Dissecting Storage I/O Tracing Overhead of eBPF: A Component-Level Analysis

Mohammed Islam Naas
L3i, EA 2118,
La Rochelle University
France
mohammed.naas@univ-lr.fr

Pierre Olivier
The University of Manchester
United Kingdom
pierre.olivier@manchester.ac.uk

Stéphane Rubini
Lab-STICC, CNRS UMR 6285,
University of Western Brittany
France
Stephane.Rubini@univ-brest.fr

Frank Singhoff
Lab-STICC, CNRS UMR 6285,
University of Western Brittany
France
Frank.Singhoff@univ-brest.fr

Jalil Boukhobza
Lab-STICC, CNRS UMR 6285, ENSTA,
Institut Polytechnique de Paris
France
jalil.boukhobza@ensta.fr

Abstract

Extended Berkeley Packet Filter (eBPF) has become a key technology for low-overhead observability in Linux-based systems, enabling secure, dynamic and programmable tracing of kernel subsystems. However, when applied to intensive storage I/O workloads, the actual performance impact of eBPF tracing and the related cost of its internal components remain insufficiently characterized. In this paper, we present a component-level analysis of eBPF-based I/O tracing, focusing on the I/O execution path and proposing a decomposition of the tracing pipeline into its main cost factors. To quantify the cost of each stage, we introduce a controlled experimental framework that incrementally enables tracing components to isolate their individual overhead. Our results show that eBPF tracing overhead is highly workload-dependent, reaching up to 41% under small I/O requests with high concurrency. A component-level analysis identifies userspace event retrieval and probe triggering as the primary bottlenecks, responsible for up to 24% and 11% of the overall overhead, respectively.

CCS Concepts: • **Software and its engineering → Maintaining software**; **Software maintenance tools**; • **Information systems → Cloud based storage**.

Keywords: eBPF, Storage I/O, Linux Kernel Tracing, Performance Overhead, BCC, IOTracer.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CHEOPS '26, Edinburgh, Scotland Uk

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-2604-0/26/04
https://doi.org/10.1145/3805687.3806254

ACM Reference Format:
Mohammed Islam Naas, Pierre Olivier, Stéphane Rubini, Frank Singhoff, and Jalil Boukhobza. 2026. Dissecting Storage I/O Tracing Overhead of eBPF: A Component-Level Analysis. In *6th Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems (CHEOPS '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3805687.3806254

1 Introduction

Modern applications such as HPC, large-scale data analytics, and AI workloads increasingly rely on data-intensive pipelines and generate massive volumes of storage I/Os. As storage systems evolve in complexity, diagnosing performance anomalies and identifying runtime bottlenecks becomes a difficult and time-consuming task [18].

Tracing is an effective approach for analyzing storage performance [23]. By instrumenting critical points of the software stack, tracing enables the collection of fine-grained runtime information that can be leveraged to understand system behavior, identify performance bottlenecks, and characterize interactions between applications, the operating system, and storage hardware. Reducing the overhead introduced by tracing code is critical for several reasons. High-overhead tracers can overload the system by consuming CPU cycles, memory, or I/O bandwidth, which may in turn disturb the behavior of the monitored application and distort the measured performance metrics.

Among Linux tracing technologies [1–3, 8, 15, 22], extended Berkeley Packet Filter (eBPF) has gained significant adoption due to its flexibility and low deployment cost [10, 13, 18]. eBPF enables safe and dynamic kernel instrumentation without requiring kernel recompilation or the insertion of custom kernel modules. It provides access to multiple layers of the Linux I/O stack, ranging from system calls and virtual file system (VFS) operations to block I/O, making

it a powerful tool for observability, debugging, and performance monitoring [19]. Despite its popularity, the performance impact of eBPF tracing on high-throughput storage workloads remains insufficiently characterized. In particular, eBPF tracing relies on a multi-stage pipeline, including event triggering, in-kernel data collection, filtering, buffering, and eventual delivery to user space. Each stage may contribute differently to the overall overhead depending on workload characteristics such as request size, concurrency level, and event rate.

Several recent studies have investigated the eBPF ecosystem, including efforts to reduce tracing overhead as well as challenges related to usability, correctness, and security [6, 7, 17, 21, 24, 26]. However, the performance impact of eBPF tracing under high-throughput storage I/O workloads, and the relative cost of its internal components, remain insufficiently characterized.

