# Diagnosing applications' I/O behavior through system call observability

Tânia Esteves
*INESC TEC & U. Minho*
tania.c.araujo@inesctec.pt

Ricardo Macedo
*INESC TEC & U. Minho*
ricardo.g.macedo@inesctec.pt

Rui Oliveira
*INESC TEC & U. Minho*
rui.oliveira@inesctec.pt

João Paulo
*INESC TEC & U. Minho*
jtpaulo@inesctec.pt

***Abstract*—We present DIO, a generic tool for observing inefficient and erroneous I/O interactions between applications and in-kernel storage systems that lead to performance, dependability, and correctness issues. DIO facilitates the analysis and enables near real-time visualization of complex I/O patterns for data-intensive applications generating millions of storage requests. This is achieved by non-intrusively intercepting system calls, enriching collected data with relevant context, and providing timely analysis and visualization for traced events.**

**We demonstrate its usefulness by analyzing two production-level applications. Results show that DIO enables diagnosing resource contention in multi-threaded I/O that leads to high tail latency and erroneous file accesses that cause data loss.**

***Index Terms*—Storage systems, I/O diagnosis, tracing, analysis**

## I. Introduction

The performance, correctness and dependability of data-intensive applications (*e.g.*, databases, key-value stores, analytical engines, machine learning frameworks) is highly influenced by the way these interact with in-kernel POSIX storage backends, such as file systems and block devices [1], [2].

Due to human error and lack of detailed knowledge on how to efficiently and correctly access the storage backend, developers often implement applications that exhibit: *i)* costly access patterns, such as small-sized I/O requests or random accesses; *ii)* I/O contention caused by having concurrent requests accessing shared storage resources; and *iii)* erroneous usage of I/O calls, for example, by accessing wrong file offsets. These patterns lead to inefficient or incorrect storage I/O accesses, which not only compromise the usefulness of optimizations implemented within each storage backend (*e.g.*, caching, scheduling), but can ultimately degrade end-to-end performance, negatively impact availability, and even cause data loss for applications.

The sheer amount of storage operations generated by these applications, which can range from hundreds to thousands of operations per second, makes their analysis a complex and time-consuming task when done manually. Thus, diagnosis tools that can help users and developers to profile more precisely the I/O interaction between applications and corresponding storage backends are crucial for debugging errors, finding performance and dependability issues, and identifying potential optimizations for applications [3]–[5].

The main insight of this paper is that, by combining system call (or *syscall* for short) tracing with a customizable analysis pipeline, one can provide non-intrusive and comprehensive I/O diagnosis for applications using in-kernel POSIX storage systems (*e.g.*, file system, Linux block device). Doing so requires overcoming the following limitations of existing approaches.

**Intrusiveness.** The collection of information about I/O requests is often done through source code instrumentation [6]–[9]. This approach is not easily applicable across different applications, as it requires users to manually analyze and instrument distinct and potentially large codebases.

**Practicality.** I/O requests can be intercepted non-intrusively with kernel-level tracing technologies. However, the performance penalty imposed on the application by widely-used solutions, such as strace [10], can make this choice unpractical for data-intensive workloads. Namely, it significantly increases the time for tracing requests and, due to the performance slowdown, can hide subtle concurrency issues, such as I/O contention or starvation [4], [11]. This challenge motivated the emergence of technologies such as eBPF [12] and LTTng [13], which follow a non-blocking tracing strategy that reduces performance overhead at the cost of potentially discarding I/O events that cannot be processed in a timely fashion.

**Lack of analysis pipeline.** While efficient I/O tracing is an important step for profiling applications, by itself is not sufficient, given the large amount of collected events (easily reaching tens of millions) that must be parsed, correlated, and visually represented to provide insightful information (*e.g.*, showcase contention in multi-threaded I/O). Several solutions only cover the tracing collection step, delegating these other time-consuming tasks to users [10], [14]–[16].

**Flexibility.** Solutions offering a complete pipeline for application diagnosis are designed for rigid analysis scenarios, such as detecting unreproducible builds [17], observing file offset access patterns [5], or identifying security issues [18], [19]. Thus, for multi-purpose profiling tasks, one must combine several of these tools and repeat multiple times the tracing, analysis, and visualization of the same application. Ideally, diagnosis tools should provide the flexibility to narrow or broaden both tracing and analysis scopes based on user goals. This would enable exploring a wider range of performance, correctness, and dependability issues that applications may exhibit, as those presented in §III.

This paper proposes DIO, a generic tool for observing and diagnosing applications' storage I/O, which addresses these

challenges with the following contributions.

**Non-intrusive, comprehensive, and flexible tracing.** DIO offers a new eBPF-based tracer that intercepts syscalls issued by applications without requiring changes to their source code or instrumentation of binaries. The tracer supports 42 storage-related syscalls and records a comprehensive set of information for each collected operation, including its type, arguments, return value, timestamps, ProcessID (PID), and ThreadID (TID). By offering a flexible design, DIO allows collecting only events of interest, filtering them (in kernel-space) by syscall type, PID, TID, or file paths. This enables narrowing the tracing scope according to users' requirements and minimizing performance overhead over the targeted application.

**Enriched analysis.** DIO enriches data gathered for each syscall with additional context available at the kernel (*e.g.,* process name, file type, file offset), which can be used to improve the correlation and analysis of requests (*e.g.*, associating different syscalls to a file path, differentiating operations over regular files or directories). These features enable a richer and wider analysis of incorrect or inefficient I/O patterns.

**Asynchronous event handling.** Only syscall interception is done synchronously, while collected events are sent and stored at a remote backend asynchronously. This avoids adding extra latency in the critical path of I/O requests and enables practical analysis of long and data-intensive storage workloads.

**Near real-time pipeline.** DIO offers a practical and customizable pipeline so that users can create their own queries, correlation algorithms, and visualization dashboards to analyze collected data. The pipeline follows an in-line approach, meaning that traced information is automatically parsed and forwarded to the analysis and visualization components as soon as it is captured without requiring manual user intervention.

