IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 11 September 2023, accepted 25 September 2023, date of publication 4 October 2023,
date of current version 11 October 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3322104

RESEARCH ARTICLE

Toward a Practical and Timely Diagnosis of Application's I/O Behavior

TÂNIA ESTEVES, RICARDO MACEDO, RUI OLIVEIRA, AND JOÃO PAULO
INESC TEC—Institute for Systems and Computer Engineering, Technology and Science, 4200-465 Porto, Portugal
Department of Informatics, University of Minho, 4710-057 Braga, Portugal

Corresponding author: Tânia Esteves (tania.c.araujo@inesctec.pt)

This work was supported by FCT - Portuguese Foundation for Science and Technology through the Ph.D. grant DFA/BD/5881/2020 and realized within the scope of the project LA/P/0063/2020.

ABSTRACT We present DIO, a generic tool for observing inefficient and erroneous I/O interactions between applications and in-kernel storage backends that lead to performance, dependability, and correctness issues. DIO eases 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 four production-level applications. Results show that DIO enables diagnosing inefficient I/O patterns that lead to poor application performance, unexpected and redundant I/O calls caused by high-level libraries, resource contention in multithreaded I/O that leads to high tail latency, and erroneous file accesses that cause data loss. Moreover, through a detailed evaluation, we show that, when comparing DIO's inline diagnosis pipeline with a similar state-of-the-art solution, our system captures up to 28*x* more events while keeping tracing performance overhead between 14% and 51%.

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].[1]

Due to human error, lack of detailed knowledge on how to efficiently and correctly access the storage backend or usage of high-level libraries that obfuscate the actual POSIX requests being made, developers often implement applications that exhibit: *i)* costly access patterns, such as small-sized I/O requests or random accesses; *ii)* redundant operations, such as unnecessarily re-opening and closing a given file; *iii)* I/O contention caused by having concurrent requests accessing shared storage resources; and *iv)* 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, ranging 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 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], [4], [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 achieve non-intrusive and comprehensive

The associate editor coordinating the review of this manuscript and approving it for publication was Alberto Cano.

[1]By "*in-kernel storage backend*", we refer to storage solutions that expose a kernel-based POSIX interface for user-space applications to persist and access data.

I/O diagnosis for applications using in-kernel POSIX storage backends (*e.g.*, file system, Linux block device).[2] Doing so requires overcoming the following limitations.

(A) *Intrusiveness*: The collection of information about I/O requests is often done through source code instrumentation [7], [8], [9], [10]. This approach is not easily applicable across different applications as it requires users to manually analyze and instrument distinct and potentially large codebases (*e.g.*, RocksDB has approximately 440K lines of code written in six different programming languages).

(B) *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 [11], 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], [12]. This challenge motivated the emergence of technologies such as eBPF [13] and LTTng [14], 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.

(C) *Lack of analysis pipeline*: While efficient I/O tracing is an important step for profiling applications, by itself it is not sufficient, given the large number 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 multithreaded I/O). Several solutions only cover the tracing collection step, delegating these other time-consuming tasks to users [11], [15], [16], [17].

(D) *Flexibility*: Solutions offering a complete pipeline for application diagnosis are designed for rigid analysis scenarios, such as detecting unreproducible builds [18], observing file offset access patterns [5], or identifying security issues [19], [20]. Thus, for multipurpose 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, such as those identified at §IV.

This paper proposes DIO, a generic tool for observing and diagnosing applications' storage I/O. It addresses the aforementioned challenges with the following contributions:

[2]This is an expanded version of the work published in [6]. We improved our earlier publication by further detailing DIO's design to emphasize the tool's relevance, introducing two new use cases with production-level applications that demonstrate the tool's readiness and relevance, and providing new experiments that evaluate and compare DIO with related solutions.

A. 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. By operating at kernel-space, DIO is able to intercept syscalls submitted by any application that makes POSIX requests to the storage backend. The tracer supports 42 storage-related syscalls and records a comprehensive set of information for each operation, including its type, arguments, return value, timestamp, ProcessID (PID), and ThreadID (TID). By offering a flexible design, DIO allows collecting only events of interest, filtering them (at kernel-level) by syscall type, PID, TID, or file paths. This enables narrowing the tracing scope according to users' requirements, reducing the size of the stored trace, and minimizing performance overhead over the targeted application.

B. ENRICHED ANALYSIS

DIO enriches data gathered for each syscall with additional context available at the kernel (*e.g.,* process name, file type, 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.

C. ASYNCHRONOUS EVENT HANDLING

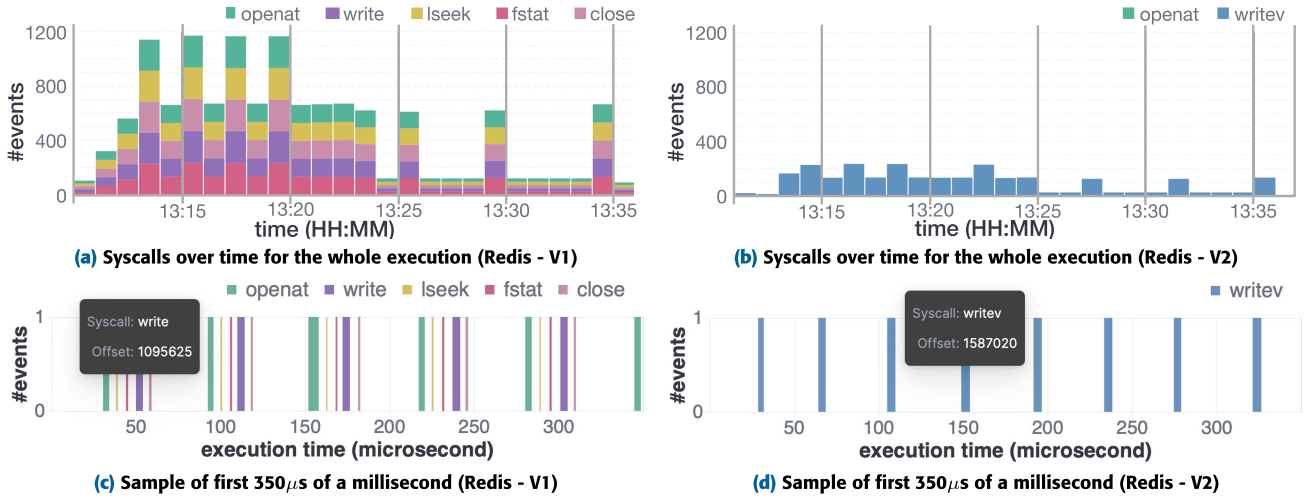
Only syscall interception is done synchronously, while traced events are collected and processed in user-space asynchronously. This avoids adding extra latency in the critical path of I/O requests and enables practical analysis of data-intensive storage workloads.

D. NEAR REAL-TIME PIPELINE

DIO offers a practical and customizable pipeline so that users can create their own queries, correlation algorithms, and dashboards to analyze collected data. The pipeline follows an inline approach, meaning that traced events are automatically parsed and forwarded to the analysis and visualization components as soon as they are collected in user-space, without requiring manual user intervention.

Contrarily to state-of-the-art and widely used syscall tracers, like Strace [11] and Sysdig [15], DIO collects more comprehensive information about I/O requests (*e.g.*, DIO is the only tool capturing the file offset for `read` and `write` syscalls), and provides an integrated analysis pipeline which allows users to query and visualize captured data in a practical and near real-time fashion.

DIO is implemented as an open-source prototype using eBPF [13], Elasticsearch [21], and Kibana [22], and validated with production-level systems. Results show that DIO enables the diagnosis of *i)* inefficient use of syscalls that lead to poor storage performance in Redis, *ii)* unexpected file access patterns caused by the usage of high-level libraries that



(a) Syscalls over time for the whole execution (Redis - V1)

(b) Syscalls over time for the whole execution (Redis - V2)

(c) Sample of first 350μs of a millisecond (Redis - V1)

(d) Sample of first 350μs of a millisecond (Redis - V2)

FIGURE 1. Log file access pattern, depicting syscalls issued within second and microsecond resolutions, for Redis's version including inefficient I/O patterns ((a) and (c)) and for the fixed version ((b) and (d)).

lead to redundant I/O calls in Elasticsearch, *iii)* erroneous file accesses that cause data loss in Fluent Bit, and *iv)* resource contention in multi-threaded I/O that leads to high tail latency for user workloads in RocksDB.

Moreover, we conduct a thorough experimental evaluation that highlights the different trade-offs in terms of performance overhead, resource usage, and tracing accuracy when using different tracing modes and configurations provided by DIO while validating our solution against two state-of-the-art syscall-based tracers: Strace [11] and Sysdig [15]. Results show that when compared with an inline diagnosis pipeline using Sysdig, DIO provides timely analysis for users and improves the amount of captured events by up to 28x while keeping performance overhead between 14% and 51%.

All artifacts discussed in this paper, including DIO, workloads, scripts, and the corresponding analysis and visualization outputs, are publicly available at https://github.com/dsrhaslab/dio.

The rest of the paper is structured as follows. §II motivates the need for a tool like DIO, and §III describes its design and implementation details. §IV and §V evaluate DIO qualitatively (with three more use cases) and quantitatively (under intensive I/O workloads). §VI and §VII discuss related work and conclude the paper.

II. MOTIVATION

To showcase the benefits that integrated syscall tracing, analysis, and visualization bring towards validating inefficient I/O behavior from applications, let us consider a previously known issue identified in the Redis in-memory data store [23]. Specifically, the server log file is repeatedly opened and closed for every written line, which adds extra latency for log operations and can potentially slow down Redis performance.[3]

To identify this behavior, users could run a workload on top of Redis and trace the syscalls submitted to kernel. In this example, we used *redis-benchmark* to generate 5M requests to the database, which yield >200M syscalls.[4] Inspecting these events without proper filtering, correlation, and visualization mechanisms is a non-trivial and time-consuming task.