In this work, we first propose a decomposition of the eBPF tracing pipeline into its main stages, including probe triggering, event filtering, ring buffer reservation, event construction, event committing, and user-space event retrieval. We then conduct a component-level experimental analysis to quantify the overhead induced by each stage when monitoring storage I/O workloads. We generate controlled I/O workloads using a user-level benchmarking tool (FIO), exploring multiple configurations in terms of request sizes and parallel job counts. Experiments are conducted on two storage backends: a Samsung SSD mSATA formatted with the ext4 file system, and a RAM Disk mounted on `/tmp` to represent an ultra-fast storage medium. A baseline execution without tracing is first established, after which eBPF tracing components are progressively enabled to isolate and measure their individual contributions.

The experimental results show that eBPF overhead is strongly workload-dependent and varies significantly with workload intensity, I/O size, and the underlying storage medium (SSD or RAM disk). Across all tested configurations, the throughput degradation ranges from negligible values (around 1–2%) for large I/O requests to severe penalties under small I/O sizes and high concurrency, reaching up to 38% on SSD and 41% on RAM disk. The component-level analysis further highlights that the most overhead-sensitive stages are probe triggering and, more critically, event retrieving (trace delivery to userspace), while event filtering, event building, and event committing contribute only marginally to the overall cost (typically below 5%). Overall, these results confirm that eBPF tracing overhead is primarily driven by high event rates, and that specific pipeline stages become dominant bottlenecks when tracing fine-grained I/O activity on high-performance storage devices. Based on these findings, we provide practical guidelines to reduce the intrusiveness of eBPF tracers.

The remainder of the paper is organized as follows. Section 2 describes the eBPF tracing architecture and details the role of each pipeline component. Section 3 presents the evaluation methodology and discusses the results. Section 4 summarizes the main findings and provides practical guidelines to reduce tracing intrusiveness. Section 5 discusses related work. Finally, Section 6 concludes the paper and outlines future research directions.

2 Ebpf Tracing Architecture and Component Roles

In this section, we first present the internal architecture of the eBPF framework, highlighting the role of each component and its main configuration parameters. Then, we decompose the tracing pipeline into a sequence of stages, describing the contribution of each stage to the overall tracing process.

2.1 Background: Internal Ebpf Architecture

Figure 1 provides a comprehensive view of the internal architecture of the eBPF framework and illustrates the interaction between userspace applications and kernel eBPF programs through the BCC front-end.

2.1.1 eBPF Program Development. Traditionally, writing eBPF programs required manually implementing tracing logic in eBPF assembly and using the kernel `bpf_asm` assembler, which was error-prone and limited programmability. With the introduction of the BPF Compiler Collection (BCC), developers can now write kernel-side probe handlers in C and manage the tracing workflow from userspace scripts (commonly in Python or Lua), covering the full application lifecycle including probe activation, event retrieval, and post-processing of collected data.

2.1.2 Compilation and Loading. The kernel-side tracing logic written in C is compiled into BPF bytecode using the LLVM Clang compiler. The bytecode is then loaded into the Linux kernel via the `bpf()` system call [5]. During loading,

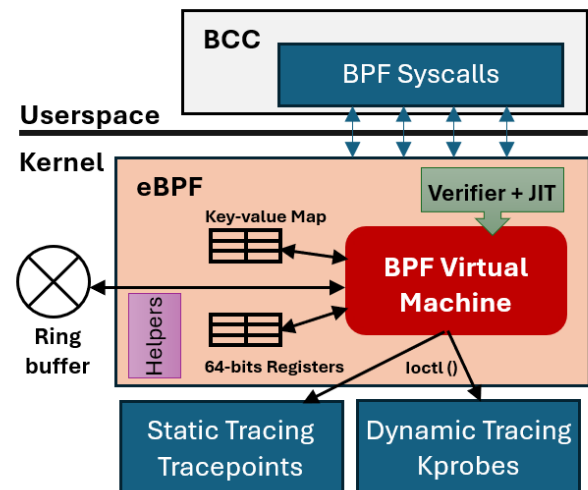


Figure 1. The eBPF internal architecture.

the eBPF verifier ensures program safety by checking for unbounded loops, invalid memory accesses, or unsafe kernel operations. After verification, the bytecode is translated into native machine instructions by the kernel JIT compiler, improving execution efficiency and reducing interpretation overhead [14].

2.1.3 Probe Activation and Execution. Once loaded, eBPF programs are attached to kernel execution points using dynamic probes or static tracepoints via helper functions such as `bpf_attach_*`. Probe activation occurs at runtime, typically through `ioctl()` system call, allowing flexible instrumentation without kernel recompilation. When a traced kernel function executes, the attached eBPF handler runs in the kernel context under strict execution constraints (limited stack, restricted instruction set, 10 registers). Helper functions allow controlled access to kernel resources, persistent state updates, and writing of trace records.

