No Two Snowflakes Are Alike: Studying eBPF Libraries' Performance, Fidelity and Resource Usage

Carlos Machado
INESC TEC & U. Minho
Braga, Portugal
carlos.e.machado@inesctec.pt

Bruno Gião
INESC TEC & U. Minho
Braga, Portugal
bruno.d.giao@inesctec.pt

Sebastião Amaro
IST Lisbon & INESC-ID
Lisbon, Portugal
sebastiao.amaro@tecnico.ulisboa.pt

Miguel Matos
IST Lisbon & INESC-ID
Lisbon, Portugal
miguel.marques.matos@tecnico.ulisboa.pt

João Paulo
INESC TEC & U. Minho
Braga, Portugal
jtpaulo@di.uminho.pt

Tânia Esteves
INESC TEC & U. Minho
Braga, Portugal
tania.esteves@inesctec.pt

Abstract

As different eBPF libraries keep emerging, developers are left with the hard task of choosing the right one. Until now, this choice has been based on functional requirements (*e.g.*, programming language support, development workflow), while quantitative metrics have been left out of the equation. In this paper, we argue that efficiency metrics such as performance, resource usage, and data collection fidelity also need to be considered for making an informed decision. We show it through an experimental study comparing five popular libraries: bpftrace, BCC, libbpf, ebpf-go, and Aya. For each, we implement three representative eBPF-based tools and evaluate them under different storage I/O workloads. Our results show that each library has its own strengths and weaknesses, as their specific features lead to distinct trade-offs across the selected efficiency metrics. These results further motivate experimental studies to increase the community's understanding of the eBPF ecosystem.

CCS Concepts

• **General and reference → Evaluation**; **Measurement**.

Keywords

eBPF, experimental study, eBPF libraries

ACM Reference Format:
Carlos Machado, Bruno Gião, Sebastião Amaro, Miguel Matos, João Paulo, and Tânia Esteves. 2025. No Two Snowflakes Are Alike: Studying eBPF Libraries' Performance, Fidelity and Resource Usage. In *3rd Workshop on eBPF and Kernel Extensions (eBPF '25), September 8–11, 2025, Coimbra, Portugal.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3748355.3748364

1 Introduction

The extended Berkeley Packet Filter (eBPF) has seen rapid adoption across industry and academia for a wide range of use cases, including profiling and tracing [8, 29, 34], networking [14, 19, 20, 26, 32, 33], and security [7, 10, 22].

eBPF '25, Coimbra, Portugal

ACM ISBN 979-8-4007-2084-0/2025/09
https://doi.org/10.1145/3748355.3748364

To support the growing adoption of eBPF, the community has developed a rich ecosystem of libraries, including BCC [30], bpftrace [27], libbpf [15], ebpf-go [6], Aya [2], among others [3, 16–18, 31].[1] These provide varying levels of abstraction, programming language support, and development workflows (see Table 1). For example, bpftrace offers a high-level scripting language abstraction ideal for rapid prototyping of tracing tools. Libraries like libbpf, ebpf-go, and Aya support *Compile Once–Run Everywhere (CO-RE)*, enhancing portability across kernel versions while catering to different user space languages—C, Go, and Rust, respectively.

Despite the proliferation of eBPF libraries, developers lack systematic guidance on their relative performance and efficiency, often leading to suboptimal choices that can impact production systems, especially in performance-critical, large-scale, or resource-constrained environments.

Related Work. Existing comparisons in the literature, primarily blog posts and documentation, focus mainly on feature sets, functionality, and programming language support rather than quantitative performance metrics. For instance, Brendan Gregg compares bpftrace and BCC based on their intended use cases, noting that bpftrace suits quick, ad hoc tasks due to its simplicity, while BCC is better for building full-fledged applications [11]. In a broader survey, Red Hat reviews several other eBPF libraries (*e.g.*, libbpf, libbpfgo, libbpf-rs, ebpf-go, Aya, and libxdp), focusing on language integration and development tooling [24]. Likewise, Liz Rice's book [25] and a Chirp blog post [23] expand the landscape by covering additional libraries (*e.g.*, gobpf, redbpf, rust-bcc), providing usage examples and feature summaries. Although insightful, these works lack a quantitative efficiency evaluation of the libraries examined.

Our work addresses this gap by identifying and researching three key challenges that have not been explored in the eBPF library ecosystem. First, when developers choose a library based solely on functional characteristics, such as supported languages, portability, and ease of use, it can lead to degraded system performance and efficiency. The magnitude of this degradation and its relationship to the workload characteristics remain poorly understood.

Second, efficiency cannot be assessed solely by performance metrics. As shown in §3, a library with seemingly low performance overhead may achieve this through inefficient resource usage and/or

[1] For simplicity, we use the term "*(eBPF) libraries*" throughout the paper to refer collectively to eBPF libraries and/or toolkits.

reduced data collection fidelity (*i.e.*, increased event loss), potentially compromising the reliability of eBPF-based solutions. Understanding these trade-offs is critical for informed decision-making.

Third, designing an evaluation framework for comparing eBPF libraries across multiple efficiency dimensions is not trivial due to distinct interfaces, complex interactions between user space and kernel components, and other factors.