In this paper, we argue that an analysis pipeline integrating the previous mechanisms would greatly simplify users' work. In particular, for this specific use case, by intercepting only the syscalls submitted to the file system, while discarding Redis's `read` and `write` operations for network sockets, one would just need to trace ≈600K storage-related syscalls (*i.e.,* ≈0.3% of the original tracing sample).

Then, through correlation, users could further filter these storage events and explore only the syscalls being directed into the log file reported at the issue. Finally, through visualization, it would be possible to observe the pattern shown in Figs. 1a and 1c. The former shows a set of syscalls being made repeatedly over the log file. The latter depicts a sample of the first 350μs within a millisecond, showing the exact order and duration of the requests for one of these sets (*i.e.*, `openat` → `lseek` → `fstat` → `write` → `close`). By interacting with the latter visualization (*i.e.*, visually exploring the syscalls' arguments), it would also be possible to observe that those syscalls are accessing consecutive file offsets, suggesting a sequential file access pattern.

As suggested in the pull request for the issue, this inefficient I/O pattern can be fixed by: *i)* keeping the log's file descriptor opened while the file is being used, and *ii)* using `writev` to write log lines more efficiently. As depicted in Figs. 1b and 1d, by using the same analysis pipeline, users can also validate the suggested fix, where redundant `open` and `close` operations are avoided, along with the need for

[3] *Logging improvements* issue from Redis' GitHub repository: https://github.com/redis/redis/pull/10531

[4] https://redis.io/docs/management/optimization/benchmarks/

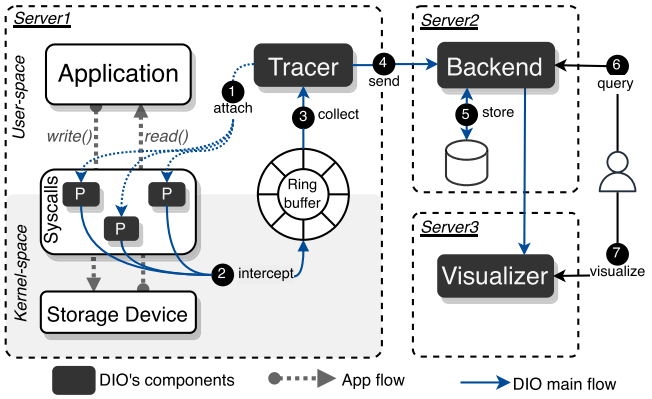


FIGURE 2. DIO's design and flow of events.

using `lseek` before every write operation. Also, `writev` is now used to write log lines instead of `write`.

The aforementioned visualizations are real outputs of using DIO for this use case (available at https://dio-tool.netlify.app/use-cases/redis). Next, we further detail the proposed solution, while in §IV, we show that it can be used to discover other types of undesired I/O behaviors, observe erroneous file access patterns that cause data loss, and assist with the root cause analysis of applications exhibiting high tail latencies.

III. THE DIO TOOL

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

Transparency and reduced overhead: 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. Moreover, DIO uses tracing technologies that allow minimizing the extra processing done in the critical path of I/O requests to reduce the performance overhead imposed over targeted applications.

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

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`, `read`), metadata (*e.g.*, `openat`, `stat`), extended attributes (*e.g.*, `getxattr`, `setxattr`), and directory management (*e.g.*, `mknod`, `mknodat`) requests. Users can choose to capture only the relevant ones for their analysis goals and further filter them based on targeted PIDs, TIDs, and file paths. Moreover, two tracing modes (§III-C) are provided that allow users to configure the amount of detail collected for each I/O syscall. These tracing modes and filters allow minimizing the performance impact and storage overhead (*i.e.*, the size of traced data) imposed by DIO.

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 data exploration and building customized visualizations.

A. SYSTEM OVERVIEW

DIO consists of three main components, namely the *Tracer*, the *Backend*, and the *Visualizer*, as depicted in Fig. 2. 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 latter persists and indexes events (❺) and allows users to query and summarize (*e.g.*, aggregating) stored information (❻). Meanwhile, the *Visualizer* provides near real-time visualization of the traced events by directly querying the *Backend* (❼). Users rely on the *Visualizer* to ease the process of data exploration and 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 [13], which allows instrumenting the Linux kernel by executing small programs (*i.e.*, eBPF programs) whenever a given point of interest (*e.g.*, tracepoint, kernel probe) is called, without requiring the costly syscall context switching from user-space to kernel-space as in other state-of-the-art tracing mechanisms (*e.g.*, the use of `ptrace` by Strace) [4], [12].

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 desired information about the request, including *entry* (*e.g.*, arguments) and *exit* (*e.g.*, return value) related data, 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*.

TABLE 1. 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`

To minimize both network and performance overhead, the *Tracer* groups several events into buckets that are sent and indexed in batches at the *Backend*.

1) SUPPORTED SYSCALLS AND FILTERS

Table 1 depicts the syscalls supported by DIO. Since instrumenting syscalls can introduce extra processing in the critical path of I/O requests, DIO allows users to filter requests by:

(*a*) *type of syscall* - activates only the tracepoints for the provided syscall types.
(*b*) *process name* - captures only syscalls issued by a process with the provided name.
(*c*) *process or thread ID(s)* - captures only syscalls issued by a given list of process or thread IDs.
(*d*) *file or directory path(s)* - captures only syscalls targeting one of the provided file or directory paths.

The flexibility offered by these filters allows users to better configure the *Tracer* according to their goals and balance the tracing accuracy with the storage and performance overheads. Namely, by specifying the targeted syscall types *(a)*, the *Tracer* avoids activating unnecessary tracepoints, thus reducing the amount of I/O requests with extra processing in their critical path. Moreover, by applying the remaining filters (namely, *(b)*, *(c)*, and *(d)*) in kernel-space, DIO reduces the amount of information to be sent and processed in user-space.

C. COLLECTED INFORMATION

For each intercepted syscall, DIO collects information related to the:

- *syscall* - type, arguments, and return value.
- *process* - PID, TID, and process name.
- *time* - *entry* and *exit* timestamps.

Since the amount of captured information can influence the overhead imposed on the targeted application, DIO offers different tracing modes (*raw* and *detailed*) that balance the detail of information captured with the extra processing added to the critical path of I/O requests.

Raw: The less detailed mode (referred to as *raw*) captures the information above without pre-processing it. This means that for arguments referring to memory regions (*e.g.*, `char *pathname` in `stat` syscall, or `char * name` in `getxattr` syscall), only their hexadecimal value is saved (*e.g.*, `"name":"$0\times55555556$feab"`). Also, for numerical arguments (*e.g.*, `int flags` in `openat` syscall, or `int fd` in `write` syscall) no translation is done and their values are saved in their original form (*e.g.*, `"flags":1089`).

By saving information in its raw format, DIO reduces the extra processing in the critical path of I/O requests, the amount of data transferred to user-space, and the information that must be analyzed posteriorly. As shown n §IV-C, there are scenarios where the information provided by this mode is sufficient for diagnosing I/O issues.

Detailed: For a more in-depth analysis, DIO offers a *detailed* mode, which provides comprehensive information about the requests by pre-processing some arguments before saving the events (§III-C1), enriching the traced information with context from the kernel (§III-C2) and translating file descriptors to their corresponding file paths (§III-C3).

1) DATA PRE-PROCESSING

Instead of keeping the collected information in its raw format, the *detailed* mode transforms values into a human-readable form (*e.g.*, `"flags": "O_WRONLY|O_CREAT|O_APPEND"`), simplifying the analysis process done by users. Furthermore, this mode captures the actual memory content for pointer arguments instead of saving their hexadecimal values (*e.g.*, `"name": "system.posix_acl_access"`).

For buffers being handled by data-related syscalls (*e.g.*, `void *buf` in `read` syscall), the *detailed* mode provides the option to compute a hash sum of their content (*e.g.*, `"buf": "This is the first log line"` → `"signature": 114a83d4`). This way, DIO compacts the amount of information that reaches the analysis pipeline, minimizing the storage overhead while still allowing the observation of syscalls handling the same data content (as shown in §IV-B). The hash sum can either be computed in user-space, which requires transferring the buffers' content from kernel, or in the kernel, which reduces the amount of information sent to user-space but adds extra processing to the critical path of I/O requests.

2) ENRICHED INFORMATION

While the aforementioned information already provides valuable insights about applications' I/O behavior, correlating this data with other types of information further enriches and eases the analysis made by users (as discussed in §IV). Therefore, the *detailed* mode leverages eBPF's access to

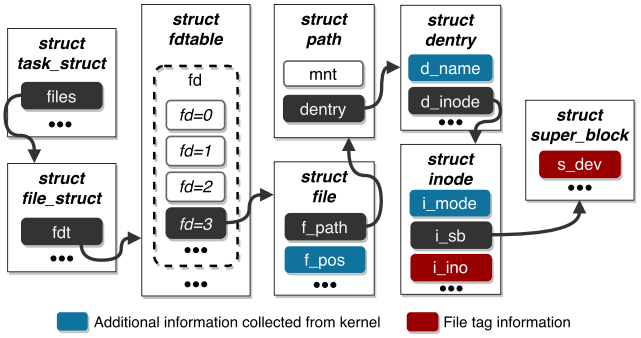


FIGURE 3. Kernel structures used by DIO. *d_name*, *i_mode*, and *f_pos* are used to obtain the file path, offset, and type. *i_ino* and *s_dev* are used to create a unique file tag.

kernel structures (as depicted in Fig. 3) and complements traced information with:

- The *file path* being accessed by syscalls. Since many syscalls access files through a file descriptor (*e.g.,* `read`, `close`, `fgetxattr`), obtaining the corresponding file path provides more specific information about the file being handled.
- The *file type* targeted by syscalls. This additional information allows differentiating accesses to regular files, directories, sockets, block/char devices, pipes, symbolic links, and others.
- The *file offset* being accessed by data-related syscalls. Information about offsets allows observing file access patterns (*e.g.*, sequential/random accesses), even for syscalls that do not provide the file offset as an argument (*e.g.*, `read`).