2.1.4 Data Storage and Retrieval. The output of eBPF programs is stored in two main kernel-resident structures:

- **Ring buffer:** holds tracing events for high-throughput asynchronous communication with userspace.
- **eBPF maps:** store statistical data such as counters, histograms, or latency distributions.

Userspace retrieves data via two mechanisms: consuming events from the ring buffer or accessing map contents through the `bpf()` syscall. These retrieval mechanisms represent a major configuration point, since user-space consumption speed directly impacts tracing performance, and may lead to event drops when the ring buffer becomes saturated.

In the following section, we decompose the tracing pipeline into several stages, highlighting their functional contributions to the tracing process.

2.2 Tracing Pipeline Decomposition

In this work, we consider the tracing pipeline as a sequential process composed of six main stages: probe triggering, event filtering, memory-space reservation, event building, event committing, and event retrieving. This decomposition is motivated by the fact that eBPF overhead is not caused by a single mechanism, but rather results from the accumulation of multiple operations executed along the critical path of the traced workload. Therefore, quantifying the contribution of each stage is essential to identify the dominant sources of intrusiveness and to better understand why certain workloads experience higher performance degradation than others. Each stage is evaluated independently by incrementally enabling or disabling specific tracing components.

2.2.1 Probe Triggering. The first stage corresponds to the triggering of the probe itself. When an instrumented kernel function is reached, the CPU execution flow is redirected to the probe handler, and the eBPF program execution context is initialized. This stage is entirely independent of the tracing logic and occurs even if the probe handler performs no processing. In order to isolate this overhead, the probe handler is configured as a minimal program that immediately returns without accessing any kernel structure, map, or buffer. This setup makes it possible to measure the pure overhead induced by probe triggering, including context switching to the eBPF execution engine.

2.2.2 Event Filtering. The second stage corresponds to the execution of filtering conditions within the probe handler. Filtering is commonly used in eBPF tracers, and it is executed every time the probe is triggered to reduce the number of traced events and avoid collecting irrelevant information. In our case, we apply a PID-based filtering condition that checks whether the current process corresponds to the traced workload. This configuration measures the combined cost of probe triggering and filtering execution. Since probe triggering is measured separately, the additional overhead introduced by filtering can be isolated by subtraction.

2.2.3 Memory-space Reservation. Once an event is considered relevant, the tracer must allocate a memory region to store it before writing the payload. With modern Linux kernels and BCC-based tracers, this is typically performed using the ring buffer reservation API (`bpf_ringbuf_reserve()`). This stage allocates a contiguous memory slot in the ring buffer for the event and prepares the kernel-side metadata required for committing the record. To evaluate this overhead independently, we configure the probe handler to perform filtering and then reserve space in the ring buffer, but return immediately without writing any event data. This enables us to quantify the additional overhead induced by memory reservation, isolated from subsequent stages.

2.2.4 Event Building. Event building refers to the construction of the trace record, i.e., collecting and preparing the event payload fields. This includes accessing kernel structures and extracting I/O metadata required for post-processing. In our evaluation, we consider a single realistic event structure, denoted *Event_building*, which contains the following fields: *timestamp, LBA, size, level, PID, TID, inode* (48 bytes total). This event format reflects a practical storage I/O tracing scenario where multiple attributes are required for performance diagnosis [18]. To isolate the overhead of event building, PID filtering is kept enabled to restrict tracing to the target workload. However, ring buffer reservation and submission are disabled in this stage, so that the probe handler only constructs the event without storing it in the `Ring buffer`. This configuration allows quantifying the overhead of event data extraction and preparation independently from memory allocation and kernel-to-user communication.

2.2.5 Event Committing. After the event payload has been built and the reserved memory slot has been filled, the

record must be committed into the ring buffer. This step is performed using `bpf_ringbuf_submit()` API, which makes the event visible to the userspace consumer. Event committing finalizes the record and updates the ring buffer internal state. To measure this overhead, we enable event reservation and submission, while disabling userspace retrieval. This configuration ensures that events are committed in the ring buffer, but are not consumed, allowing us to isolate the overhead of committing operations. The additional cost of this stage is obtained by subtracting the overheads of previous stages.

2.2.6 Event Retrieving.

The final stage corresponds to the retrieval of committed events from the ring buffer into userspace. This includes waking up the user-space consumer thread, copying event data from kernel memory to user memory, decoding raw structures, and optionally formatting or writing events to disk. In practice, this stage can dominate the overall overhead when the event rate becomes very high, since it involves kernel-to-user transitions and user-space processing costs. Unlike kernel-side stages, event retrieving overhead cannot be measured in strict isolation, since it depends on the implementation strategy of the user-space program. However, since all kernel-side stages are quantified independently, the remaining overhead observed in the end-to-end tracing configuration primarily reflects the cost of event retrieval and userspace event processing.

