributors. verifier.c in linux kernel source. https://github.com/torvalds/linux/blob/052d534373b7ed33712a63d5e17b2b6cdbce84fd/kernel/bpf/verifier.c, 2024. Accessed: 2024-01-14.

[TR17] Jeff Taylor and Paul Renno. What role will container technology play in the digital journey? *Forbes*, Jun 2017.

[Wea13] Vincent M. Weaver. Linux perf event features and overhead. *University of Maine*, 4 2013.

[WYY$^+$21] Tianjun Weng, Wanqi Yang, Guangba Yu, Pengfei Chen, Jieqi Cui, and Chuanfu Zhang. Kmon: An in-kernel transparent monitoring system for microservice systems with ebpf. In *2021 IEEE/ACM International Workshop on Cloud Intelligence (CloudIntelligence)*, pages 25–30, 2021.