3) FILE DESCRIPTOR TRANSLATION

Associating syscalls with their corresponding file paths is fundamental to enabling detailed tracing information. However, this is not a trivial task.

The typical approach to address this challenge is to correlate the file descriptor with the file path argument of the previous `open` call that initialized it (*i.e.*, the `open` syscall that originated the file descriptor). However, this approach is not accurate as there are other mechanisms to obtain a given file descriptor (*e.g.,* through the creation of new processes via `fork`, duplication of file descriptors through `dup`, `dup2`, or `fcntl`).

State-of-the-art solutions using eBPF [4] access kernel structures to find the file path corresponding to a specific file descriptor. However, transferring file paths from kernel to user-space for each event accessing a file induces significant tracing overhead and leads to potential loss of traced information.

DIO introduces a different approach by creating a custom event (*EventPath*) that contains information about a specific file (*i.e.*, file path, file type) and by sending it only once to user-space. Each *EventPath* is labeled with a unique *file tag*, which is then associated with any syscall accessing that specific file.

Algorithm III.1 EventPath and File Tag Generation

Input:
fd: file descriptor
curTimestamp: current timestamp
openedInodes: list of currently opened inodes
curEvent: current event structure

```
1  Function checkInode(fd, curTimestamp)
2      inodeNo ← getInodeNo(fd)
3      deviceNo ← getDeviceNo(fd)
4      fileTag ← createFileTag(inodeNo, deviceNo)
5      if fileTag not in openedInodes then
6          filePath ← getFilePath(fd)
7          submitNewEventPath(filePath, fileTag)
8          openedInodes.append(fileTag, curTimestamp)
9          curEvent.add(fileTag, curTimestamp)
10     else
11         timestamp ← getTimestamp(fileTag)
12         curEvent.add(fileTag, timestamp)
```

Alg. III.1 further details how DIO creates both the *file tag* and *EventPath* event. First, whenever an intercepted syscall accesses a file through a file descriptor, DIO goes through Linux kernel structures (as depicted in Fig. 3) and obtains information about the file's inode number and the file system's device number (L2-L3). DIO relies on the assumption that every inode has a unique number inside the same file system. Thus, by combining the inode number with the device number, DIO can create a unique file tag (L4).

After generating the *file tag*, DIO verifies if it is included in the list of opened inodes (*i.e.,* inodes accessed during the tracing execution). If the list does not contain the current *file tag*, DIO calculates the *file path* for the current file descriptor (L6) and sends to user-space a new *EventPath*, containing the file path, type, and tag (L7). The *file tag* is then added to the list of opened inodes (L8) and associated with the current event being handled (L9). If the *file tag* already exists in the list of opened inodes, it only needs to associate the *file tag* to the current event being handled (L11-L12). Finally, when an inode is destroyed, the corresponding *file tag* is removed from the list of opened inodes.

Since inodes are recycled over time, DIO distinguishes reused inode by assigning to each file tag the timestamp of the first captured access to that inode, and in user-space, each event will have associated a tag composed of the device number, inode number, and timestamp *(e.g.*, `"file_tag": "7340032|12|6707719730779287"`).

This approach minimizes the amount of redundant data transferred between kernel and user-space and, consequently, the storage and performance overheads. Moreover, even if an *EventPath* is lost, which prevents the association of the file

path to the event, a file access pattern analysis would still be possible by using the *file tag*. §III-D further describes how each event is correlated to the corresponding file path through the generated *file tags* and *EventPaths*.

D. BACKEND

The *Backend* allows persisting, searching, and analyzing data from traced events. It uses the Elasticsearch [21] 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/app/log.txt`).

Algorithm III.2 File Path Correlation Algorithm

```
   Input:
       sysEvents: list of events with a file tag
       evtPaths: list of file paths
1  Function correlateFP(sysEvents, evtPaths)
2      for sys ← sysEvents do
3          for path ← evtPaths do
4              if sys.FileTag = path.FileTag then
5                  sys.FilePath ← path.FilePath
6                  sys.FileType ← path.FileType
```

Alg. III.2 shows the file path correlation performed by the *Backend*. The algorithm receives two lists as arguments: *i)* the syscalls events (*sysEvents*), and *ii)* all *EventPath* events (*evtPaths*) generated during the Tracer execution. By relying on the unique *file tags*, the algorithm matches each syscall event with the corresponding *EventPath* (L2-L4), updating the former with the file path and type information (L5-L6).

E. 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 [22], 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.

Nanosecond visualization: The minimum time unit supported by Kibana for visualization is restricted to the millisecond. This prevents users from observing the order and time spread for requests occurring in sub-millisecond time windows, which occur frequently when using modern storage and network hardware (*e.g.,* NVMe, persistent memory, RDMA). Namely, data-intensive applications generate hundreds to thousands of I/O operations per second. Consequently, many of these operations can occur within the same millisecond. As shown in §II and IV-A, observing the order and time spreading of requests at a smaller time interval (*i.e.*, microsecond or nanosecond) is important to diagnose applications' I/O by allowing, for instance, to visualize duplicate syscalls or to understand the sequence and duration of syscalls made in such a short time window. Therefore, we designed a new representation that depicts I/O events order and time spacing at the nanosecond time unit resolution (Figs. 1c, 1d and 4b). It is fully integrated with Kibana's dashboards and automatically queries the *Backend* to collect the required data.

F. 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 in the kernel, and sending it to the *Backend*. 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. The nanosecond representation is implemented with the Vega-lite [24] (v4) visualization grammar and provided along with DIO's predefined dashboards.

G. 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 server, to avoid negatively impacting the performance of the targeted application (*e.g.*, additional resource consumption), the analysis pipeline can be installed on separate servers (Fig. 2). 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.[5]

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* only to record events that target them.

Moreover, users can choose between *raw* or *detailed* tracing modes and further configure the latter to deactivate the collection of arguments that require transferring large amounts of data to user-space. Namely, data buffers' content and file paths obtained from syscall arguments may only be relevant to some specific cases and, therefore, can be collected/ignored according to the analysis goals. As we show in the next section, data buffers are irrelevant for use cases §II, §IV-A and §IV-C, while the file paths obtained from syscall such as `lstat` or `unlink` are only relevant for use cases §IV-A and §IV-B.

All these configurations, along with the analysis pipeline's parameters (*e.g.*, Elasticsearch URL), can be set through a configuration file.

IV. USE CASES

Our evaluation showcases how DIO eases the process of observing and validating known issues or exploring unknown applications and finding potential problems. To this end, besides the Redis use case discussed at §II, we analyzed three additional production-level applications: Elasticsearch, RocksDB, and Fluent Bit. Results show that DIO :

- provides valuable information about applications' I/O requests that can be used to uncover or confirm inefficient (§II) or unexpected (§IV-A) I/O patterns;
- is a practical tool for validating the root causes of correctness (§IV-B) and performance (§IV-C) issues, without instrumenting large codebases.

Except for Fig. 7, all the remaining figures in this section were generated with DIO (with minimal modifications for readability). The full set of DIO's visual representations is available at https://dio-tool.netlify.app.

[5]Multiple instances of DIO's tracer can be deployed to diagnose distributed applications across the servers where their components are running. Each instance of DIO's tracer will generate an independent tracing index at the same *Backend* (containing information about the targeted host), allowing for later analysis and correlation of each tracing execution.

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 (used for storing tracing data), and a 512 GiB SATA SSD (used for hosting the datasets). DIO's *Backend* and *Visualization* components run on two separate servers, both equipped with a 6-core Intel i5-9500, 16 GiB of memory, and a 250 GiB NVMe SSD.

Workloads: Both benchmarks and custom workloads were selected to reproduce specific but realistic interactions between the targeted applications and underlying storage backends. As shown next, these workloads validate that DIO can be used to explore the I/O patterns of real applications and identify the root cause of real known issues. Further details of the workloads and benchmark configurations are provided along with each use case.

A. TOP-DOWN EXPLORATION AND DIAGNOSIS OF APPLICATIONS' I/O

Next, we show how DIO can be used to explore and obtain additional insight into the I/O behavior of applications and then, by following a top-down approach, how one can use our solution to diagnose inefficient file access patterns.

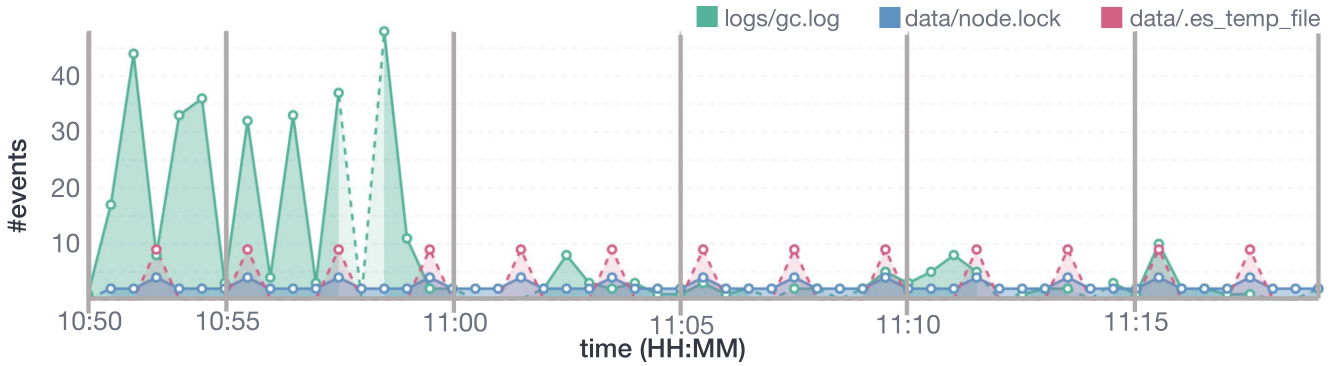
We chose Elasticsearch [21] (v8.3.0), a distributed search and analytics engine, as the targeted application.[6] Due to the use case's exploratory nature, DIO was configured to capture all supported syscalls. We used the *Rally* benchmark with the default workload (*geonames*) to generate load for Elasticsearch.[7] This workload indexes ≈11M documents and executes different queries (*e.g.*, filter, sort).