Contributions. This paper is a first step towards the design of such a framework by performing the first systematic efficiency analysis of popular eBPF libraries, while addressing the following research questions:

(1) **Performance Impact:** How do different libraries affect application performance in terms of throughput, latency, and runtime overhead?
(2) **Resource Efficiency:** Which libraries are the most efficient in terms of computing resources and energy?
(3) **Fidelity:** How do different tools compare in their ability to accurately capture events without loss?
(4) **Library Trade-offs:** Are there correlations across these dimensions that reveal fundamental trade-offs in the usage of eBPF libraries?

We evaluate five widely-used eBPF libraries (bpftrace, BCC, libbpf, ebpf-go, and Aya) using three representative I/O tracing tools. In this preliminary study, we focus on storage I/O profiling, a common and well-understood eBPF use case [9, 21, 28], while designing controlled, reproducible experiments that highlight key differences among libraries.

Our results show that no single library consistently outperforms the others across all dimensions. For instance, while libraries like Aya and libbpf can collect more events, especially for larger payloads, this comes with a performance penalty under certain workloads. Similarly, the choice of user space programming language and the approach used to interact with kernel code can substantially impact both performance and resource consumption.

Rather than attempting to rank these libraries definitively, our goal is to provide developers with quantitative insights into their inherent trade-offs, allowing for more informed decisions based on the specific application requirements. By quantifying these differences, we aim to contribute to the ongoing evolution of the eBPF ecosystem and encourage more efficiency-oriented design and implementation decisions.

Ethics: *This work raises no ethical concerns.*

2 Methodology

This section outlines the selected eBPF libraries, the tools and workloads used, and the experimental setup.

eBPF Libraries. We selected five widely used libraries, whose main characteristics are depicted in Table 1. BCC [30] (v0.33) and bpftrace [27] (v0.22.1) are two of the earliest libraries that are still actively used and maintained. BCC provides Python bindings for eBPF development, while bpftrace offers a high-level scripting language designed for tracing tasks. libbpf [15] (v1.5.0), ebpf-go [6] (v0.17.3) and Aya [2] (v0.13.1) are more recent libraries with support for *Compile Once–Run Everywhere (CO-RE)*, enhancing portability across kernel versions. libbpf is a C library used as the foundation for other eBPF libraries, while ebpf-go provides Go bindings. Aya allows writing both user space and kernel eBPF programs in Rust, with support for asynchronous user space code using runtimes like *Tokio* and *Async-std*.

Table 1: Characteristics of studied eBPF libraries.

Library	User space Language	Kernel space Language	CO-RE Support
BCC	Python/Lua/C++	C	✗
bpftrace	Scripting language		✗
libbpf	C/C++	C	✓
ebpf-go	Go	C	✓
Aya	Rust	C / Rust	✓

The experiments consider an isolated testing setup for each library, namely, `BCC`, `bpftrace`, `libbpf`, `ebpf-go`, and `AyaSync`. For the latter, we also consider an alternative setup (`AyaAsync`) that uses the *Tokio* asynchronous runtime. We compare these against a `vanilla` setup not using eBPF.

Tools. For each of the selected libraries, we implemented three eBPF-based tools: *syscount*, *rw-tracer* and *rw-tracer-all*.

syscount is a lightweight tool designed to count system call events in the kernel. It is inspired by the well-known syscount performance analysis tool [4], which has been previously implemented using various eBPF libraries (*e.g.*, `bpftrace`, `BCC`). The tool instruments the *raw_syscalls* tracepoint and employs eBPF maps of type *BPF_HASH* to record and expose the number of occurrences per syscall type (*e.g.*, open, write) to user space. Since the aggregated statistics are processed only once in user space (*i.e.*, upon tool termination), it requires minimal kernel-to-user data transfer.

rw-tracer targets a common eBPF use case: tracing storage I/O operations. It intercepts *read* and *write* calls at the Virtual File System layer via *vfs_read* and *vfs_write* kernel probes, capturing detailed event information including timestamps, arguments (excluding the data buffers), and return values. Each event, with ≈156 bytes, is immediately transferred to user space through an eBPF ring buffer configured with a size of 256 KiB and a polling timeout set to 100 ms (default values used by `bpftrace`). eBPF maps are used for sharing data between *entry* and *exit* probes and for counting the number of intercepted and lost events. This tool is more intensive than *syscount* in terms of kernel-to-user data transfer.

rw-tracer-all is a variant of *rw-tracer* that captures up to 4 KiB of *read* and *write* operations' data buffer content, increasing events' size to ≈4252 bytes. eBPF ring buffer and map configurations are the same as those used in *rw-tracer*.

Each tool was implemented across all five libraries, maintaining the functional equivalence of the implementations as closely as possible.[2]. This involved using consistent configuration settings across libraries and simple, single-threaded user space code to reduce behavioral variability.

Since efficient epoll-based ring buffer polling in Aya is natively supported only for asynchronous user space code (*i.e.*, using the

[2]Code, scripts, and experimental results are available at https://github.com/dsrhaslab/ebpf-lib-eval

Tokio runtime combined with the *AsyncFd* interface), for comparison purposes, we implemented a basic synchronous version that actively polls from the ring buffer. Throughout the paper, these two versions are referred to as `AyaAsync` and `AyaSync`, respectively.

Workload Generation. We used the Flexible I/O Tester (FIO), a widely adopted I/O benchmarking tool, to generate consistent and reproducible stress-based I/O workloads [1]. All FIO workloads were configured with four concurrent processes, each writing sequentially 32 GiB of data with a block size of 4 KiB. To assess how the libraries perform under different I/O operations, we employed three types of workloads: *(i) read-only*, *(ii) write-only*, and *(iii) mixed* (with 50/50 distribution of reads and writes).