DIO is implemented as an open-source prototype using eBPF [12], Elasticsearch [20], and Kibana [21], and validated with production-level systems. Results show that DIO enables the diagnosis of *i)* erroneous file accesses that cause data loss in Fluent Bit, and *ii)* resource contention in multi-threaded I/O that leads to high tail latency for user workloads in RocksDB. All artifacts discussed in this paper, including DIO, workloads, scripts, and the corresponding analysis and visualization outputs are publicly available.[1]

## II. The DIO Tool

DIO is a generic tool for observing and diagnosing the I/O interactions between applications and in-kernel POSIX storage systems. Its design is built over the following core principles, which address the challenges discussed in §I.

**Transparency.** DIO relies on the Linux kernel tracing infrastructure, namely tracepoints and kernel probes, to intercept applications' syscalls without requiring any modification to their source code or underlying libraries.

**Practical and timely analysis.** Traced data is asynchronously sent to a remote analysis pipeline, avoiding adding extra latency on the critical I/O path of applications while enabling users to visualize collected data in near real-time.

[1] https://github.com/dsrhaslab/dio

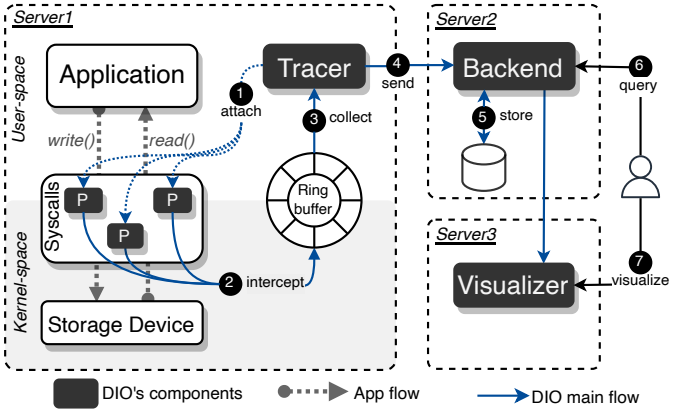


**Fig. 1:** DIO*'s design and flow of events.*

**Post-mortem analysis.** DIO allows storing different tracing executions from the same or different applications and posteriorly analyzing and comparing them.

**Flexible and comprehensive tracing.** DIO intercepts different types of storage-related syscalls, covering data (*e.g.*, `write`), metadata (*e.g.*, `stat`), extended attributes (*e.g.*, `getxattr`), and directory management (*e.g.*, `mknod`) requests. Users can choose to capture only the relevant syscalls for their analysis goals and further filter these based on targeted PIDs, TIDs, or file paths.

**Enriching syscall analysis.** DIO enriches the information provided directly by each syscall (*i.e.*, type, arguments, return value) with additional context from the kernel, such as the name of the process that originated the request, the type of the file being accessed by it, and its offset.

**Data querying and correlation.** With DIO, users can query traced data, apply filters to analyze specific information (*e.g.*, syscalls executed by a specific TID), and correlate different types of data (*e.g.*, associate file descriptors with file paths).

**Customized visualization.** DIO comprises a visualization component that provides mechanisms for simplifying the exploration of traced data and to build customized visualizations.

### A. System overview

DIO consists of three main components, namely the *tracer*, the *backend*, and the *visualizer*, as depicted in Fig. 1. DIO's analysis pipeline includes the latter two components.

The *tracer* intercepts syscalls from applications, filters them according to the user's configurations (*e.g.*, by TID), and packs their information into events that are asynchronously sent to the *backend* (❹). The *backend* persists and indexes events (❺), and allows users to query and summarize (*e.g.*, aggregating) stored information (❻).

The *visualizer* provides near real-time visualization of the traced events by querying the *backend* (❼). Users rely on the *visualizer* to ease the process of data exploration and

**TABLE I:** *Syscalls supported by* DIO.

| Type | Syscall |
|---|---|
| ***Data*** | `read, pread64, readv, write, pwrite64 writev, fsync, fdatasync, readahead` |
| ***Metadata*** | `creat, open, openat, close lseek, truncate, ftruncate rename, renameat, renameat2 unlink, unlinkat, readlink, readlinkat stat, lstat, fstat, fstatfs, fstatat` |
| ***Extended*** | `getxattr, lgetxattr, fgetxattr setxattr, lsetxattr, fsetxattr listxattr, llistxattr, flistxattr removexattr, lremovexattr, fremovexattr` |
| ***Directory*** | `mknod, mknodat` |

analysis, by selecting specific types of data (*e.g.*, syscall types, arguments) to build different and customized representations.

### B. Tracer

The *tracer* intercepts syscalls done by applications in a non-intrusive way. To that end, it relies on the eBPF technology [12], which allows instrumenting the Linux kernel by executing small programs (*i.e.*, eBPF programs) whenever a given point of interest (*e.g.*, tracepoints, kprobes) is called.

In detail, DIO's *tracer* comprises a set of eBPF programs that, at the initialization phase (❶), are automatically and transparently attached to syscall tracepoints. Whenever these tracepoints are reached (*i.e.*, a syscall is invoked), the eBPF program gathers the information about the request and places it in a per-CPU *ring buffer* (❷), which is a contiguous memory area used for exchanging data between kernel (producers) and user-space (consumers) processes. At user-space, the *tracer* asynchronously fetches information from the *ring buffer* (❸), parses it into events (specified in JSON objects), and sends these to the *backend* (❹). To minimize both network and performance overhead, the tracer groups several events into buckets that are sent and indexed in batches at the *backend*.

Table I depicts the syscalls supported by DIO. Currently, DIO focuses on storage-related operations, but it can be extended to support other syscalls. Since instrumenting syscalls can introduce extra processing in the critical path of I/O requests, DIO allows users to filter requests by: *i)* type, *ii)* process or thread ID(s), and *iii)* targeted file or directory path(s). By implementing these filters in the kernel, DIO reduces the amount of information sent to user-space.