Under this workload, and with the help of the information collected and organized by DIO, we observed that Elasticsearch generates >1M storage-related syscalls, 99.7% of them targeting regular files and the remaining ones targeting directories. Elasticsearch uses mainly data-related operations (88%), most of them being `write` (71%), `pread64` (7%) and `read` (5%). Further, it spawns a total of 42 processes and 118 threads while accessing almost 4000 files.

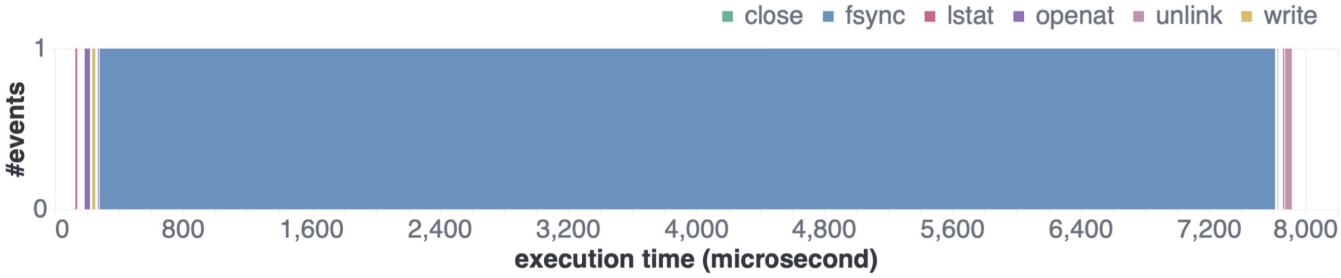
As depicted in Fig. 4a, some files exhibit a constant access pattern, even in the absence of client requests. Namely, every 30 seconds, Elasticsearch submits 2 syscalls to the `node.lock` file (■ line), and every 2 minutes, 9 syscalls to `.es_temp_file` (■ line). For the latter, DIO's nanosecond visualization (Fig. 4b) uncovered an unexpected duplication of `openat` (■) and `close` (■) syscalls. Listing 1 shows the syscalls, along with the corresponding arguments and return values, observable with DIO. Listing 2 shows Elasticsearch's Java source code responsible for accessing the `.es_temp_file`. The first `openat` is generated by the `Files.newOutputStream` method (❶), which opens an output stream used for writing data to the file (❷).

[6]Note that in this section, Elasticsearch is used as the targeted application and should not be confused with the one used to implement DIO's *Backend*.

[7]https://esrally.readthedocs.io/en/stable/install.html



(a) Sample of Elasticsearch's file access pattern.



(b) Sequence of system calls issued by Elasticsearch for file ".es_temp_file" (nanosecond visualization).

FIGURE 4. Elasticsearch's file access pattern.

```
❶ openat(".es_temp_file", "O_WRONLY|O_CREAT", ↩
      ...) = 53

❷  write(53, ...) = 22

❸ openat(".es_temp_file", "O_WRONLY", ...) = 56
  fsync(56) = 0
  close(56) = 0

❹ close(53) = 0
  lstat(".es_temp_file", ...) = 0
  unlink(".es_temp_file") = 0
```

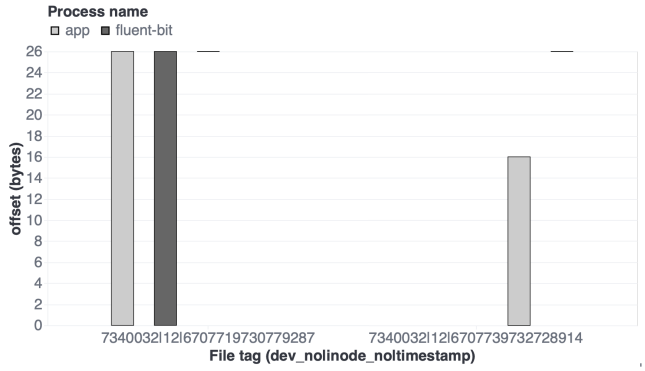
LISTING 1. System calls observable with DIO.

```
class FsHealthMonitor implements Runnable {
    static final String TEMP_FILE_NAME = ↩
        ".es_temp_file";
    ...
    private void monitorFSHealth() {
        ...
        final Path tempDataPath = path.resolve(↩
            TEMP_FILE_NAME);
        Files.deleteIfExists(tempDataPath);
    ❶   try (OutputStream os = Files.↩
            newOutputStream(tempDataPath, ↩
            StandardOpenOption.CREATE_NEW)) {
        ❷   os.write(bytesToWrite);
        ❸   IOUtils.fsync(tempDataPath, false);
        }
    ❹   Files.delete(tempDataPath);
        ...
    }
}
```

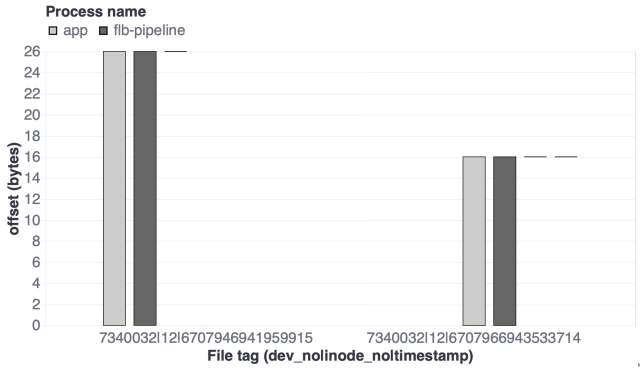
LISTING 2. Elasticsearch source code.

An `IOUtils.fsync` method is then invoked to flush dirty pages to disk (❸). However, rather than using the already opened file descriptor, created from the first `openat` call, it internally reopens and closes the file again. Finally, upon the file's removal request (❹), three syscalls are issued: a `close` corresponding to the first `openat`, a `lstat`, and an `unlink`. The information provided by DIO is relevant for identifying the file where this pattern happens (`.es_temp_file`) and, therefore, reducing the search space through the application's source code from >2.5M LoC to a single Java class with 195 LoC.[8]

[8]https://github.com/elastic/elasticsearch/blob/91413fbd685ba022648abf2e8a0e291665a15a1b/server/src/main/java/org/elasticsearch/monitor/fs/FsHealthService.java#L130



(a) Accessed file offsets for Fluent Bit (v1.4.0).



(b) Accessed file offsets for Fluent Bit (v2.0.5).

time	proc_name	syscall	ret val	file_tag(dev_no|inode_no|timestamp)	signature	offset	
1,686,303,678,731,120,896	app	openat	3	7340032|12|6707719730779287	-	-	
1,686,303,678,731,331,840	app	write	26	7340032|12|6707719730779287	114a83d4	0	1
1,686,303,678,731,625,472	app	close	0	7340032|12|6707719730779287	-	-	
1,686,303,683,518,010,624	fluent-bit	openat	23	7340032|12|6707719730779287	-	-	
1,686,303,683,523,416,320	fluent-bit	read	26	7340032|12|6707719730779287	114a83d4	0	2
1,686,303,683,525,850,112	fluent-bit	read	0	7340032|12|6707719730779287	a9458298	26	
1,686,303,688,732,088,576	app	unlink	0	-	-	-	3
1,686,303,688,732,712,448	fluent-bit	close	0	7340032|12|6707719730779287	-	-	
1,686,303,698,732,821,248	app	openat	3	7340032|12|6707739732728914	-	-	
1,686,303,698,732,984,576	app	write	16	7340032|12|6707739732728914	4a6ab3ef	0	4
1,686,303,698,733,085,696	app	close	0	7340032|12|6707739732728914	-	-	
1,686,303,703,517,364,224	fluent-bit	openat	23	7340032|12|6707739732728914	-	-	
1,686,303,703,517,946,112	fluent-bit	lseek	26	7340032|12|6707739732728914	-	-	
1,686,303,703,518,199,040	fluent-bit	read	0	7340032|12|6707739732728914	a9458298	26	5
1,686,303,718,755,793,920	fluent-bit	close	0	7340032|12|6707739732728914	-	-	

(c) Fluent Bit (v1.4.0) erroneous access pattern.

time	proc_name	syscall	ret val	file_tag(dev_no|inode_no|timestamp)	signature	offset	
1,686,303,905,942,310,656	app	openat	3	7340032|12|6707946941959915	-	-	
1,686,303,905,942,499,328	app	write	26	7340032|12|6707946941959915	114a83d4	0	1
1,686,303,905,942,590,464	app	close	0	7340032|12|6707946941959915	-	-	
1,686,303,910,003,282,688	flb-pipeline	openat	46	7340032|12|6707946941959915	-	-	
1,686,303,910,008,165,376	flb-pipeline	read	26	7340032|12|6707946941959915	114a83d4	0	2
1,686,303,910,008,785,664	flb-pipeline	read	0	7340032|12|6707946941959915	a9458298	26	
1,686,303,915,943,073,536	app	unlink	0	-	-	-	3
1,686,303,915,946,068,992	flb-pipeline	close	0	7340032|12|6707946941959915	-	-	
1,686,303,925,943,623,424	app	openat	3	7340032|12|6707966943533714	-	-	
1,686,303,925,943,786,240	app	write	16	7340032|12|6707966943533714	4a6ab3ef	0	4
1,686,303,925,943,856,896	app	close	0	7340032|12|6707966943533714	-	-	
1,686,303,930,000,810,752	flb-pipeline	openat	46	7340032|12|6707966943533714	-	-	
1,686,303,930,001,666,048	flb-pipeline	read	16	7340032|12|6707966943533714	4a6ab3ef	0	
1,686,303,930,001,923,328	flb-pipeline	read	0	7340032|12|6707966943533714	a9458298	16	5
1,686,303,930,002,521,856	flb-pipeline	read	0	7340032|12|6707966943533714	a9458298	16	
1,686,303,946,000,598,016	flb-pipeline	close	0	7340032|12|6707966943533714	-	-	

(d) Fluent Bit (v2.0.5) correct access pattern.