The eBPF tools were configured to intercept only I/O events generated by the FIO benchmark.

Experimental Setup and Metrics. To ensure isolated testing environments and avoid dependency conflicts among libraries, the experiments were conducted on five identical servers (one for each library) with an Intel® Core™ i5-9500 CPU @ 3.00 GHz with 6 cores, 16 GiB of RAM, a 500 GiB SATA HDD, and a 240 GiB NVMe SSD. All servers ran Ubuntu 24.04 with kernel version 6.8.0-58-generic.

CPU and RAM metrics were monitored using `dstat` [12], while energy consumption was measured with Intel `RAPL` [13]. We carefully controlled measurement interference by pinning monitoring tools (*i.e.*, `dstat` and `RAPL`) to CPU core 0, leaving cores 1 to 5 pinned for FIO and the eBPF tools. FIO was configured to use the NVMe SSD, while monitoring logs were stored on the SATA HDD. The output of the three tools gathered in user space (*i.e.*, syscall statistics, events' information) was redirected to a file on the SSD.

From FIO, we collected runtime, I/O throughput, and I/O latency metrics. From the eBPF tools, we recorded the number of intercepted and lost events. Each experiment was repeated three times, and we report the average and standard deviation for each metric. To identify subtle differences across servers, despite them being identical, we compared performance and resource usage measurements across servers under the `vanilla` setup. In the results, we report the `vanilla` standard deviation (shaded blue area in the plots) and consider any deviations that fall within that range inconclusive.

3 Experimental Results

This section presents experimental results and key observations organized by workload type. A broader discussion, including the main takeaways, is deferred to §4.

3.1 Read-only Workload

Fig. 1 shows the results for the *read-only* workload, reporting runtime ((a)-(d)), throughput ((e)-(h)), and latency ((i)-(l)). Fig. 2 shows the number of intercepted and lost events for the *rw-tracer* and *rw-tracer-all* tools, as both rely on a ring buffer that can experience event loss. Fig. 3 presents CPU (user + system), memory (used + buffers), and energy consumption.

Performance Impact. The `vanilla` setup runs under 77.73 s ± 0.14, with a throughput of 1686.13 MiB/s ± 2.94 and an average latency of 9.07 ms ± 0.02. All setups exhibit performance close to `vanilla`, though `libbpf` introduces the highest overhead among both the *syscount* and *rw-tracer* tools, with increases around 0.43% and 2.18%, respectively. In *rw-tracer-all* case, the highest performance overhead is imposed by `AyaSync`, ≈2.39% over `vanilla`.

Lost Events. The benchmark generates a fixed amount of ≈33M events. Overall, *rw-tracer* captures more events than *rw-tracer-all*. Among the libraries, `libbpf` performs best, capturing all events, followed by Aya variants at 98% (≈32.8M). `BCC` and `ebpf-go` capture 47.45% (≈15.9M) and 36.95% (≈12.4M), respectively, while `bpftrace` records only 13.64% (≈4.6M).

For *rw-tracer-all*, `ebpf-go` and `BCC` perform significantly worse, collecting 0.03% (≈10k) and 0.07% (≈22.6k) of events, respectively, both falling behind `bpftrace`, which captures 0.24% (≈80k). `AyaSync` and `AyaAsync` capture the most events, with 0.72% (≈241.9k), followed by `libbpf` with 0.66% (≈221.2k).

Resource Usage. For *syscount*, all libraries show CPU usage close to `vanilla` (23.98% ± 0.10), with a slight increase up to 1.10× with `AyaSync`. In *rw-tracer* and *rw-tracer-all*, usage nearly doubles (1.79×–2.35×), with `libbpf` being most efficient and `bpftrace` the most demanding.

Memory usage remains close to `vanilla` (311.36 MiB ± 23.93) for all tools. *Syscount* shows little variation (up to 1.19× with `BCC`), while *rw-tracer* and *rw-tracer-all* see moderate increases (up to 1.42× and 1.31×). `BCC` and `ebpf-go` use the most memory, while `bpftrace` and `libbpf` use the least.

Energy consumption follows CPU trends. *Syscount* stays close to `vanilla` (12.82 W ± 0.70), peaking at 1.17× (`BCC`). For the other tools, it rises up to 1.72×, with `bpftrace` and `BCC` showing the highest consumption, and `ebpf-go` the lowest.

Summary. In read-only workloads, all setups show similar performance with minimal overhead. `libbpf`, `AyaAsync`, and `AyaSync` achieve the highest fidelity under *rw-tracer* (>97%), but all tools perform poorly under *rw-tracer-all* (<1%). `bpftrace` is the most CPU-intensive, `BCC` and `ebpf-go` use the largest amount of memory, and `BCC` usually leads in energy consumption.

3.2 Write-only Workload

Fig. 4 shows performance metrics for the *write-only* workload, Fig. 5 reports the number of intercepted and lost events, and Fig. 6 presents CPU, memory, and energy usage.