Overall, this decomposition provides a structured methodology for analyzing the intrusiveness of eBPF tracing and enables identifying the most overhead-sensitive mechanisms under high-throughput workloads.

3 Measuring the Intrusiveness of Each Ebpf Component

In this section, we experimentally evaluate the overhead induced by each tracing stage defined in Section 2.2. Our objective is to quantify the intrusiveness of each stage separately and identify the most overhead-sensitive components under high-throughput storage workloads. We first describe the experimental platform and benchmark setup, then present and discuss the obtained results.

3.1 Experimental Platform and Setup

To assess the intrusiveness of eBPF tracing, we rely on controlled micro-benchmarks generated using `FIO` [4]. The tracing process is performed using `IOTracer` [18], an eBPF-based storage tracing tool. In our experiments, we activate only two probes, `vfs_read` and `vfs_write`, in order to trace read and write operations at the VFS level.

The benchmark is configured to generate a mixed read/write workload (50% read, 50% write) using the `sync` I/O engine. To evaluate the scalability of the tracing pipeline under different workload intensities, we vary the following parameters:

- **Concurrency:** We use 1, 8, and 16 concurrent jobs to emulate increasing levels of contention and event generation rates.
- **I/O block size:** We test 4 KB, 64 KB, and 256 KB request sizes. This range covers both IOPS-intensive workloads (small blocks) and throughput-oriented workloads (large blocks).

In addition, to study the impact of both the number and the location of probes on tracing overhead, we considered two complementary scenarios. For this additional study, we fix the workload configuration to 8 concurrent jobs and an I/O request size of 64 KB in order to maintain a stable and representative event generation rate while isolating the effect of the tracing configuration.

- Case 1 - Multi-level Tracing: We placed probes at different levels of the I/O stack along the same execution path, ensuring that all probes are triggered for each I/O request. We evaluated configurations with 2, 4, and 8 probes distributed across different I/O layers, including the VFS, filesystem, page cache, and block layer. Each layer contributes two probes in the 8-probe configuration. For the 4-probe setup, probes are placed at the VFS and filesystem levels, while in the 2-probe configuration, only VFS-level probes were used.
- Case 2 - VFS-Only Tracing: We evaluate the impact of the presence of probes independently from their activation. In this scenario, all probes were placed at the same level (VFS), but only a subset of them is effectively triggered by the workload. More specifically, we configured 2, 4, and 8 probes at the VFS level, while only two probes (`vfs_read` and `vfs_write`) are actively triggered. The remaining probes correspond to operations not used by the FIO workload (e.g., symbolic link creation or directory operations), and therefore remain idle. This setup allows to isolate the effect of probe activation frequency from the simple presence of probes in the tracing system.

Each experiment is executed for 30 seconds, which provides stable measurements while keeping the tracing volume manageable. Experiments are conducted on two storage backends: a Samsung SSD mSATA formatted with the `ext4` file system, and a RAM Disk mounted on `/tmp` to represent an ultra-fast storage medium.

To obtain reproducible and low-noise measurements, we flush the page cache and disable swap before each run. For each configuration (including the baseline and each tracing stage), we perform three independent executions and report the average results. We first establish a baseline without tracing, then enable tracing incrementally according to the stages defined in Section 2. This methodology allows us to isolate and quantify the overhead induced by each internal component.

To quantify the performance impact of tracing, we collect the following metrics:

- **Throughput (MB/s):** The sustained aggregate data transfer rate. We compare the baseline throughput against the traced throughput to compute throughput degradation.
- **Component-wise Overhead (%):** The relative performance cost introduced by each individual stage of the tracing pipeline. This overhead is computed by progressively enabling tracing stages defined in Section 2 and measuring the resulting throughput degradation at each step. For a given stage i, the overhead is defined as:
$$Overhead_i = \frac{T_{i-1} - T_i}{T_{base}} \times 100$$
where T_i denotes the throughput measured when stage i is enabled, and T_{i-1} corresponds to the throughput measured at the previous stage.
- **Event rate (events/s):** The number of traced events processed per second. This metric helps correlate performance degradation with tracing intensity, especially under high concurrency and small I/O sizes.

To prevent event loss from affecting the measurements, the ring buffer is configured with its maximum supported size (1 GB). This ensures that throughput degradation is primarily due to tracing overhead rather than trace drops caused by buffer saturation.