FIGURE 5. Fluent bit erroneous access pattern leading to data loss.

This behavior shows that applications' methods can be translated into multiple syscalls by the high-level libraries these are using. For I/O-intensive files, this duplication may lead to performance degradation and I/O contention at the storage backend [25].

This use case shows two important features of DIO. First, how it aids in exploring I/O interactions between applications and storage backends. Second, how it can be used to find unexpected I/O patterns being issued by applications and help users narrow the portion of source code that must be inspected to fix such patterns.

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 [26]. Existing issues report that data is lost when using the `tail` input plugin, which is used to fetch new content being added to log files.[9,10] 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.[9] DIO was used to simultaneously trace and analyze the client program and Fluent Bit by filtering the syscalls belonging to these applications' processes.

Fig. 5a shows a visualization generated by DIO representing the accessed offsets for the `app.log` file for both client (*app*) and Fluent Bit (*fluent-bit*) applications. This visual representation shows that: *i)* two files are being accessed (different file tags); *ii)* the first file is accessed by both *app* and *fluent-bit* applications from offset 0 to offset 26; and *iii) app* accesses the second file from offset 0 to offset 16, but *fluent-bit* only accesses the offset 26. Complementing this information with the one provided by the tabular visualization of Fig. 1c (also generated by DIO), one can further understand these file accesses. The *app* program 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). The hash *signatures* at the table validate that *fluent-bit* reads exactly the same content as written by *app*. 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

[9]https://github.com/fluent/fluent-bit/issues/1875
[10]https://github.com/fluent/fluent-bit/issues/4895

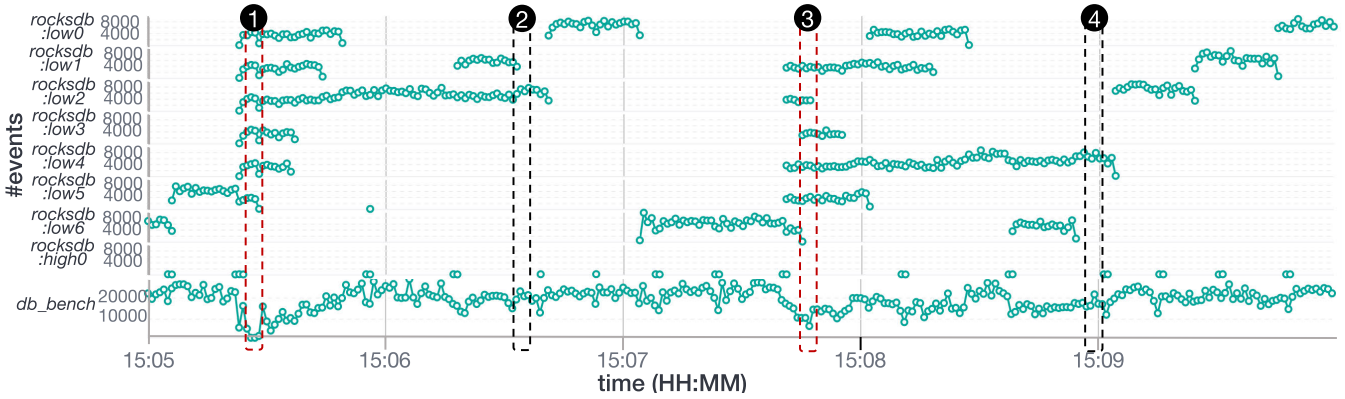


FIGURE 6. Syscalls issued by RocksDB over time, aggregated by thread name. `db_bench` includes the 8 client threads, `rocksdb:low[0-6]` refers to each compaction thread, and `rocksdb:high0` refers to the flush thread.

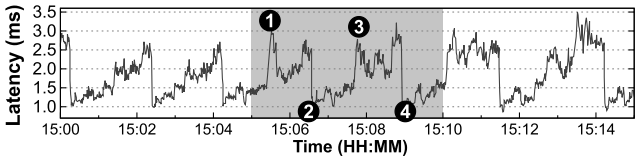


FIGURE 7. 99th percentile latency for RocksDB client operations.

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. Note that the hash signatures are different for the content written by *app* and read by *fluent-bit*.

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 in 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 access pattern, we used DIO to analyze a more recent version of Fluent Bit (v2.0.5), where fixes were applied to avoid this data loss issue. Figs. 1b and 5d show similar visualizations for the fixed version. While the erroneous and correct versions present similar initial 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*. In the correct version, the hash signatures for the 16 bytes written by *app* and read by *fluent-bit* match.

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 of high tail latency at client requests issued to RocksDB, an embedded key-value store (KVS) [27].

This phenomenon was first observed in SILK [28] and, therefore, we followed the same testing methodology to reproduce it. We used the *db_bench* benchmark configured with 8 client threads performing a mixture of read-write requests in a closed loop (YCSB A [29]).[11] RocksDB was configured with 8 background threads, namely 1 for flushes and 7 for compactions. Fig. 7 reports a sample of a 5-hour-long execution and depicts the 99th 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 of 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

[11] https://github.com/facebook/rocksdb/wiki/Benchmarking-tools

trace, analyze, and visualize RocksDB execution, as depicted in Fig. 6. 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` refers 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. 6 and 7 (in intervals ❶ and ❸, at least 5 compaction threads submit requests). When fewer compaction threads perform I/O, the performance of `db_bench` 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 compactions ($L_0 \rightarrow L_1$), these can be made in parallel. A common problem with compactions, however, is the interference between I/O workflows, which generates 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 [30], [31], and that can be observed with DIO without any code instrumentation.

D. PERFORMANCE IMPACT AND I/O EVENTS HANDLING

We now analyze the performance impact induced by diagnosing I/O calls with DIO.

DIO's setups: For these experiments, we configured DIO to capture only the required information for diagnosing the aforementioned I/O issues. Namely, the RocksDB use case (§IV-C) requires information about the type and number of syscalls, their timestamp, and the name of the process(es) that issued them. Thus, DIO can be configured with the less detailed tracing mode (*raw*).

For the Redis use case (§II), we also need to collect information about file paths and file offsets, thus requiring DIO's detailed tracing mode. Since the syscalls relevant for this use case all handle file descriptors, we can avoid collecting the syscalls arguments that require transferring large amounts of data from kernel to user-space, as explained in §III-G (*i.e.*, file paths contained in the syscalls' arguments and data buffers' content). We refer to this setup as $detailedP_{fds}$.

Elasticsearch use case (§IV-A), on the other hand, requires the analysis of syscalls whose file path information is obtained from their arguments. Thus, for this use case, we configure DIO with the $detailedP_{all}$ setup, which also collects the file paths from syscall arguments.

Lastly, the Fluent Bit use case (§IV-B) was configured with the $detailedP_{all}C_{khash}$, which also captures the data buffers' content and calculates a hash sum in kernel-space, which is helpful for validating when both applications are processing the same data content.

TABLE 2. Minimum DIO's tracing mode for successfully diagnosing each use case.

	raw	$detailedP_{fds}$	$detailedP_{all}$	$detailedP_{all}C_{khash}$
RocksDB	✓✓	✓	✓	✓
Redis	✗	✓✓	✓	✓
Elasticsearch	✗	✗	✓✓	✓
Fluent Bit	✗	✗	✗	✓✓

Table 2 shows DIO's minimal configuration for successfully diagnosing each use case (✓✓) while pointing to more comprehensive configurations that also allow observing these (✓). §V-A1 provides further details about DIO's setups.

Fig. 8 shows the execution times of Elasticsearch, Redis, and RocksDB under the workloads described at §IV-A, §II, and §IV-C, respectively. Fluent Bit's use case was excluded as it does not include a benchmark. For each application, we compared its *vanilla* deployment (*i.e.*, without tracing its execution) with DIO and two state-of-the-art syscall tracers: Strace [11] and Sysdig [15].

Performance analysis: The performance overhead imposed over *vanilla* setups is influenced by the I/O load generated by each application. For Elasticsearch, the least I/O intensive application, all tracers introduce negligible overhead, increasing the *vanilla* execution time (73.31 min) by up to 82 seconds.