**Collected information.** For each intercepted syscall, DIO collects information related to the: *i)* request (type, arguments, and return value); *ii)* process (PID, TID, and process name) and *iii)* time (*entry* and *exit* timestamps). While this information alone already provides valuable insights about applications' I/O behavior (*e.g.*, syscalls issued over time, size of I/O requests), correlating it with other types of data further enriches and eases the users' analysis (as further discussed in §III). Therefore, the *tracer* leverages eBPF's access to kernel structures and enriches traced information with:

- the *file type* targeted by syscalls, which enables differentiating accesses to regular files, directories, sockets, block/char devices, pipes, symbolic links, and other files.
- the *file offset* being accessed by data-related syscalls. This information allows observing file access patterns (*e.g.*, random accesses), even for syscalls that do not provide the file offset as an argument (*e.g.*, `read`, `write`).
- a *file tag* that labels syscalls handling file descriptors (*e.g.*, `read`, `close`) with a tag containing the device number, inode number, and first file access timestamp that uniquely identify the file being accessed.

### C. Backend

The *backend* allows persisting, searching, and analyzing data from traced events. It uses the Elasticsearch [20] distributed engine for storing and processing large volumes of data. Its flexible document-oriented schema allows indexing events as documents, even if these have potentially different structures (*e.g.*, distinct fields corresponding to syscall arguments). Moreover, it provides an interface for searching, querying, and updating documents, which allows users to develop and integrate customized data correlation algorithms.

**File path correlation algorithm.** We have implemented a custom algorithm to enable the correlation of syscalls with specific accessed file paths. Using Elasticsearch's data querying and updating features, the file tags (*i.e.*, unique identifiers generated by the *tracer* component) associated with syscalls are translated into the actual file paths being accessed at the storage backend (*e.g.*, `/tmp/fileA`).

### D. Visualizer

The *visualizer* provides an automated approach towards exploring (*e.g.*, query and filter events) and visually depicting (*e.g.*, through tables, histograms, time-series graphs) the analysis findings. This component uses Kibana [21], the data visualization dashboard software for Elasticsearch, which is often used for log and time-series analytics and application monitoring. Kibana also allows building custom visualizations, thus being aligned with the design principles of DIO.

### E. Implementation

The tracer is implemented in ≈8K LoC, divided into two parts: *i)* the eBPF programs that run in kernel-space and *ii)* the user-space code including the remaining tracer's logic.

The eBPF programs are implemented in C and are responsible for collecting and filtering relevant storage I/O events. The user-space code is implemented in Go (v17.4) and is responsible for enabling the desired I/O tracepoints (attaching eBPF programs), specifying the user-defined filters applied at each tracepoint, gathering and parsing the information collected at kernel-space, and sending it to the *backend* component. This is done using the BPF Compiler Collection (BCC) framework through the *gobpf* lib (v0.2.0), which provides an abstraction for creating, attaching, and interacting with eBPF

programs. For communication with Elasticsearch, we use the *go-elasticsearch* (v7.13.1) module, taking advantage of the bulk indexing API for sending multiple events simultaneously.

The *backend* and *visualizer* components use Elasticsearch (v8.5.2) and Kibana (v8.5.2), respectively. The file path correlation algorithm can be automatically executed by the *tracer* or on-demand by users.

### F. Configuration and Usage

The installation and configuration of DIO are performed in two phases: *i)* the setup and initialization of the analysis pipeline and *ii)* the configuration and execution of the *tracer*.

**Analysis pipeline.** Although all DIO's components can be deployed in the same machine, to avoid negatively impacting the performance of the targeted application (*e.g.*, additional resource consumption), the analysis pipeline can be installed on a separate server(s) (Fig. 1). Further, as the *tracer* component labels each tracing execution with a unique session name, one can deploy DIO as a service, setting up the analysis pipeline on dedicated servers and allowing multiple executions of DIO's *tracer* on different machines and by distinct users.

The deployment and configuration of the analysis pipeline comprise its software installation (*i.e.*, Elasticsearch and Kibana) and importing its predefined dashboards. As soon as tracing data arrives at the pipeline, users can access Kibana's web page and visualize DIO's dashboards, apply analysis filters, and edit or create new visualizations and dashboards.

**Tracer.** Once the analysis pipeline is deployed, users can use DIO's *tracer* to collect information. The *tracer* executes along with the targeted application, stopping once its main and child processes finish or upon explicit users' instruction.

By default, DIO's *tracer* enables tracepoints for the full set of supported syscalls. However, users can specify a list of syscalls to observe, and the *tracer* will only activate tracepoints for those operations. Also, users may specify a list of files/directories to observe, instructing the *tracer* to only record events that target them. All these configurations, along with the analysis pipeline's parameters (*e.g.*, Elasticsearch URL), can be set through a configuration file.

## III. Evaluation

Our evaluation showcases how DIO can be used to ease the process of observing/confirming known issues and validating their fixes. To this end, we analyzed two production-level applications: Fluent Bit and RocksDB. Results show that DIO is a practical tool for validating the root causes of correctness (§III-B) and performance (§III-C) issues, without instrumenting large codebases. With the exception of Fig. 3, all the remaining figures in this section were generated by DIO (with minimal modifications for readability)[2].

### A. Experimental Setup

Our testbed comprises three servers running *Ubuntu 20.04 LTS* with kernel *5.4.0*. The server running the application and DIO's *tracer* is equipped with a 4-core Intel Core i3-7100, 16 GiB of memory, a 250 GiB NVMe SSD (which hosts the datasets), and a 512 GiB SATA SSD (used for logging). DIO's *backend* and *visualization* components run on two separate servers, equipped with a 6-core Intel i5-9500, 16 GiB of memory, and a 250 GiB NVMe SSD.

### B. Identifying erroneous actions that lead to data loss

DIO can assist developers and users in diagnosing the correctness of their applications. We demonstrate this by showing erroneous I/O access patterns that result in data loss.