Performance Impact. As expected with NVMe SSDs' lower write bandwidth, `vanilla` shows longer runtime (319.43 s ± 21.66), lower throughput (412.53 MiB/s ± 24.71), and higher latency (37.79 ms ± 2.58) in the *write-only* workload. For *syscount*, `bpftrace` degrades throughput by ≈16% and increases runtime and latency by ≈19%. `BCC` shows similar impact under *rw-tracer*. For `rw-tracer-all`, all setups but `BCC` show significant degradation. Throughput drops up to 31% (`AyaSync`), with runtime and latency overheads reaching ≈46%. `libbpf` and `AyaAsync` also exceed 20% overhead.

Lost Events. With *rw-tracer*, most libraries capture the majority of events. `libbpf` leads with 99%, followed by Aya variants at ≈96%. The lower I/O throughput in the *write-only* workload improves capture rates for `BCC` and `ebpf-go` (≈94%), though `bpftrace` still trails at 52.60% (≈17.7M).

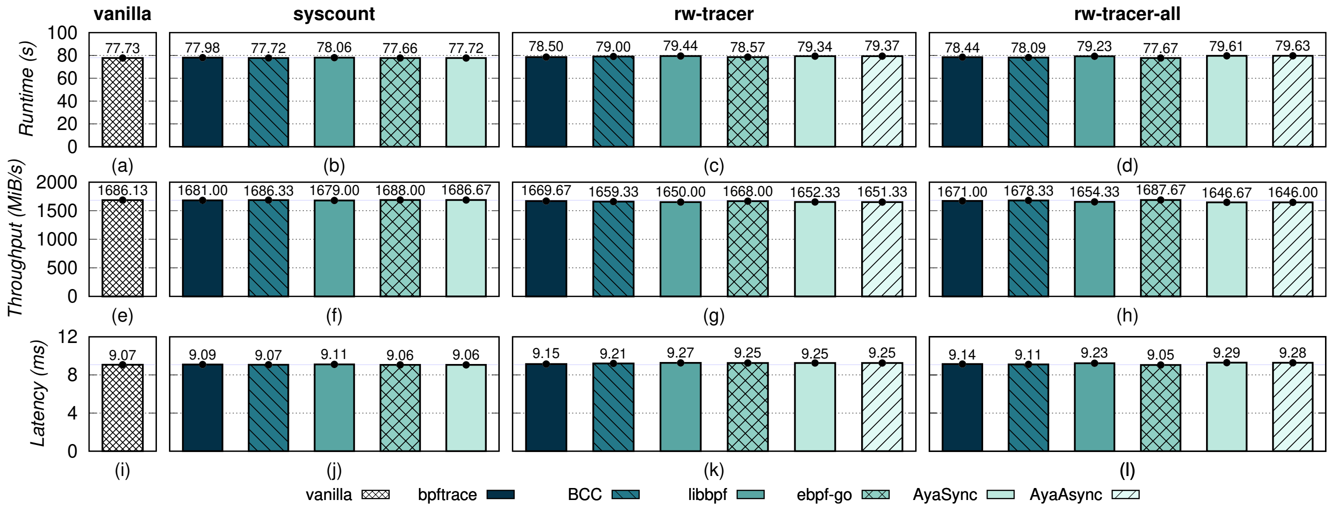

Figure 1: Runtime, throughput, and latency for the *read-only* workload, broken down by tool and setup.

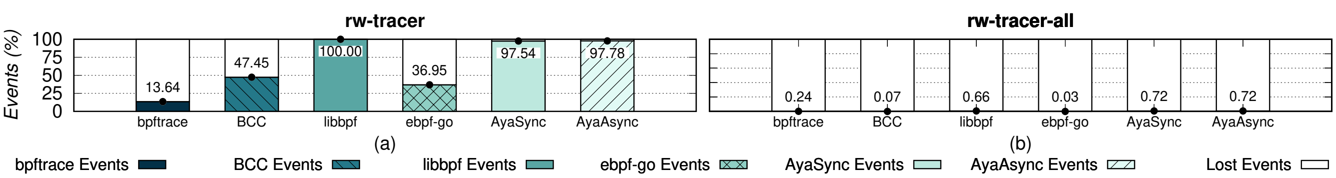

Figure 2: Percentage of saved and lost events for the *read-only* workload, broken down by tool and setup.

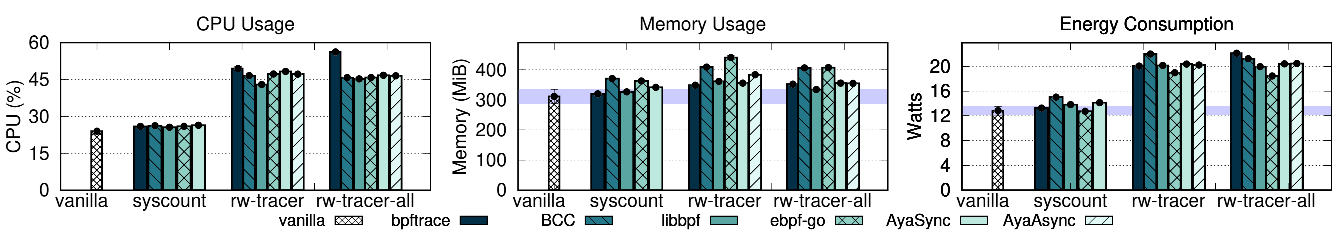

Figure 3: Resource usage during the *read-only* workload, broken down by tool and setup.

For *rw-tracer-all*, all libraries lose over 96% of events. Aya variants perform best (≈4%, ≈1.3M), followed by `libbpf` at 3.26%. The remaining libraries fall below 1.11% (≈372.7k).