All experiments were carried out on a Lenovo ThinkPad T430 featuring an Intel Core i7-3520M processor clocked at 2.90 GHz (TurboBoost up to 3.60 GHz), with 2 physical cores and 4 hardware threads, and 16 GB of RAM. The system is equipped with a Samsung SSD mSATA (256 GB), hosting the Linux installation on an `ext4` partition. The experiments are performed under Linux kernel 6.0.

3.2 Results

We first analyze the throughput degradation according to the workload intensity and traced event volume. Then, we discuss the overhead contribution of each internal stage of the eBPF tracing pipeline. Figure 2 summarizes the performance evaluation of eBPF tracing on both the SSD and the RAM Disk. Specifically, Figures 2(a) and (b) report the throughput evolution under different workloads, Figures 2(c) and (d) present the component-wise overhead breakdown across tracing stages, while Figures 2(e) and (f) show the traced event rate (events/s).

3.2.1 Throughput Degradation.

The results show that eBPF tracing overhead is strongly workload-dependent and closely correlated with the intensity of traced events, which is mainly driven by I/O block size and concurrency level.

For large I/O requests (256 KB), the tracing overhead remains limited. On the SSD, the throughput degradation ranges from only 2.24% (1 job) up to 8.45% (8 jobs), while on the RAM Disk it remains between 10.17% and 15.12%. This behavior can be explained by the low event rate generated under large I/O sizes: although the throughput is high, the number of I/O requests per second is relatively small, resulting in fewer probe activations and fewer events delivered to user space.

In contrast, for small I/O requests (4 KB), the overhead becomes significant. On the SSD, throughput degradation reaches 37.06% with 8 jobs and 38.09% with 16 jobs. Similarly, on the RAM Disk, overhead reaches 39.99% and 41.17% for 8 and 16 jobs, respectively. These configurations correspond to IOPS-intensive workloads, where millions of I/O requests are issued per second, generating an extremely high volume of traced events. In such conditions, the eBPF tracing pipeline becomes CPU-bound and directly competes with the workload for CPU cycles.

The 64 KB configurations exhibit intermediate behavior. On the SSD, throughput degradation increases from 5.55% (1 job) to 19.18% (16 jobs), while on the RAM Disk it varies from 9.43% to 19.79%. These results confirm that the overhead scales with concurrency, since higher job counts lead to a higher request rate, increasing the number of probe triggers and the pressure on the ring buffer retrieval mechanism.

A key observation is that the RAM Disk systematically exhibits a higher overhead than the SSD for most configurations. This can be attributed to the fact that the RAM Disk provides a much faster storage backend, enabling higher request completion rates and therefore higher traced event rates. Consequently, tracing overhead becomes more visible because the storage system itself is no longer the bottleneck, and the tracing pipeline becomes the limiting factor.

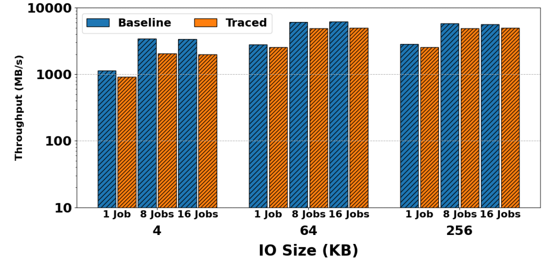
3.2.2 Component-level Overhead Breakdown.

In both storage scenarios SSD and RAM Disk, the results show that eBPF tracing overhead is primarily determined by the event rate rather than absolute throughput. Workloads generating high-frequency events (small I/O sizes and multiple concurrent jobs) incur substantial overhead, while large-block workloads produce minimal performance impact. Across all configurations, probe triggering and event retrieving dominate the total overhead.

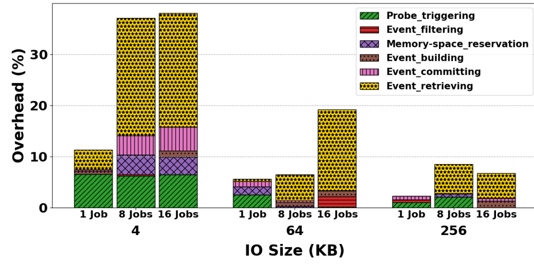
Probe triggering introduces a constant cost for each instrumented kernel function execution, including context initialization and execution of the JIT-compiled eBPF program. In our experiments, it contributes up to 6.52% on SSD (1 job, 4 KB) and 11.07% on RAM Disk (8 jobs, 4 KB), reflecting that every I/O request pays this cost regardless of the probe handler logic.