For this use case, we consider Fluent Bit (v1.4.0), a high-performance logging and metrics processor and forwarder [22]. Existing issues[3,4] report that data is lost when using the `tail` input plugin, which is used to fetch new content being added to log files. Thus, we implemented a client program that simulates the generation of log files to be processed by Fluent Bit and mimics the I/O behavior reported in Issue #1875[3]. DIO was used to simultaneously trace and analyze the client program and Fluent Bit by filtering the syscalls belonging to the set of processes of these applications.

Fig. 2a shows a tabular visualization generated by DIO. The client program (*app*) starts by creating the `app.log` file, writing 26 bytes starting from offset 0, and closing the file (❶). Then, Fluent Bit (*fluent-bit*) detects content modification at the file, opens it, and reads 26 bytes from offset 0, which means that *fluent-bit* processes the full content previously written by *app* (❷). Later, *app* removes the file with the `unlink` syscall, and *fluent-bit* closes the corresponding file descriptor (❸). At the operating system level, this means that the inode number associated with this file (`12`) is now unused and will later be attributed to a new file. However, a possible scenario is this inode number being mapped to a newly created file with the same name. This happens when *app* creates a new file with the same name as the previous one (`app.log`) and writes 16 bytes to it (❹). The incorrect behavior reported at the issue, and observable with DIO, happens when *fluent-bit* opens the new log file for reading its content, but instead of reading from offset 0, as expected, it starts reading at offset 26 (❺). By starting at the wrong offset, the `read` syscall returns zero bytes, and the 16 bytes written by *app* are lost.

To understand the reason for this behavior, we examined Fluent Bit's code responsible for reading new content entries in log files. Before reading a file, Fluent Bit updates the file position to the number of bytes already processed. This value is kept on a database for each tracked file, identified by its name plus inode number. Erroneously, database entries are not deleted when files are removed from the file system. Therefore, and going back to our running example, since the same file name (`app.log`) and inode number (`12`) are attributed to the newly created file, *fluent-bit* erroneously assumes that the first 26 bytes of the latter log file were already processed.

To validate the correction of this erroneous access pattern, we used DIO to analyze a more recent version of Fluent Bit

[2]DIO's visualizations are available at https://github.com/dsrhaslab/dio.

[3]https://github.com/fluent/fluent-bit/issues/1875

[4]https://github.com/fluent/fluent-bit/issues/4895

| time | proc_name | syscall | ret val | file_tag (dev_no\|inode_no\|timestamp) | offset | |
|---|---|---|---|---|---|---|
| 1,679,308,382,363,981,568 | app | openat | 3 | 7340032\|12\|2156997363734041 | - | |
| 1,679,308,382,364,387,584 | app | write | 26 | 7340032\|12\|2156997363734041 | 0 | ❶ |
| 1,679,308,382,364,442,624 | app | close | 0 | 7340032\|12\|2156997363734041 | - | |
| 1,679,308,386,884,300,800 | fluent-bit | openat | 23 | 7340032\|12\|2156997363734041 | - | |
| 1,679,308,386,889,688,320 | fluent-bit | read | 26 | 7340032\|12\|2156997363734041 | 0 | ❷ |
| 1,679,308,386,892,196,096 | fluent-bit | read | 0 | 7340032\|12\|2156997363734041 | 26 | |
| 1,679,308,392,364,854,016 | app | unlink | 0 | - | - | ❸ |
| 1,679,308,392,365,804,032 | fluent-bit | close | 0 | 7340032\|12\|2156997363734041 | - | |
| 1,679,308,402,365,455,104 | app | openat | 3 | 7340032\|12\|2157017365367381 | - | |
| 1,679,308,402,365,598,976 | app | write | 16 | 7340032\|12\|2157017365367381 | 0 | ❹ |
| 1,679,308,402,365,668,864 | app | close | 0 | 7340032\|12\|2157017365367381 | - | |
| 1,679,308,406,884,280,320 | fluent-bit | openat | 23 | 7340032\|12\|2157017365367381 | - | |
| 1,679,308,406,884,805,120 | fluent-bit | lseek | 26 | 7340032\|12\|2157017365367381 | 26 | |
| 1,679,308,406,885,053,440 | fluent-bit | read | 0 | 7340032\|12\|2157017365367381 | 26 | ❺ |
| 1,679,308,422,386,589,952 | fluent-bit | close | 0 | 7340032\|12\|2157017365367381 | - | |

**(a)** *Fluent Bit (v1.4.0) erroneous access pattern.*

| time | proc_name | syscall | ret val | file_tag (dev_no\|inode_no\|timestamp) | offset | |
|---|---|---|---|---|---|---|
| 1,679,248,356,503,484,160 | app | openat | 3 | 7340032\|12\|2096971503238627 | - | |
| 1,679,248,356,503,664,128 | app | write | 26 | 7340032\|12\|2096971503238627 | 0 | ❶ |
| 1,679,248,356,503,719,680 | app | close | 0 | 7340032\|12\|2096971503238627 | - | |
| 1,679,248,361,001,024,256 | flb-pipeline | openat | 46 | 7340032\|12\|2096971503238627 | - | |
| 1,679,248,361,007,723,776 | flb-pipeline | read | 26 | 7340032\|12\|2096971503238627 | 0 | ❷ |
| 1,679,248,361,008,218,112 | flb-pipeline | read | 0 | 7340032\|12\|2096971503238627 | 26 | |
| 1,679,248,366,503,962,624 | app | unlink | 0 | - | - | ❸ |
| 1,679,248,366,506,702,336 | flb-pipeline | close | 0 | 7340032\|12\|2096971503238627 | - | |
| 1,679,248,376,505,657,344 | app | openat | 3 | 7340032\|12\|2096991505568257 | - | |
| 1,679,248,376,505,789,184 | app | write | 16 | 7340032\|12\|2096991505568257 | 0 | ❹ |
| 1,679,248,376,505,878,272 | app | close | 0 | 7340032\|12\|2096991505568257 | - | |
| 1,679,248,381,000,811,264 | flb-pipeline | openat | 46 | 7340032\|12\|2096991505568257 | - | |
| 1,679,248,381,001,634,304 | flb-pipeline | read | 16 | 7340032\|12\|2096991505568257 | 0 | |
| 1,679,248,381,001,834,496 | flb-pipeline | read | 0 | 7340032\|12\|2096991505568257 | 16 | ❺ |
| 1,679,248,381,002,218,240 | flb-pipeline | read | 0 | 7340032\|12\|2096991505568257 | 16 | |
| 1,679,248,397,000,544,000 | flb-pipeline | close | 0 | 7340032\|12\|2096991505568257 | - | |