For Redis, DIO, Sysdig and Strace increase *vanilla* execution time (23.5 min) by 1.04x (24.0 min), 1.62x (37.3 min), and 4.86x (111.9 min), respectively. By filtering events to Redis' working directory, Sysdig and DIO can discard non-storage related requests (*i.e.*, `read` and `write` syscalls issued to network sockets), which account for ≈99% of the events generated by Redis (as shown in §II). However, by applying these filters in kernel-space, DIO reduces the computation in the critical path of I/O requests and the amount of data sent to user-space, imposing less overhead than Sysdig. Strace cannot filter events by directory paths (only by specifying all file paths) and, consequently,

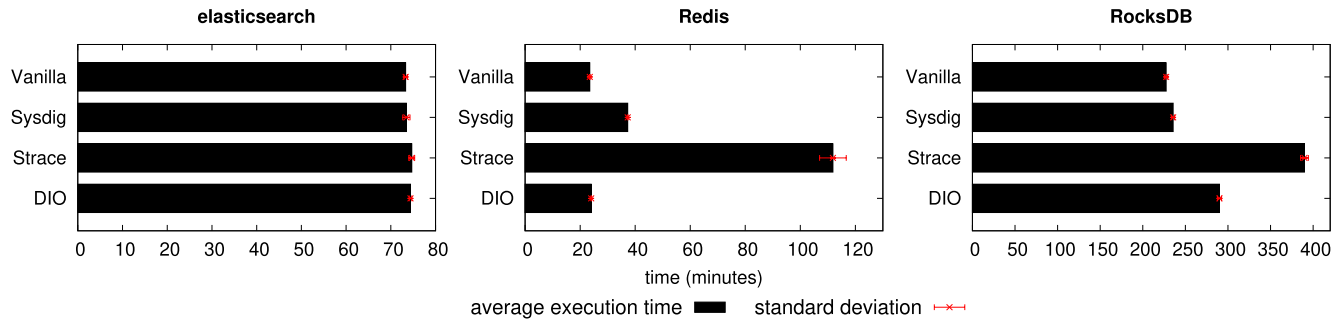


FIGURE 8. Average execution time for Elasticsearch, Redis, and RocksDB use cases with DIO, Sysdig and Strace.

intercepts all generated syscalls, including those targeting network sockets, which explains its significantly higher performance overhead.

For RocksDB, the most I/O intensive application, Sysdig, DIO, and Strace increase *vanilla* execution time (227.5 min) by 1.07x (235.6 min), 1.38x (290.1 min), and 1.74x (389.8 min), respectively. Although Sysdig presents the smallest performance overhead, DIO is the only tracer capable of providing near real-time analysis and visualization of collected data and can still reduce the overhead imposed by Strace.

I/O events handling: As discussed in §III, 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 aforementioned experiments, DIO is able to capture all storage syscalls generated by Elasticsearch (1M) and Redis (600K). For RocksDB, given its more intensive I/O behavior, 6% of the issued syscalls (≈34M of 538M) were discarded at the ring buffer and, therefore, not stored at DIO's *backend*.

Regarding the storage space needed by DIO's backend to store the collected information, the Redis, Elasticsearch and RocksDB use cases require approximately 86MiB, 282MiB and 90GiB, respectively.

E. SUMMARY

The previous use cases demonstrate that DIO is useful for diagnosing distinct I/O patterns. Namely, with Redis, RocksDB, and Fluent Bit, we show that DIO can be used by developers to observe and confirm known issues and to validate the correction of their fixes. With Elasticsearch, we show that our tool is useful when users wish to explore unknown applications. Indeed, DIO is used to observe an inefficient I/O pattern that was not known a priori. Our integrated tracing and analysis pipeline enables users to observe these I/O patterns without resorting to code instrumentation or needing to manually combine multiple tools.

Experimental results show that DIO can collect, parse, and forward to the analysis pipeline all the required traced information while imposing reduced performance overhead. When compared to Strace, DIO reduces execution time for all applications. When compared with Sysdig, performance overhead varies with the amount of information captured at kernel, sent to user-space, and reported to users. Despite the discarded I/O events in RocksDB, we show that DIO can pinpoint resource contention and help diagnose its root cause. Moreover, unlike in Strace and Sysdig, DIO's traced information is automatically made available for analysis as soon as it is collected and transmitted to the *backend* component.

V. EXPERIMENTAL STUDY

We now focus on studying how DIO behaves under intensive I/O workloads to answer the following questions:

- *What is the performance and resource usage of* DIO *when tracing I/O intensive applications?*
- *How much information can* DIO *capture without discarding events?*
- *What is the performance and accuracy impact of* DIO*'s configurations (e.g., ring buffer size, batch size) and optimizations (i.e., tracing modes and filters)?*
- *How does* DIO *compare to other state-of-the-art syscall tracers?*

To that end, we first compare DIO with other state-of-the-art solutions (§V-B), and then further study DIO's inline pipeline (§V-C), adaptability to different I/O rates (§V-D), and the impact of its filtering mechanisms (§V-E).

A. METHODOLOGY

For the conducted experiments, we compared DIO with two state-of-the-art syscall tracers:

- *Strace*: a popular diagnostic, debugging, and instructional user-space utility that leverages the `ptrace` kernel feature to non-intrusively intercept syscalls [11].
- *Sysdig*: a tool for system troubleshooting, analysis, and exploration that also leverages the eBPF technology to intercept syscalls [15].

Similar to DIO, these tracers intercept syscalls invoked by user-space applications and collect information regarding

TABLE 3. Description of each setup used in the experiments for Strace, Sysdig, and DIO tracers.

Setup	Description	Tracer
Raw	Syscalls information is saved in its *raw* format without any kind of pre-processing.	Strace DIO
$\boldsymbol{DetailedP_{args}}$	Syscalls information is pre-processed, and file paths are obtained from syscall arguments only (*i.e.*, no translation of file descriptors to file paths).	Strace
$\boldsymbol{DetailedP_{fds}}$	Syscalls information is pre-processed, and file paths are obtained from file descriptors only (*i.e.*, file paths from syscalls' arguments are discarded).	DIO
$\boldsymbol{DetailedP_{all}}$	Syscalls information is pre-processed, and file paths are obtained from both file descriptors and syscalls' arguments.	Strace Sysdig DIO
$\boldsymbol{DetailedP_{all}C_{plain}}$	Similar to $detailedP_{all}$, but including data buffers' content in plaintext.	Strace Sysdig
$\boldsymbol{DetailedP_{all}C_{uhash}}$	Similar to $detailedP_{all}$, but including a hash sum of data buffers' content computed in user-space.	DIO
$\boldsymbol{DetailedP_{all}C_{khash}}$	Similar to $detailedP_{all}$, but including a hash sum of data buffers' content computed in kernel-space.	DIO

their type, arguments, and return value. They also offer different filtering capabilities and allow configuring which data to collect (*e.g.*, enabling/disabling the collection of data buffers).

1) SETUPS

To evaluate the impact of collecting more or less detailed information, we configured each tracer (whenever possible) with the setups described in Table 3.

2) STORAGE BACKENDS

While DIO offers an integrated analysis pipeline to collect, analyze, and visualize tracing data, Strace and Sysdig focus only on the tracing phase, saving collected data to disk. To fairly compare the three tracers and study the impact of different storage backends, in addition to the default deployment of DIO (*i.e.*, sending collected information directly to the remote analysis pipeline), we also evaluated DIO's performance when saving traced data directly to disk.

Moreover, to study the feasibility of building an integrated diagnosis pipeline with existing tracing solutions, we use the Logstash [32] data processing tool to automatically parse and forward Sysdig events to a similar inline analysis pipeline as in DIO (*i.e.*, composed by Elasticsearch and Kibana). Specifically, for these experiments, Sysdig is configured to write collected data to the standard output, which is redirected (via a Unix pipe) to Logstash. The latter reads, parses, and forwards collected events, in batches, to Elasticsearch.

3) WORKLOAD AND COLLECTED METRICS

To produce a stress-test scenario, we used the Filebench benchmark, a popular framework for file system and storage benchmarking [33], [34]. Experiments consisted of running Filebench with the *FileServer* workload, configured to access 10.000 files, each sizing 128 KiB, through 4 threads performing storage I/O requests for 20 minutes. The experiments were conducted in the same testbed as described in §IV. Results include the average and standard deviation of the number of operations per second (*ops/s*) for three independent runs. Unless stated otherwise, the standard deviation for all experiments is equal or inferior to 3% of the corresponding throughput.

The Dstat [35] tool was used to obtain system resource statistics, including CPU, memory, disk, and network usage.

Finally, the storage overhead imposed by each tested setup (*i.e.*, size of the generated tracing file or Elasticsearch's index size) was also computed, as well as the number of intercepted syscalls, including *complete* (events saved with all the information), *incomplete* (events saved with partial information), and *lost* events (events discarded in kernel-space).

B. COMPARISON WITH STATE-OF-THE-ART TRACERS

First, we compare DIO's performance, resource usage, and tracing accuracy against Strace and Sysdig in a stress-test environment (depicted in Fig. 9). We start by doing an individual analysis for each tracer, and then we discuss how these compare to each other.

1) PER-TRACER ANALYSIS

a: VANILLA

The Filebench benchmark running without any tracer (*vanilla* setup) generates approximately 164 Kops/s (depicted by the black dashed line - - -). Unless stated otherwise, the performance overhead values discussed in this section correspond to the decrease in throughput percentage of a given setup when compared with the *vanilla* setup.

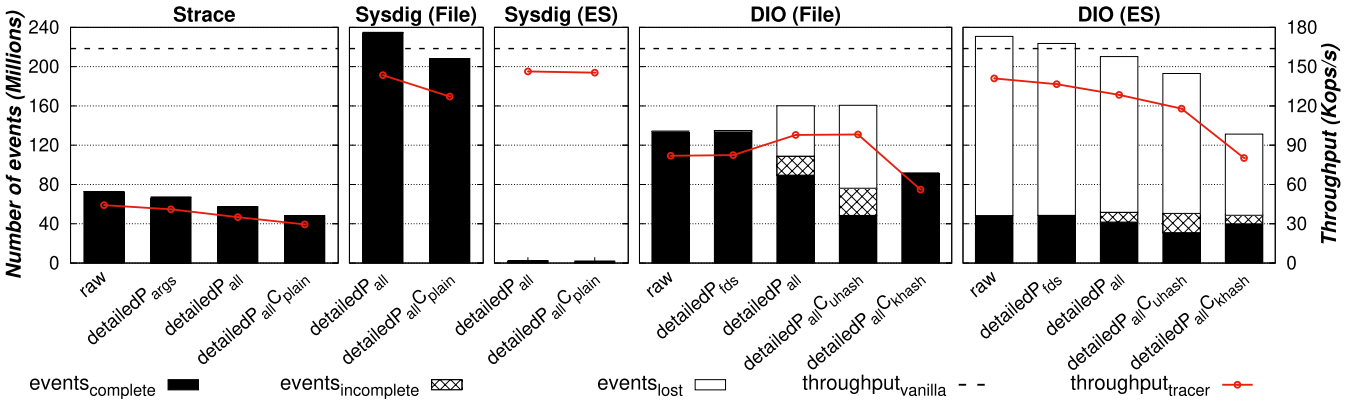


FIGURE 9. Performance overhead and collected events when tracing Filebench with Strace, Sysdig, and DIO.

b: STRACE

Strace imposes high performance overhead over Filebench's workload (depicted by the red line ——), reducing throughput by 73% with the least detailed setup (*raw*). The imposed overhead increases further as more detailed information is captured. Namely, with the more detailed setup ($detailedP_{all}C_{plain}$), Strace achieves only 30 Kops/s, increasing the overhead to 82%.