Event retrieving, corresponding to transferring committed events from the ring buffer to user space, is the main bottleneck. On SSD, it contributes 3.91–23.00%, while on RAM Disk it goes up to 23.55% (16 jobs, 4 KB), confirming that high event rates stress kernel-to-user transfers.

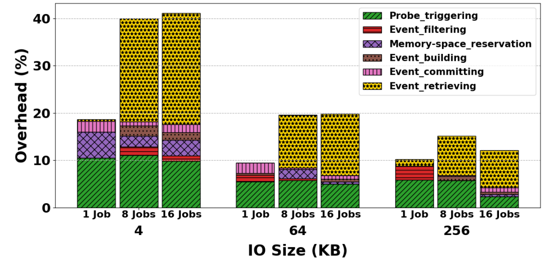
(a) SSD Throughput: Baseline vs. Traced



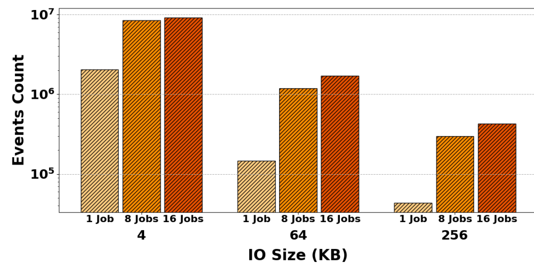
(b) RAM Disk Throughput: Baseline vs. Traced



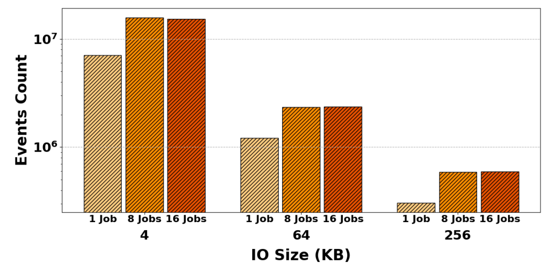
(c) SSD Component-wise Overhead (%)



(d) RAM Disk Component-wise Overhead (%)



(e) SSD Event Rate (events/s)



(f) RAM Disk Event Rate (events/s)

Figure 2. Performance evaluation of eBPF tracing on SSD vs RAM Disk.

Memory-space reservation generates a moderate cost, peaking at 3.84% on SSD and 5.48% on RAM Disk under high-frequency workloads. Event filtering and building impose negligible overhead (<2%), and event committing remains below 5% except in extreme scenarios (e.g., 4.61% on SSD, 16 jobs, 4 KB).

Number of Probes	Overhead (%)	Events (Millions)
2	7.49	11.2
4	11.41	21.5
8	16.56	35.4

Table 1. Case 1: probes placed along the I/O critical path.

3.2.3 Impact of Probe Number and Location .

Table 1 and Table 2 report the results for the multi-level and VFS-only configurations, respectively, illustrating the relationship between probe configuration (number and location), the volume of generated events, and the resulting tracing overhead.

Number of Probes	Overhead (%)	Events (Millions)
2	6.18	11.4
4	6.08	11.4
8	6.91	11.2

Table 2. Case 2: probes are placed at the same level, but only a subset is triggered.

Case 1 – Multi-level Tracing. In this configuration, probes are placed along the critical path of I/O operations, such that all probes are triggered for each request. As the number of probes increases, the number of generated events grows proportionally, leading to higher tracing overhead. The results reveal an approximately linear relationship between event volume and overhead, with about 1% of additional overhead incurred for every 2 million extra events.

This behavior indicates that overhead scales linearly with the rate of event generation.

Case 2 – VFS-only Tracing. This experiment aims to evaluate whether the mere presence of probes affects the tracer's intrusiveness. To this end, additional probes are inserted but remain inactive during execution. Consequently, although the total number of probes increases, the number of generated events remains unchanged at approximately 11.2M. The observed overhead remains stable at about 6.5%, indicating that intrusiveness is primarily governed by probe activation frequency rather than by the total number of probes.

Discussion. Overall, these results indicate that the impact of probe location within the I/O stack is secondary compared to the effect of event generation. While Case 1 may suggest that overhead increases with the number of probes, a more detailed analysis reveals that it is in fact strongly correlated with the total number of processed events. This observation reinforces the existence of an approximately linear relationship between event volume and overhead. Taken together, these findings demonstrate that tracing overhead is primarily driven by the *event processing rate*, defined as the frequency at which probes are triggered and events are handled, rather than by the absolute number of probes or their precise location within the I/O stack.

4 Takeaways and Practical Guidelines

Based on our experimental observations, we highlight the following key takeaways:

(1) Tracing overhead is primarily event-rate dependent. Workloads with small I/O sizes and high concurrency generate a very large number of events per second, which amplifies the cost of every pipeline stage, even those with marginal per-event overhead.