**(b)** *Fluent Bit (v2.0.5) correct access pattern.*

**Fig. 2:** *Fluent Bit erroneous access pattern leading to data loss.*

(v2.0.5), where fixes were applied to avoid this data loss issue. Fig. 2b shows a similar tabular visualization for the fixed version. While the two versions present similar behavior (same file accesses for ❶-❹), the difference relies on the file offset being accessed by Fluent Bit (*flb-pipeline*) when reading from a new file (❺). This time, Fluent Bit starts reading from the beginning of the file (offset 0), being able to read the new 16 bytes written by *app*.

This example shows that DIO helps users diagnose incorrect I/O behavior from applications and find the root cause for dependability issues such as data loss. Further, while this example only showcases a small amount of lost data, it can be significantly higher when dealing with larger log files. Moreover, this use case also exemplifies how DIO helps validate the corrections applied to the applications' implementation.

### C. Finding the root cause of performance anomalies

We now demonstrate how DIO can also ease the process of diagnosing performance issues by identifying the root cause for high tail latency at client requests issued to RocksDB, an embedded key-value store (KVS) [23].

This phenomenon was first observed in SILK [24] and, therefore, we followed the same testing methodology to reproduce it. We used the *db_bench* benchmark [25] configured with 8 client threads performing a mixture of read-write requests in a closed loop (YCSB A [26]). RocksDB was configured with 8 background threads, namely 1 for flushes and 7 for compactions. Fig. 3 reports a sample of a 5-hour long execution and depicts the $99^{th}$ percentile latency experienced by clients. Throughout this sample, clients observe several latency spikes that range between 1.5 ms to 3.5 ms.

Finding the root cause for this performance penalty through RocksDB codebase instrumentation would require inspecting more than 440K LoC and adding debugging code to several core components. Alternatively, with DIO, one can easily trace, analyze, and visualize RocksDB execution, as depicted

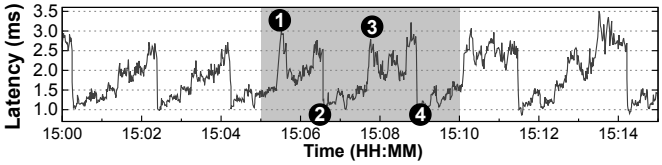


**Fig. 3:** *$99^{th}$ percentile latency for RocksDB client operations.*

in Fig. 4. Since the workload is data-oriented, we configured DIO's *tracer* to capture exclusively `open`, `read`, `write`, and `close` syscalls. Client threads are represented as `db_bench`, while `rocksdb:high0` respects to the flushing thread, and the remainder (`rocksdb:lowX`) to compaction threads.

By observing the syscalls submitted over time by different RocksDB threads, one can identify performance contention. Namely, as shown by the highlighted red boxes, when multiple compaction threads submit I/O requests, the number of syscalls of `db_bench` threads decreases, causing an immediate tail latency spike perceived by clients, as depicted in Figs. 3 and 4 (in intervals ❶ and ❸, at least 5 compaction threads submit requests). When fewer compaction threads perform I/O, the performance of `db_bench` threads improves both in terms of tail latency and throughput (in intervals ❷ and ❹, only 1 to 2 compaction threads are performing I/O).

If one complements the previous observation with knowledge of how Log Structured Merge-tree (LSM) KVSs work, the problem becomes clear: RocksDB uses foreground threads to process client requests (`db_bench` threads), which are enqueued and served in FIFO order. In parallel, background threads serve internal operations, namely flushes (`rocksdb:high0`) and compactions (`rocksdb:lowX`). Flushes ensure that in-memory key-value pairs are sequentially written to the first level of the persistent LSM tree ($L_0$), and these can only proceed when there is enough space at $L_0$. Compactions are held in a FIFO queue, waiting to be executed by a dedicated thread pool. Except for low-level com-

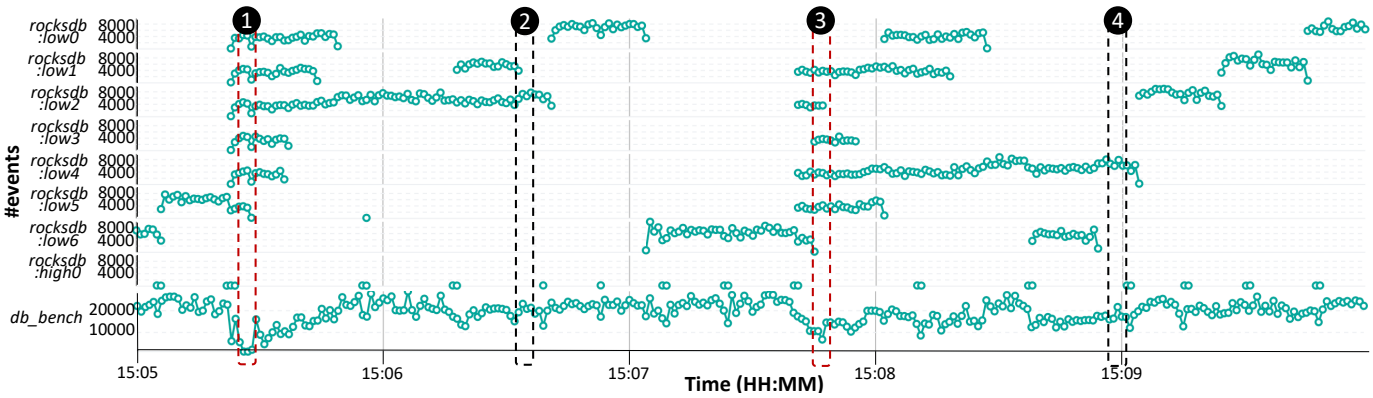


**Fig. 4:** *Syscalls issued by RocksDB over time, aggregated by thread name.* `db_bench` *includes the 8 client threads,* `rocksdb:low[0-6]` *refers to each compaction thread, and* `rocksdb:high0` *refers to the flush thread.*