While Strace can save all intercepted syscalls without losing tracing data (*i.e.*, no *incomplete* or *lost* events), the number of collected events decreases with more detailed setups, ranging from 73M (*raw*) to 48M ($detailedP_{all}C_{plain}$). This is a consequence of Strace's performance overhead, which decreases the number of operations per second done by Filebench and, consequently, the total number of issued syscalls.

c: SYSDIG

Sysdig incurs reduced performance overhead over Filebench, decreasing throughput by 12% with the $detailedP_{all}$ setup. Similarly to Strace, capturing more information imposes higher overhead. Namely, with the $detailedP_{all}C_{plain}$ setup, Sysdig achieves 127 Kops/s, increasing the overhead to 22%.

With the reduced performance overhead, Filebench generates more operations per second, and consequently, more syscalls are performed in total. When writing data to disk, Sysdig saves between 208M ($detailedP_{all}$) to 235M ($detailedP_{all}C_{plain}$) of *complete* events, with only 41 incomplete events (*i.e.*, missing information about file paths). However, in these experiments, Sysdig is unable to report the process name for all captured events.

When sending collected data to Elasticsearch, Sysdig still imposes reduced overhead (≈11%) but only saves 2M events, some of them with incomplete data (between 131 and 314 *incomplete* events). This is a consequence of Sysdig producing data faster than Logstash can process it, filling the Unix pipe connecting Sysdig and Logstash and forcing Sysdig to discard events. Interestingly, when saving data to Elasticsearch, Sysdig can obtain the process name for all events.

d: DIO

The performance overhead imposed by DIO varies depending on the storage backend used (*i.e.*, file or Elasticsearch) and the amount of computation performed in kernel space (*i.e.*, in the critical path of I/O requests).

When writing data to disk, the *raw* and $detailedP_{fds}$ setups process all intercepted syscalls, saving ≈134M *complete* events, but reduce Filebench's throughput by 50% (with a standard deviation of 6% for $detailedP_{fds}$). The $DetailedP_{all}$ and $detailedP_{all}C_{uhash}$ setups start losing tracing data (between 20M-28M *incomplete* and 51M-85M *lost* events) but impose less overhead (40%), achieving 98 Kops/s.

By capturing the file paths from syscall arguments ($detailedP_{all}$) and the data buffers' content ($detailedP_{all}C_{uhash}$), DIO needs to transfer larger events from kernel to user-space. Moreover, by computing a hash sum of the buffers' content in user-space, DIO further delays the events' pulling from the *ring buffer*. If no space is available on the *ring buffer* to send an event to user-space, the event is discarded in the kernel and considered *lost*.

Interestingly, as the number of *lost* events increases, the performance overhead decreases due to the additional computation of copying data from the kernel to user-space that happens in the critical I/O path of a request.

The $detailedP_{all}C_{khash}$ setup reduces the amount of data to transfer to user-space by computing the hash sum of buffers' content in kernel space. While it allows minimizing the loss of events (*i.e.*, no *incomplete* or *lost* events are observed), it ends up adding heavy computation to I/O requests' critical path, imposing higher overhead (66%).

A similar phenomenon is visible when using Elasticsearch as DIO's storage backend: DIO saves fewer events (48M to 52M) but imposes less overhead (14% to 51%). However, these results show two other phenomena. First, DIO's rate for processing and saving events to Elasticsearch is limited to ≈43K events/s, thus filling up the *ring buffer* more quickly

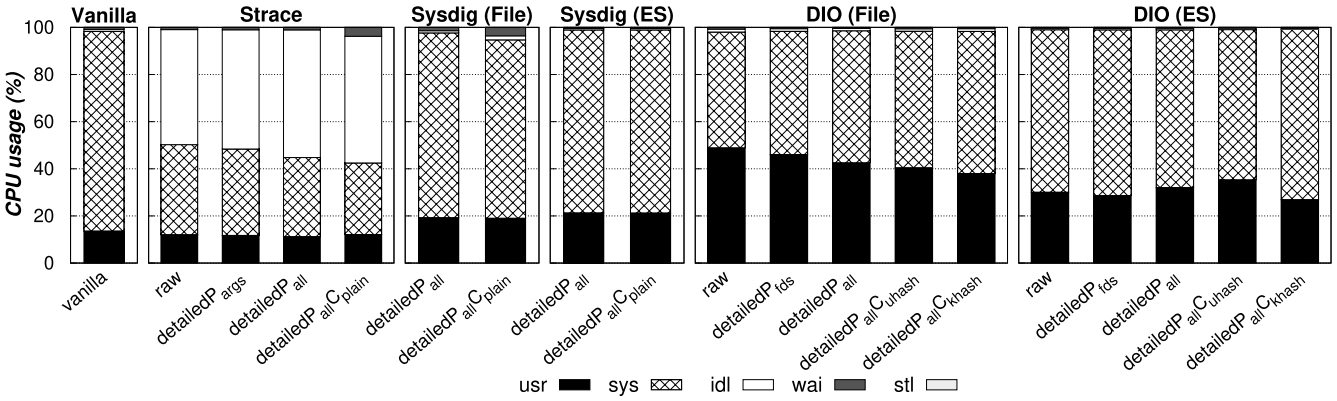


FIGURE 10. CPU usage by Strace, Sysdig, and DIO.

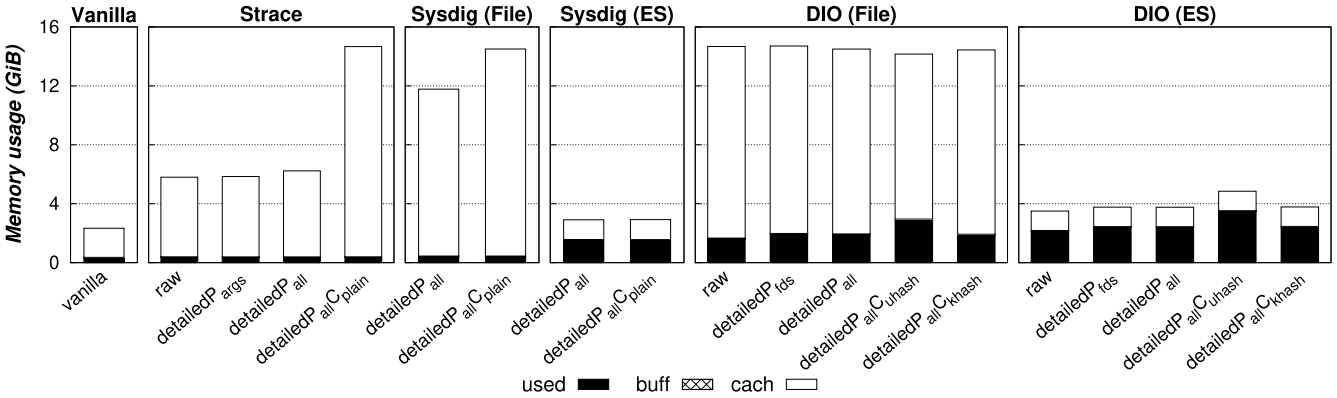


FIGURE 11. Memory usage by Strace, Sysdig, and DIO.

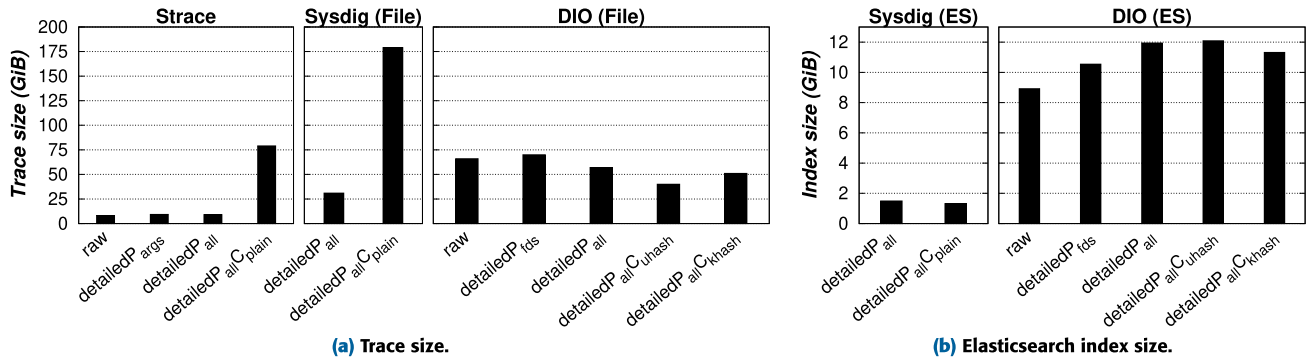


FIGURE 12. Storage usage by Strace, Sysdig, and DIO.

and losing more events. Second, when writing data to disk, the performance overhead is especially dictated by the extra computation of copying data to user-space. However, when writing data to Elasticsearch, the overhead imposed by the extra processing in kernel to gather more detailed information has a higher impact (*i.e.*, more detailed setups impose higher overhead). This is more noticeable for the $detailedP_{all}C_{khash}$ setup that due to computing the hash sum in kernel, reduces throughput by 51% (with a standard deviation of 4%).