Resource Usage. CPU usage in this workload is notably lower, with `vanilla` consuming 7.02% ± 0.50, reflecting reduced throughput. Under *syscount*, CPU usage stays close to `vanilla`, with `libbpf` and `AyaSync` reaching 1.07×. In contrast, *rw-tracer* significantly increases CPU usage, from 1.84× with `libbpf` to ≈3.64× with `bpftrace` and `AyaSync`. *Rw-tracer-all* raises it further, with all setups exceeding a 3× overhead, ranging from 3.09× (`AyaSync`) to 3.63× (`bpftrace`).

Memory usage exceeds that of the *read-only* workload, with `vanilla` using 378.53 MiB ± 22.71. `BCC` and `ebpf-go` consume the most (up to 1.27× in *rw-tracer*), while `bpftrace` is the most efficient under *syscount* (1.02×) and `libbpf` under the other tools (≈1.06×).

Energy consumption under *syscount* stays near `vanilla` (6.03 W ± 0.54), with a maximum increase of 1.11× (`AyaSync`, `libbpf` and `BCC`). *Rw-tracer* and *rw-tracer-all* cause larger increases, peaking at 2.02× (`AyaSync`) and 2.22× (`BCC`). `ebpf-go` remains the most energy-efficient overall.

Summary. With write-only workload, performance differences became more pronounced. Most setups perform similarly under *syscount* and *rw-tracer*, though `bpftrace` and `BCC` show higher overhead and variability. Under *rw-tracer-all*, `AyaSync`, `libbpf`, and `AyaAsync` show greater overhead. Fidelity improves, with all setups except `bpftrace` capturing over 94% of events under *rw-tracer*, and with `AyaSync`, `libbpf`, and `AyaAsync` reaching nearly 4% under *rw-tracer-all*. Resource-wise, `bpftrace` and `AyaSync` are among the most CPU- and energy-intensive, while `BCC` and `ebpf-go` remain the heaviest on memory consumption.

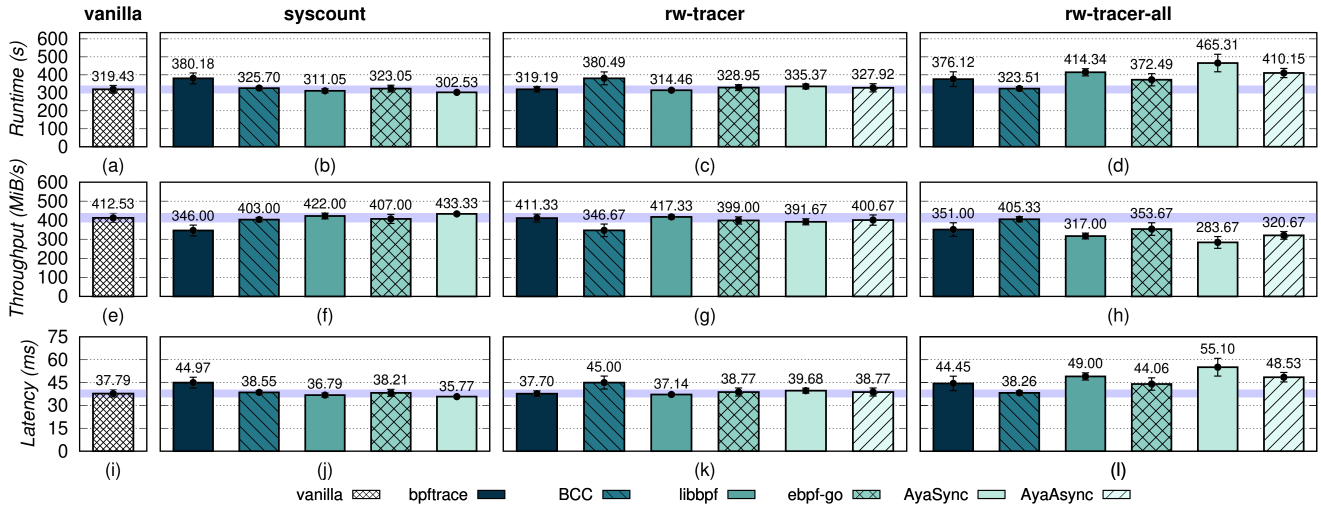

Figure 4: Runtime, throughput, and latency for the *write-only* workload, broken down by tool and setup.

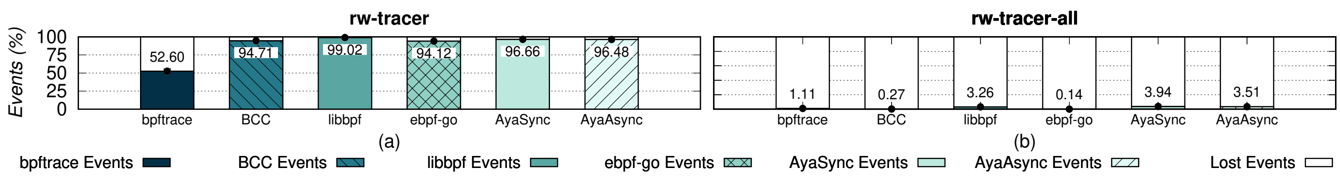

Figure 5: Percentage of saved and lost events for the *write-only* workload, broken down by tool and setup.