(2) Probe triggering overhead is unavoidable and affects all calls. Once a probe is attached, every execution of the instrumented kernel function incurs a redirection cost, even if minimal filtering is applied. Therefore, tracing highly frequent kernel functions should be avoided unless strictly necessary.

(3) Kernel-to-user event delivery is a major bottleneck. The event retrieving stage becomes dominant under high-IOPS workloads due to frequent wakeups, polling overhead, and context switching. This makes user-space consumption capacity a key scalability limitation.

(4) Filtering and event construction have limited impact compared to delivery. PID filtering and log construction represent only a minor fraction of the total overhead in most configurations, suggesting that optimization efforts should prioritize delivery mechanisms rather than lightweight in-kernel computations.

(5) Storage type influences the perceived overhead. RAM Disk experiments exhibit higher absolute throughput and event rates, making tracing overhead more pronounced than on SSD. This indicates that faster storage media can expose tracing limitations more clearly.

From these takeaways, we derive several practical guidelines to reduce tracer intrusiveness. First, the number of active probes should be reduced, especially those attached to high-frequency execution paths, by favoring selective instrumentation and enabling probes only when needed. Second, sampling or event aggregation strategies should be adopted to limit the volume of traced events, particularly under high concurrency. Third, event delivery mechanisms should be optimized by tuning ring buffer parameters, reducing user-space wakeups, and avoiding excessive polling frequency. In addition, unnecessary metadata collection should be minimized, since large event payloads increase memory pressure and transfer overhead. Finally, tracing configurations should be adapted to workload intensity by restricting tracing scope for high-IOPS workloads and enabling detailed tracing only during short diagnostic windows.

5 Related Work

Several recent studies have investigated the eBPF ecosystem from different perspectives, including tracing overhead mitigation, program optimization, usability challenges, and runtime correctness. For instance, Craun et al. [6] proposed a copy-on-write kernel view mechanism to eliminate tracing overhead for untraced processes, at the cost of additional memory usage. Other works explored optimization techniques such as superoptimization to reduce eBPF execution overhead [16], as well as alternative runtime environments aiming to reduce context-switch costs [9]. Lightweight monitoring approaches have also demonstrated that kernel-resident data structures can provide accurate metrics with limited overhead [25].

Beyond performance aspects, the complexity of developing and operating eBPF applications has been studied through empirical analyses [7]. In parallel, correctness and security issues of the eBPF verifier and runtime have been addressed through enhanced verification and fuzzing-based approaches [17, 26], as well as foundation-driven efforts to improve verifier safety guarantees [11, 12].

Recent work has also investigated the impact of front-end tools and libraries on eBPF-based tracing. In particular, authors in [20] conduct a comparative study of several widely used eBPF libraries (e.g., bpftrace, BCC, libbpf), evaluating their performance, resource usage, and data collection fidelity under different storage I/O workloads. Their results highlight important trade-offs across these frontends and stress the importance of considering quantitative performance metrics when selecting an eBPF-based solution.

While these works provide valuable insights into the performance and reliability of the eBPF ecosystem, they do not provide a detailed characterization of the overhead induced

by each internal stage of the eBPF tracing pipeline under high-throughput storage I/O workloads. Our work fills this gap through a component-level breakdown and systematic experimental evaluation. To the best of our knowledge, no prior work provides a fine-grained breakdown of the eBPF tracing overhead at the level of its internal execution stages.

6 Conclusion

This work presented a component-level analysis of eBPF tracing overhead for storage I/O workloads. Our results show that overhead is highly workload-dependent and is mainly driven by probe triggering and kernel-to-user event retrieval, while filtering, event construction, and committing remain marginal (typically below 5%). Under small I/O sizes and high concurrency, total overhead can reach up to 41%, whereas large block sizes exhibit significantly lower impact, making eBPF tracing more suitable for throughput-oriented workloads. For future work, we plan to extend this analysis to real-world applications (e.g., Big Data, Cloud/HPC, and AI workloads), evaluate additional metrics such as I/O latency and CPU utilization, and assess eBPF tracing across heterogeneous platforms ranging from high-end servers to low-power devices.

References

[1] Dtrace web site. URL http://dtrace.org/blogs/about/.

[2] Lttng web site. URL https://lttng.org/.

[3] Systemtap web site. URL https://sourceware.org/systemtap/.

[4] Jens Axboe. Fio - flexible i/o tester, 2017. URL https://fio.readthedocs.io/en/latest/fio_doc.html.

[5] Jonathan Corbet. An introduction to the bpf compiler collection. URL https://lwn.net/Articles/742082/.