**TABLE II:** *Average execution time and standard deviation for 3 independent runs of RocksDB.*

| | *vanilla* | *sysdig* | *DIO* | *strace* |
|---|---|---|---|---|
| **Average execution time** | 03h48m | 03h56m | 05h12m | 06h30m |
| **Standard deviation** | ±1.2m | ±1.7m | ±2.4m | ±4.6m |
| **Overhead** | | 1.04× | 1.37× | 1.71× |

pactions ($L_0 \rightarrow L_1$), these can be made in parallel. A common problem of compactions, however, is the interference between I/O workflows, generating latency spikes for client requests. Specifically, latency spikes occur when client threads cannot proceed because $L_0 \rightarrow L_1$ compactions and flushes are slow or on hold, which happens, for instance, when several threads compete for shared disk bandwidth (creating contention). This is precisely the phenomenon identified in SILK, which can negatively impact the response time and even the availability of KVSs and services that use them [27], [28], and that can be observed with DIO without any code instrumentation.

### D. Performance impact and I/O events handling

To understand the performance impact induced by DIO when intercepting I/O syscalls, we selected the RocksDB use case, which includes a benchmark, and measured the average execution time of three independent runs.

**Performance analysis.** Table II compares the *vanilla* deployment (*i.e.*, without tracing its execution) with DIO, *strace* [10] – a widely used syscall tracer, and *Sysdig* [14] – a state-of-the-art eBPF-based syscall tracer.

Executing the *vanilla* setup requires approximately 3 hours and 48 minutes. Compared to *vanilla*, *Sysdig* imposes the smallest overhead (1.04×), using 8 minutes more to execute, while *strace* has the highest overhead (1.71×), lasting for more 2 hours and 42 minutes. DIO increases performance overhead by 1.37×, needing extra 1 hour and 16 minutes when compared to *Sysdig*, but saving up to 1 hour and 18 minutes regarding *strace*.

The difference between *strace* and the two eBPF-based tracers (i.e., *sysdig* and DIO) can be explained by the underlying tracing technology. The trap mechanism used to intercept syscalls and the context switching done by *strace* impose considerable overhead over the targeted application [11].

Regarding the eBPF-based tracers, *Sysdig* presents smaller performance overhead than DIO, but it also reports less information to users. Namely, while DIO is unable to report file paths for up to 5% of the collected events (due to discarded events), *Sysdig* is unable to report these for 45% of the events. As expected, making more information available to the user induces a higher performance penalty in the applications' execution time.

**I/O events handling.** As discussed in §II, DIO uses a fixed-sized *ring buffer* to collect information at user-space, which was configured with 256 MiB per CPU core for these experiments. When this buffer is full (*i.e.*, if kernel processes are producing I/O events to the *ring buffer* at a faster pace than the user-space processes can consume them), new I/O events being intercepted at the kernel level are discarded.

For the RocksDB experiments, given its intensive I/O behavior, 3.5% of the syscalls (≈19M of 549M) were discarded at the *ring buffer* and, therefore, not stored at DIO's *backend*.

### E. Summary

The previous use cases demonstrate DIO's capabilities for diagnosing distinct I/O patterns, which enables users to observe applications' I/O behaviors, confirm known issues, and validate the correction of their fixes. Moreover, our integrated tracing and analysis pipeline allows users to observe these I/O patterns without resorting to code instrumentation or needing to manually combine multiple tools.

Our preliminary experimental results show that DIO can collect, parse, and forward to the analysis pipeline all the required tracing information while imposing reduced performance overhead. Further, despite the discarded I/O events in RocksDB, we show that DIO is still able to pinpoint resource contention and help diagnose its root cause. Moreover, unlike in *strace* and *sysdig*, DIO's traced information is made available for visualization as soon as it is intercepted and transmitted to the *backend* component.

## IV. Related Work

**I/O tracing.** Storage I/O diagnosis is often done by capturing applications' requests in user-space through source code instrumentation [6]–[9]; through middleware libraries [29], [30] that are restricted to specific sets of applications (*e.g.*, *LD_PRELOAD* only works with dynamic libraries); or at lower kernel layers [5], [19], [30], such as the Virtual File System, where optimizations like I/O merging make it impossible to observe the exact requests submitted by applications.

To intercept I/O operations non-intrusively and closer to the requests made by applications, other solutions rely on the syscall interface. As shown in Table III, these explore distinct tracing technologies, including ptrace ([10], [17]), eBPF ([4], [14], [16]), LTTng ([3], [15], [31]), and auditd ([18]), which allow gathering information related with the *entry* and *exit* points of syscalls, including their arguments, return value, timestamps, PIDs, *etc*. Similar to DIO, some tools enrich traced data with additional information such as the *process name* ([4], [14], [16], [18]), which is useful for observing the I/O patterns at §III-B, and §III-C. However, DIO is the only tool that collects *file offsets*, which are crucial for diagnosing the use case presented in §III-B.

Only CaT [4], Tracee [16], and DIO aggregate the information contained at the *entry* and *exit* points of each syscall into a single event, thus simplifying its posterior analysis. This is done at kernel-space to reduce the data transferred to user-space. Further, these are the only tools, along with *strace* [10] and *Sysdig* [14], that support filtering at the tracing phase.