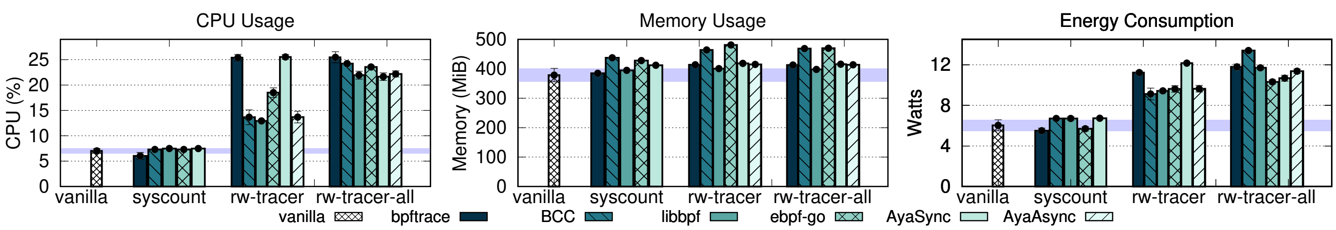

Figure 6: Resource usage during the *write-only* workload, broken down by tool and setup.

3.3 50-50 Mixed Read-Write Workload

Due to space constraints, we omit the results for the *mixed* workload from the paper. [3] Performance-wise, all libraries are close to `vanilla`, with a slight overhead noticeable in the *rw-tracer* and *rw-tracer-all* tools for `libbpf`, `AyaSync`, and `AyaAsync`. Remarkably, these three libraries capture the highest number of events.

CPU usage follows a pattern between the *write-only* and *read-only* workloads, with `bpftrace` and `AyaSync` exhibiting the highest overhead in *rw-tracer*. `BCC` and `ebpf-go` show the highest memory consumption, with the former also recording the highest energy usage and the latter consistently being more energy-efficient.

Summary. Performance, fidelity, and resource usage reflect a combination of the behaviors observed in *read-only* and *write-only* workloads, *i.e.*, the *mixed* workload results fall in between the tendencies observed for the other two.

4 Discussion

Our evaluation reveals a complex interplay between workload characteristics, tool design, and the eBPF libraries used to implement it. The results highlight important trade-offs and design implications for both developers and practitioners aiming to deploy eBPF tools in production environments.

Fidelity vs. Performance. Our findings show an inherent tension between fidelity and performance impact, driven mainly by two factors: workload intensity and the data volume processed in user space. In all experiments, `libbpf`, `AyaSync`, and `AyaAsync` consistently save the most events but incur the greatest performance overhead in the *rw-tracer* and *rw-tracer-all* tools.

[3] Mixed experiments results are available at https://github.com/dsrhaslab/ebpf-lib-eval/blob/main/docs/results.md

In less intensive workloads (*write-only*), event size is a key factor. Smaller events (*rw-tracer*) allow libraries to process more data from the ring buffer, enhancing fidelity. Conversely, larger events (*rw-tracer-all*) incur greater processing costs (*e.g.*, writing to disk), leading to higher performance overhead and increased event loss. In more intensive workloads (*read-only*), the link between fidelity and performance overhead becomes less obvious. The rapid generation of events quickly fills up the ring buffer, leading to high event loss. In turn, this leads to fewer events reaching user space and consequently to a negligible performance overhead.

These findings confirm that focusing on a single axis may overlook other factors that can critically impact production deployments. Applications requiring high-fidelity monitoring, like compliance auditing or distributed tracing, may need to accept a loss in performance, while performance-critical applications might need to implement sampling strategies or accept some degree of event loss.

Programming Language vs Fidelity and Resource Usage. The choice of user space programming language, with its inherent runtime characteristics (*i.e.*, speed, resource footprint), may have a different impact on fidelity and resource usage. `bpftrace`, `BCC`, and `ebpf-go` exhibit higher event loss rates than other setups, even with smaller payloads like those in *rw-tracer*, which is likely caused by how quickly the user space code processes events from the ring buffer. Further, `BCC` and `ebpf-go` show consistently higher memory consumption, noticeable even before the workloads start.

These findings point out that the development convenience of higher-level language abstractions comes with a measurable cost, which may be significant in resource-constrained or high-throughput environments.

Polling Strategy vs Resource Usage. Our results show that the strategies used for exchanging data between kernel and user space affect CPU usage. Active polling implementations such as *AyaSync* show higher CPU consumption, especially under less intensive workloads like *write-only*, as opposed to the other setups (including `AyaAsync`) that leverage a more efficient `epoll`-based polling strategy. As expected, this also results in increased energy consumption, especially for tools like *rw-tracer-all*, which involve frequent polling and substantial kernel-to-user space data exchange.

`bpftrace` internally uses the *bpf_ringbuf_output* helper, which performs implicit and costly data copies from kernel memory to the ring buffer. This contrasts with the zero-copy *reserve/commit* strategy employed by other setups, and contributes directly to its higher CPU overhead.

These results underscore the importance of selecting appropriate data exchange strategies with user space (*e.g.*, active vs. epoll-based polling, copy-based vs. zero-copy event insertion into ring buffers), as naive configurations may lead to unnecessary CPU consumption, particularly under less intensive workloads or idle periods.

Limitations and Future work. While this study opens an interesting new path for a comprehensive evaluation framework of eBPF libraries, it also reveals that other key aspects, and even some observed results, must be further understood.