[6] Milo Craun, Khizar Hussain, Uddhav Gautam, Zhengjie Ji, Tanuj Rao, and Dan Williams. Eliminating ebpf tracing overhead on untraced processes. In *Proceedings of the ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions*, pages 16–22, 2024.

[7] Mugdha Deokar, Jingyang Men, Lucas Castanheira, Ayush Bhardwaj, and Theophilus A Benson. An empirical study on the challenges of ebpf application development. In *Proceedings of the ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions*, pages 1–8, 2024.

[8] Omkar Desai, Seungmin Shin, Eunji Lee, and Bryan S Kim. A principled approach for selecting block i/o traces. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, pages 52–58, 2022.

[9] J. Doe and M. Roe. bpftime: A user-space runtime for high-performance ebpf execution. *arXiv preprint arXiv:2311.07923*, 2023. Explores user-space eBPF execution to reduce kernel context switch costs.

[10] Théophile Dubuc, Pascale Vicat-Blanc, Pierre Olivier, Mar Callau-Zori, Christophe Hubert, and Alain Tchana. Trackiops: Real-time nfs performance metrics extractor. In *Proceedings of the 4th Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*, pages 1–8, 2024.

[11] eBPF Foundation Formal Methods Group. Formal verification of ebpf verifier algorithms. eBPF Foundation Research Grant Summary, 2024. Formal methods to prove correctness of core eBPF verifier algorithms.

[12] eBPF Foundation Research Award Team. Verifier-cooperative instrumentation for enhanced ebpf safety. eBPF Foundation Research Report, 2024. Community-funded project for improving eBPF verifier precision with low overhead.

[13] Tânia Esteves, Ricardo Macedo, Rui Oliveira, and João Paulo. 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)*, pages 1–8. IEEE, 2023.

[14] William Findlay. *Extended berkeley packet filter for intrusion detection implementations*. PhD thesis, Honours Thesis Proposal, Carleton University, 2019.

[15] Brendan Gregg. Perf examples. URL http://www.brendangregg.com/perf.html.

[16] Kaiming Huang and A. Collaborator. Epso: Caching-guided superoptimization for ebpf bytecode. *arXiv preprint arXiv:2511.15589*, 2025. Superoptimization techniques for reducing eBPF bytecode size and execution overhead.

[17] Hsin-Wei Hung and Ardalan Amiri Sani. Brf: Fuzzing the ebpf runtime. *Proceedings of the ACM on Software Engineering*, 1(FSE):1152–1171, 2024.

[18] Mohammed Islam Naas, François Trahay, Alexis Colin, Pierre Olivier, Stéphane Rubini, Frank Singhoff, and Jalil Boukhobza. Eziotracer: Unifying kernel and user space i/o tracing for data-intensive applications. *ACM SIGOPS Operating Systems Review*, 55(1):88–98, 2021.

[19] Ganguk Lee, Yeaseul Park, Jeongseob Ahn, and Youngjin Kwon. Slicing the io execution with relaytracer. *arXiv preprint arXiv:1906.07124*, 2019.

[20] Carlos Machado, Bruno Gião, Sebastião Amaro, Miguel Matos, João Paulo, and Tânia Esteves. No two snowflakes are alike: Studying ebpf libraries' performance, fidelity and resource usage. In *Proceedings of the 3rd Workshop on eBPF and Kernel Extensions*, pages 31–37, 2025.

[21] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. Creating complex network services with ebpf: Experience and lessons learned. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–8. IEEE, 2018.

[22] Steven Rostedt. Ftrace documentation. URL https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[23] Abdulqawi Saif, Lucas Nussbaum, and Ye-Qiong Song. Ioscope: A flexible i/o tracer for workloads' i/o pattern characterization. In *International Conference on High Performance Computing*, pages 103–116. Springer, 2018.

[24] Dominik Scholz, Daniel Raumer, Paul Emmerich, Alexander Kurtz, Krzysztof Lesiak, and Georg Carle. Performance implications of packet filtering with linux ebpf. In *2018 30th International Teletraffic Congress (ITC 30)*, volume 1, pages 209–217. IEEE, 2018.

[25] C. Smith and D. Lee. ehashpipe: Lightweight ebpf-based resource monitoring with sketching. *arXiv preprint arXiv:2509.09879*, 2025. Demonstrates precise low-cost metrics collection using lightweight sketches.

[26] Hao Sun, Yiru Xu, Jianzhong Liu, Yuheng Shen, Nan Guan, and Yu Jiang. Finding correctness bugs in ebpf verifier with structured and sanitized program. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 689–703, 2024.