**Integrated analysis pipeline.** Several solutions only cover the tracing step, leaving the integration with analysis pipelines to be done by users [10], [14]–[16]. Other tools provide modules for automating the analysis of traced data but follow an offline approach, where this data needs to be stored first and, only later, it is parsed and provided as input to the analysis pipeline [3], [4], [17], [31]. Only DIO and Longline [18] automatically parse and forward traced events to the analysis pipeline by following an inline (near real-time) approach.

**Syscall analysis.** Some of the existing tools support analysis modules specialized for their concrete use cases (*e.g.*, causality [4], [17], security analysis [18]), which only consider specific information collected from traces (*e.g.*, syscall types). Therefore, these do not provide the flexibility to implement custom analysis algorithms nor enable users to access and explore other information contained in the collected I/O traces. On the other hand, solutions similar to DIO that support customizable analysis fail to capture relevant information to diagnose the use cases discussed in this paper [3], [31].

DIO provides users access to the complete set of captured information (*e.g.*, syscall type, arguments, offsets), allowing them to build new algorithms over the data fields that are more relevant to their analysis goals.

**Syscall visualization.** DIO offers predefined representations that automatically summarize and allow the visualization of the I/O patterns discussed in the paper. Moreover, our tool enables users to create new visualizations commonly supported by other diagnosis solutions (*e.g.*, tables, pie charts, histograms, heatmaps, time series) [3], [18], [31].

**TABLE III:** *Comparison between* DIO *and other solutions in terms of: captured tracing information, filtering capabilities, tracing and analysis integration (O-offline, I-inline), analysis customization, and predefined visualization support. While some tools are able to trace (T) the information required for the paper's use-cases, only* DIO *provides users with the analysis (A) capabilities to diagnose them.*

| | | *Strace*[10] | *Sysdig*[14] | *Re-Animator*[15] | *RepTrace*[17] | *Tracee*[16] | *CAT*[4] | [3] | [31] | *LongLine*[18] | *DIO* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ***Tracing*** | Syscall info | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | *f_offset* | - | - | - | - | - | - | - | - | - | ✓ |
| | *f_type* | - | ✓ | - | - | - | - | - | - | - | ✓ |
| | *proc_name* | - | ✓ | - | - | ✓ | ✓ | - | - | ✓ | ✓ |
| | Filters | ✓ | ✓ | - | - | ✓ | ✓ | - | - | - | ✓ |
| ***Analysis pipeline*** | Integrated | - | - | - | O | - | O | O | O | I | I |
| | Customizable | - | - | - | - | - | - | ✓ | ✓ | - | ✓ |
| | Predefined vis. | - | - | - | - | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| ***Use cases*** | §III-B | - | - | - | - | - | - | - | - | - | TA |
| | §III-C | - | T | - | - | T | T | - | - | T | TA |

## V. Future directions

By intercepting applications' I/O syscalls in a non-intrusive way and automatically parsing and forwarding the collected data to an analysis pipeline, DIO saves users the time needed to understand the applications' source code, instrument the relevant parts, and manually parse the resulting data.

While the current prototype already provides relevant and summarized information about the targeted application, it would be interesting to further simplify the analysis process for users with, for instance, new automated correlation algorithms. Therefore, as a future direction, we intend to explore the Elasticsearch query API and build a collection of correlation algorithms that can, for instance, quickly identify the inefficient behaviors observed in the aforementioned applications.

Moreover, we plan to expand DIO's scope to showcase its capabilities for exploring other applications, even when users are unfamiliar with these, while potentially uncovering new I/O patterns and unidentified issues regarding the performance, dependability, correctness, and security of such applications.

Finally, we aim to analyze the performance overhead imposed by DIO over targeted applications in more detail and study new optimizations that minimize this penalty and reduce the number of I/O events discarded at the tracing phase.

## VI. Conclusion

This paper presents DIO, a generic tool for observing and diagnosing I/O interactions between applications and in-kernel POSIX storage systems. Through a pipeline that automates the process of tracing, filtering, correlating, and visualizing millions of syscalls, and by enriching the information provided

by these with additional context, DIO helps users observing I/O issues while reducing the search space for finding their root cause when, for instance, source code inspection is required.

Our experiments with two widely-used systems show that DIO provides key information for observing erroneous I/O access patterns that lead to data loss, and identifying resource contention in multi-threaded I/O that leads to high tail latency.

## Acknowledgments

This work was financed by the FCT - Portuguese Foundation for Science and Technology, through Ph.D. grant DFA/BD/5881/2020 (*Tânia Esteves*), and realized within the scope of the project POCI-01-0247-FEDER-045924, funded by the ERDF - European Regional Development Fund, through the Operational Programme for Competitiveness and Internationalization and by National Funds through FCT, I.P. within the scope of the UT Austin Portugal Program (*João Paulo*).

## References

[1] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions," in *15th USENIX Conference on File and Storage Technologies (FAST'17)*. USENIX Association, 2017, pp. 149–166, https://www.usenix.org/conference/fast17/technical-sessions/presentation/ganesan.

[2] D. S. Roselli, J. R. Lorch, and T. E. Anderson, "A comparison of file system workloads," in *Proceedings of the USENIX Annual Technical Conference (ATC'00)*. USENIX Association, 2000, https://www.usenix.org/legacy/event/usenix2000/general/full_papers/roselli/roselli.pdf.

[3] H. Daoud and M. R. Dagenais, "Performance analysis of distributed storage clusters based on kernel and userspace traces," *Software: Practice and Experience*, vol. 51, no. 1, pp. 5–24, 2021, https://doi.org/10.1002/spe.2889.

[4] T. Esteves, F. Neves, R. Oliveira, and J. Paulo, "CAT: Content-Aware Tracing and Analysis for Distributed Systems," in *22nd International Middleware Conference (Middleware'21)*. ACM, 2021, p. 223–235, https://doi.org/10.1145/3464298.3493396.