For instance, in the *write-only* workload, `bpftrace` and `BCC` show the highest performance overhead for the *syscount* and *rw-tracer* tools, respectively, but also exhibit high variability. Further profiling would be important for understanding exactly why this happens. Moreover, isolating and analyzing the kernel and user space components separately, both in terms of performance and event processing rate, would help eliminate certain sources of variation and provide deeper insight into the impact of each eBPF library.

Additionally, our experiments were conducted with fixed eBPF configurations (*e.g.*, ring buffer size, polling timeout). It would be interesting to explore how tuning these parameters affects fidelity, resource usage, and performance, and whether it could bring different libraries closer in behavior.

It would also be valuable to expand the evaluation to other domains, like network or security. Many eBPF use cases in these areas are not purely observational, as in tracing, but also actuate. For example, eBPF is often used to redirect network traffic for load balancing or to drop unauthorized packets for security enforcement [5, 19, 32]. In such scenarios, much of the work is performed entirely within the kernel, which raises an important question: does the choice of user space eBPF library still have a meaningful impact in these kernel-centric use cases, or is its influence diminished compared to more user-intensive workflows like tracing?

Another important direction would be to investigate how the conclusions drawn in this paper hold when testing with more complex eBPF-based tools. Real-world eBPF applications tend to have a greater code complexity than the tools tested for this study. Moreover, these eBPF applications often operate in concurrent environments (*e.g.*, by leveraging mechanisms like goroutines in Go or multi-threaded event handling in C/C++). Would the performance, resource usage, and fidelity impact differences between libraries remain consistent, or would concurrency mechanisms amplify or mask them? Exploring larger applications targeting different kernel layers, interacting more extensively with user space, and leveraging these concurrent and asynchronous mechanisms can help highlight how the eBPF libraries perform under complex use cases.

Lastly, while our current evaluation uses controlled and repeatable stress-test workloads, it remains an open question how each library performs under real-world conditions, where workloads can exhibit bursts, halts, and shifting intensity. It would be valuable to assess whether differences across libraries narrow in such settings due to amortized overheads, or instead widen because of different buffering, polling, or synchronization strategies.

Acknowledgments

We thank the anonymous reviewers for their insightful feedback. This work is funded by national funds through FCT – Fundação para a Ciência e a Tecnologia, I.P., under the support UID/50014/2023 (https://doi.org/10.54499/UID/50014/2023) *(Carlos Machado and Tânia Esteves)*, grant PTDC/CCI-COM/4485/2021, plurianual grant UIDB/50021/2020, and co-funded by the European Regional Development Fund (ERDF) through the NORTE 2030 Regional Programme under Portugal 2030, within the scope of the project BCDSM, reference 14436 (NORTE2030-FEDER-00584600) *(João Paulo)*.

References

[1] Jens Axboe. 2006. Flexible I/O Tester. (2006). Retrieved July, 2025 from https://github.com/axboe/fio

[2] Aya 2021. Aya: An eBPF library for the Rust programming language. (2021). Retrieved July, 2025 from https://github.com/aya-rs/aya

[3] Eunomia bpf organization. 2022. eunomia-bpf: dynamic loading library/runtime and a compile toolchain framework. (2022). Retrieved July, 2025 from https://github.com/eunomia-bpf/eunomia-bpf
[4] Canonical. 2019. Ubuntu Manpage: syscount - count system calls. (2019). Retrieved July, 2025 from https://manpages.ubuntu.com/manpages/jammy/man8/syscount-perf.8.html
[5] Cilium Community. 2015. Cilium: Networking, observability, and security solution with an eBPF-based dataplane. (2015). Retrieved July, 2025 from https://github.com/cilium/cilium
[6] Cilium Community. 2017. ebpf-go: Go library for working with eBPF. (2017). Retrieved July, 2025 from https://github.com/cilium/ebpf
[7] Cilium Community. 2022. Tetragon: Powerful real-time, eBPF-based Security Observability and Runtime Enforcement. (2022). Retrieved July, 2025 from https://github.com/cilium/tetragon
[8] Milo Craun, Khizar Hussain, Uddhav Gautam, Zhengjie Ji, Tanuj Rao, and Dan Williams. 2024. Eliminating eBPF Tracing Overhead on Untraced Processes. In *Proceedings of the ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions*. Association for Computing Machinery, New York, NY, USA, 16–22. https://doi.org/10.1145/3672197.3673431
[9] Tânia Esteves, Ricardo Macedo, Rui Oliveira, and João Paulo. 2023. Diagnosing applications' I/O behavior through system call observability. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE Computer Society, Washington, DC, USA, 1–8. https://doi.org/10.1109/DSN-W58399.2023.00022
[10] William Findlay, Anil Somayaji, and David Barrera. 2020. bpfbox: Simple Precise Process Confinement with eBPF. In *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop (CCSW'20)*. Association for Computing Machinery, New York, NY, USA, 91–103. https://doi.org/10.1145/3411495.3421358
[11] Brendan Gregg. 2019. A thorough introduction to bpftrace. (2019). Retrieved July, 2025 from https://www.brendangregg.com/blog/2019-08-19/bpftrace.html
[12] Red Hat. 2007. pcp-dstat. (2007). Retrieved July, 2025 from https://github.com/performancecopilot/pcp
[13] Intel. [n. d.]. RAPL: Running Average Power Limit. ([n. d.]). Retrieved July, 2025 from https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html
[14] Jung-Bok Lee, Tae-Hee Yoo, Eo-Hyung Lee, Byeong-Ha Hwang, Sung-Won Ahn, and Choong-Hee Cho. 2021. High-Performance Software Load Balancer for Cloud-Native Architecture. *IEEE Access* 9 (2021), 123704–123716. https://doi.org/10.1109/ACCESS.2021.3108801
[15] Libbpf 2018. Automated upstream mirror for libbpf stand-alone build. (2018). Retrieved July, 2025 from https://github.com/libbpf/libbpf
[16] Libbpf-rs 2020. libbpf-rs: Idiomatic Rust wrapper around libbpf. (2020). Retrieved July, 2025 from https://github.com/libbpf/libbpf-rs
[17] Libbpfgo 2020. libbpfgo: Go wrapper around the libbpf project. (2020). Retrieved July, 2025 from https://github.com/aquasecurity/libbpfgo
[18] libxdp 2019. libxdp: XDP-specific library that sits on top of libbpf and implements a couple of XDP features. (2019). Retrieved July, 2025 from https://github.com/xdp-project/xdp-tools
[19] Meta. 2018. High-performance layer 4 load balancing forwarding plane. (2018). Retrieved July, 2025 from https://github.com/facebookincubator/katran
[20] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. 2018. Creating Complex Network Services with eBPF: Experience and Lessons Learned. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE Computer Society, Washington, DC, USA, 1–8. https://doi.org/10.1109/HPSR.2018.8850758
[21] Mohammed Islam Naas, François Trahay, Alexis Colin, Pierre Olivier, Stéphane Rubini, Frank Singhoff, and Jalil Boukhobza. 2021. EZIOTracer: unifying kernel and user space I/O tracing for data-intensive applications. In *Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems (CHEOPS '21)*. Association for Computing Machinery, New York, NY, USA, Article 4, 11 pages. https://doi.org/10.1145/3439839.3458731
[22] Jaehyun Nam, Seungsoo Lee, Phillip Porras, Vinod Yegneswaran, and Seungwon Shin. 2023. Secure Inter-Container Communications Using XDP/eBPF. *IEEE/ACM Transactions on Networking* 31, 2 (2023), 934–947. https://doi.org/10.1109/TNET.2022.3206781
[23] Teodor J. Podobnik. 2025. Go, C, Rust, and More: Picking the Right eBPF Application Stack. (2025). Retrieved July, 2025 from https://ebpfchirp.substack.com/p/go-c-rust-and-more-picking-the-right?utm_source=substack&publication_id=2062956&post_id=154206053&utm_medium=email&utm_content=share&utm_campaign=email-share&triggerShare=true&isFreemail=true&r=4mi1za&triedRedirect=true
[24] Sanjeev Rampal. 2023. eBPF application development: Beyond the basics. (2023). Retrieved July, 2025 from https://developers.redhat.com/articles/2023/10/19/ebpf-application-development-beyond-basics#ebpf_application_cross_development__portability__co_re__and_kernel_api_stability
[25] Liz Rice. 2023. *Learning eBPF: Programming the Linux Kernel for Enhanced Observability, Networking, and Security* (first edition ed.). O'Reilly, Beijing Boston Farnham Sebastopol Tokyo.
[26] Alessandro Rivitti, Roberto Bifulco, Angelo Tulumello, Marco Bonola, and Salvatore Pontarelli. 2023. eHDL: Turning eBPF/XDP Programs into Hardware Designs for the NIC. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 208–223. https://doi.org/10.1145/3582016.3582035
[27] Alastair Robertson. 2016. bpftrace: High-level tracing language for Linux. (2016). Retrieved July, 2025 from https://github.com/bpftrace/bpftrace
[28] Husain Sharaf, Imtiaz Ahmad, and Tassos Dimitriou. 2022. Extended Berkeley Packet Filter: An Application Perspective. *IEEE Access* 10 (2022), 126370–126393. https://doi.org/10.1109/ACCESS.2022.3226269
[29] Polar Signals. 2021. Parca: Continuous profiling for analysis of CPU, memory usage over time, and down to the line number. (2021). Retrieved July, 2025 from https://github.com/parca-dev/parca
[30] IO visor. 2015. BCC: BPF Compiler Collection. (2015). Retrieved July, 2025 from https://github.com/iovisor/bcc/tree/master
[31] Tobias Waldekranz. 2015. ply: light-weight dynamic tracer for Linux. (2015). Retrieved July, 2025 from https://github.com/iovisor/ply
[32] Mathieu Xhonneux, Fabien Duchene, and Olivier Bonaventure. 2018. Leveraging eBPF for programmable network functions with IPv6 segment routing. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '18)*. Association for Computing Machinery, New York, NY, USA, 67–72. https://doi.org/10.1145/3281411.3281426
[33] Rui Yang and Marios Kogias. 2023. HEELS: A Host-Enabled eBPF-Based Load Balancing Scheme. In *Proceedings of the 1st Workshop on EBPF and Kernel Extensions (eBPF '23)*. Association for Computing Machinery, New York, NY, USA, 77–83. https://doi.org/10.1145/3609021.3609307
[34] Wanqi Yang, Pengfei Chen, Kai Liu, and Huxing Zhang. 2025. ZeroTracer: In-band eBPF-based Trace Generator with Zero Instrumentation for Microservice Systems. *IEEE Transactions on Parallel and Distributed Systems* 36, 7 (2025), 1478–1494. https://doi.org/10.1109/TPDS.2025.3571934