[5] A. Saif, L. Nussbaum, and Y.-Q. Song, "IOscope: A Flexible I/O Tracer for Workloads' I/O Pattern Characterization," in *High Performance Computing: ISC High Performance 2018 International Workshops*. Springer, 2018, pp. 103–116, https://doi.org/10.1007/978-3-030-02465-9_7.

[6] "Jaeger: open source, end-to-end distributed tracing," Accessed on May, 2023. [Online]. Available: https://www.jaegertracing.io

[7] "Zipkin," Accessed on May, 2023. [Online]. Available: https://zipkin.io

[8] S. J. Kim, S. W. Son, W.-k. Liao, M. Kandemir, R. Thakur, and A. Choudhary, "IOPin: Runtime profiling of parallel I/O in HPC systems," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, 2012, pp. 18–23, https://doi.org/10.1109/SC.Companion.2012.14.

[9] K. Vijayakumar, F. Mueller, X. Ma, and P. C. Roth, "Scalable I/O tracing and analysis," in *4th Annual Workshop on Petascale Data Storage*. ACM, 2009, pp. 26–31, https://doi.org/10.1145/1713072.1713080.

[10] "strace: Linux syscall tracer," Accessed on May, 2023. [Online]. Available: https://strace.io

[11] M. Gebai and M. R. Dagenais, "Survey and Analysis of Kernel and Userspace Tracers on Linux: Design, Implementation, and Overhead," *ACM Computing Surveys*, vol. 51, no. 2, pp. 1–33, 2018, https://doi.org/10.1145/3158644.

[12] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," in *Proceedings of the Winter 1993 USENIX Conference*, vol. 46. USENIX Association, 1993, pp. 259–269, https://www.usenix.org/conference/usenix-winter-1993-conference/bsd-packet-filter-new-architecture-user-level-packet.

[13] M. Desnoyers and M. R. Dagenais, "The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux," in *Ottawa Linux Symposium (OLS)*, vol. 2006. Citeseer, 2006, pp. 209–224, https://www.kernel.org/doc/ols/2006/ols2006v1-pages-209-224.pdf.

[14] "Sysdig," Accessed on May, 2023. [Online]. Available: https://github.com/draios/sysdig/

[15] I. U. Akgun, G. Kuenning, and E. Zadok, "Re-animator: Versatile high-fidelity storage-system tracing and replaying," in *13th ACM International Systems and Storage Conference (SYSTOR'20)*. ACM, 2020, pp. 61–74, https://doi.org/10.1145/3383669.3398276.

[16] "Tracee: Linux Runtime Security and Forensics using eBPF," Accessed on May, 2023. [Online]. Available: https://github.com/aquasecurity/tracee

[17] Z. Ren, C. Liu, X. Xiao, H. Jiang, and T. Xie, "Root cause localization for unreproducible builds via causality analysis over system call tracing," in *34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19)*. IEEE, 2019, pp. 527–538, https://doi.org/10.1109/ASE.2019.00056.

[18] S. Yoo, J. Jo, B. Kim, and J. Seo, "LongLine: Visual analytics system for large-scale audit logs," *Visual Informatics*, vol. 2, no. 1, pp. 82–97, 2018, https://doi.org/10.1016/j.visinf.2018.04.009.

[19] D. N. Jha, G. Lenton, J. Asker, D. Blundell, and D. Wallom, "Holistic Runtime Performance and Security-aware Monitoring in Public Cloud Environment," in *22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid'22)*. IEEE, 2022, pp. 1052–1059, https://doi.org/10.1109/CCGrid54584.2022.00128.

[20] E. B.V., "Elasticsearch: The heart of the free and open Elastic Stack," Accessed on May, 2023. [Online]. Available: https://www.elastic.co/elasticsearch/

[21] ——, "Kibana: Your window into the Elastic Stack," Accessed on May, 2023. [Online]. Available: https://www.elastic.co/kibana/

[22] "Fluent Bit: An End to End Observability Pipeline," Accessed on May, 2023. [Online]. Available: https://fluentbit.io

[23] Facebook, "RocksDB: A Persistent Key-value Store for Fast Storage Environments," Accessed on May, 2023. [Online]. Available: https://rocksdb.org

[24] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona, "SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores," in *USENIX Annual Technical Conference (ATC'19)*. USENIX Association, 2019, pp. 753–766, https://www.usenix.org/conference/atc19/presentation/balmau.

[25] "db_bench," Accessed on May, 2023. [Online]. Available: https://github.com/facebook/rocksdb/wiki/Benchmarking-tools

[26] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*. ACM, 2010, p. 143–154, https://doi.org/10.1145/1807128.1807152.

[27] J. Dean and L. A. Barroso, "The Tail at Scale," *Communications of the ACM*, vol. 56, no. 2, p. 74–80, feb 2013, https://doi.org/10.1145/2408776.2408794.

[28] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble, "Tales of the Tail: Hardware, OS, and Application-Level Sources of Tail Latency," in *ACM Symposium on Cloud Computing*. ACM, 2014, p. 1–14, https://doi.org/10.1145/2670979.2670988.

[29] S. Snyder, P. Carns, K. Harms, R. Ross, G. K. Lockwood, and N. J. Wright, "Modular HPC I/O characterization with Darshan," in *5th Workshop on Extreme-Scale Programming Tools (ESPT@SC'16)*. IEEE, 2016, pp. 9–17, https://doi.org/10.1109/ESPT.2016.006.

[30] M. I. Naas, F. Trahay, A. Colin, P. Olivier, S. Rubini, F. Singhoff, and J. Boukhobza, "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)*. ACM, 2021, pp. 1–11, https://doi.org/10.1145/3439839.3458731.

[31] I. Kohyarnejadfard, D. Aloise, M. R. Dagenais, and M. Shakeri, "A framework for detecting system performance anomalies using tracing data analysis," *Entropy*, vol. 23, no. 8, p. 1011, 2021, https://doi.org/10.3390/e23081011.