2) COMPARATIVE ANALYSIS

Next, we compare the three tracers regarding their imposed performance overhead, tracing accuracy, and resource usage.

a: PERFORMANCE OVERHEAD

Strace imposes the highest overhead over Filebench's workload, while Sysdig imposes the lowest. DIO offers better results than Strace for all setups and storage backends,

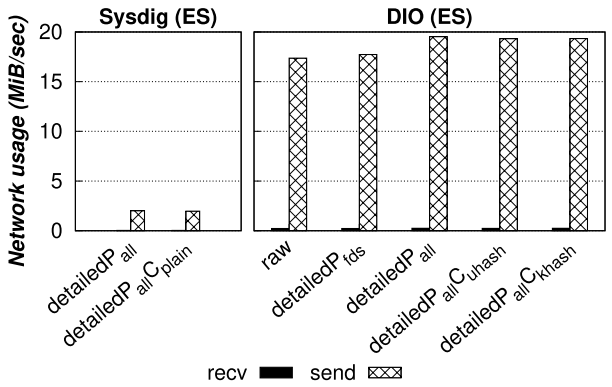


FIGURE 13. Network usage by Strace, Sysdig, and DIO.

resulting from using a low overhead technology, eBPF, instead of a more costly approach like `ptrace`. When configured with the Elasticsearch storage backend, DIO provides results closer to Sysdig. In general, all tracers impose higher overhead when collecting more detailed information.

b: COLLECTED EVENTS

By imposing the lowest performance overhead, Sysdig is also the tracer that saves more events when writing these to disk. In the same way, by imposing the highest overhead, Strace saves fewer events than the other tracers. When sending data to Elasticsearch, both Sysdig and DIO are forced to discard events. However, by including a custom implementation to communicate directly with Elasticsearch, DIO can save significantly more events than Sysdig. As in the performance overhead results, collecting more detailed information generally translates into more *incomplete* and *lost* events for DIO and Sysdig.

c: CPU USAGE

Fig.10 shows the CPU usage for each tracer and setup. Due to its synchronous approach for intercepting syscalls and higher performance overhead, Strace exhibits significant CPU *idle* time, which increases when more detailed data is collected. Vanilla, Sysdig, and DIO setups have negligible *idle* time but exhibit different values for the time spent in user-space (*usr*) and kernel-space (*sys*). The additional *usr* time used by DIO is explained by the processing done by our solution in user-space, as explained in §III-B). However, DIO's *usr* time reduces, while *sys* time increases, for setups capturing more detailed information, as these require extra processing at the critical path of I/O requests.

d: MEMORY USAGE

Regarding memory consumption (Fig. 11), it is noticeable an increased usage of cache resources (*cach*) for setups writing traced data to disk, which is a consequence of using more space from the operating system's page cache. This value increases further when considering more detailed setups. DIO presents higher values for *used* memory, which is explained by the eBPF maps and the *ring buffer* used by our tracer (*e.g.,* the default size of the *ring buffer* is 1GiB in total) and by writing events in batches to Elasticsearch for increased performance (*i.e.*, the default configuration is 7MiB per thread, 28MiB in total). The latter justifies why our solution uses more memory when using the Elasticsearch backend instead of the file one. Finally, the increase in *used* memory for Sysdig, with the Elasticsearch backend, is due to Logstash's internal buffers for processing traced data.

e: STORAGE USAGE

Fig. 12 shows the storage space used by each tracer when writing data to disk (12a) and when sending events to Elasticsearch (12b). Strace generates a trace file with a smaller size (around 9GiB for *raw*, $detailedP_{args}$, and $detailedP_{all}$ setups), which can be explained by the smaller number of events it collects. DIO collects more events and writes data to disk in a JSON format, thus generating larger files (from 40GiB with $detailedP_{all}C_{uhash}$ to 70GiB with $detailedP_{fds}$). Sysdig, on the other hand, writes data to disk in a binary format, and therefore, although being the tracer that collects more events, it can create compact files (≈31GiB with $detailedP_{all}$ setup). Nonetheless, both Strace and Sysdig generate larger trace files when capturing the data buffer's content ($detailedP_{all}C_{plain}$ setup), generating files of 79GiB and 179GiB, respectively. By contrast, DIO minimizes the trace size by saving a hash sum of the buffers' content ($detailedP_{all}C_{uhash}$ and $detailedP_{all}C_{khash}$ setups).

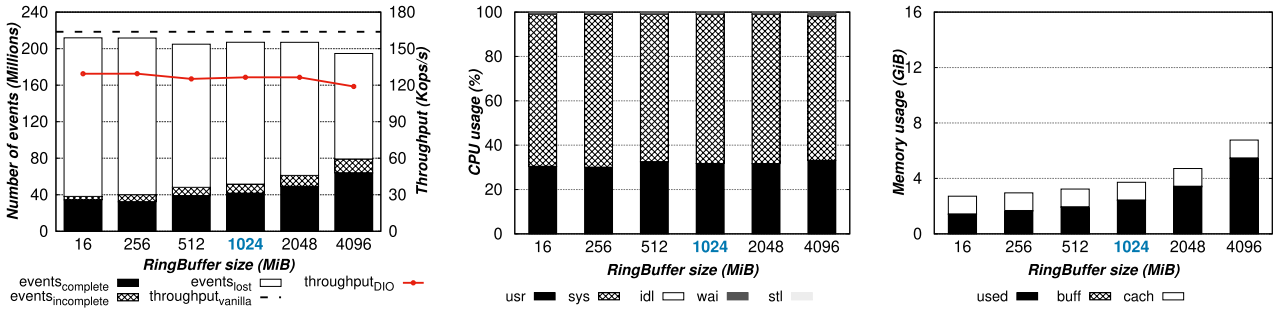
When sending data to Elasticsearch, the resulting index size for Sysdig is around 1.4GiB, while for DIO it varies between 8.9GiB and 12.1GiB. The difference between the two is mainly dictated by the number of collected events and the amount of detailed information captured.

f: NETWORK USAGE

As depicted in Fig. 13, when considering Sysdig and DIO using Elasticsearch as the storage backend, DIO consumes more network bandwidth (*i.e.*, 2MiB/s for Sysdig and between 17MiB and 20MiB/s for DIO). Since DIO sends more megabytes per second to Elasticsearch, it can process traced data from the *ring buffer* faster and save more information at the backend, as shown in Fig. 9.

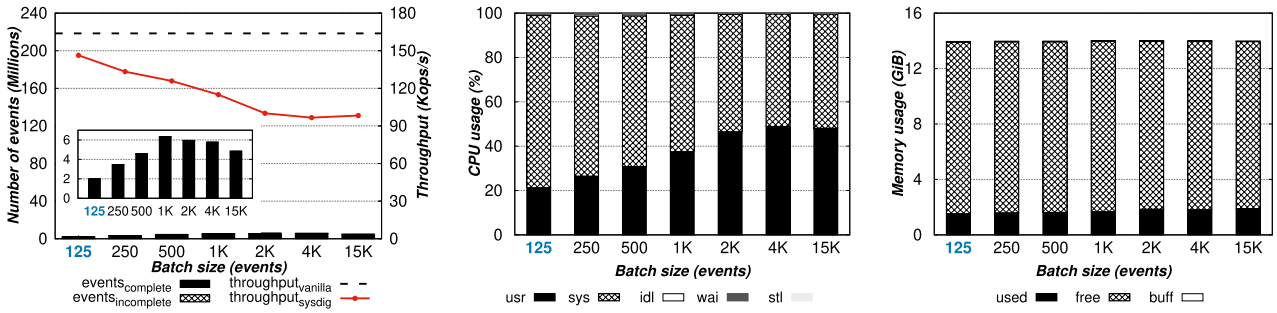
From the previous results and analysis, one can extract the following main takeaways.

> **Takeaway 1.** *Capturing more detailed information from syscalls induces higher performance and resource usage overhead. Also, when syscalls are intercepted asynchronously (i.e., in* DIO *and Sysdig), it may lead to a larger number of incomplete and lost events. For synchronous approaches (i.e., in Strace), the performance overhead is more noticeable.*



(a) Throughput and collected events. (b) CPU usage. (c) Memory usage.

FIGURE 14. Performance overhead, collected events, and resource usage for different *ring buffer* sizes in DIO. The blue color pinpoints the default configuration.



(a) Throughput and collected events. (b) CPU usage. (c) Memory usage.

FIGURE 15. Performance overhead, collected events, and resource usage for different batches sizes in Sysdig. The blue color pinpoints the default configuration.

Takeaway 2. *The storage backend where traced data is stored influences the number of collected events and performance overhead. A slower backend (i.e., using Elasticsearch instead of a high-performance local disk) reduces the overhead over the application but leads to a higher number of incomplete and lost events for Sysdig and* DIO.

Takeaway 3. *Tracers using eBPF technology (i.e., Sysdig and* DIO*) exhibit distinct trade-offs related to the balance between the computation done in user-space and in kernel-space. More computation in user-space delays the collection of events from the ring buffer, which results in more discarded events and less performance overhead. More computation in kernel induces a higher performance penalty for the traced application.*

Takeaway 4. DIO *offers the best trade-off in terms of performance overhead and collected events when considering a full inline pipeline for tracing, analyzing, and visualizing I/O syscalls (i.e., it captures between 23x to 28x more events when compared with Sysdig, while maintaining performance overhead under 51%). On the other hand, Sysdig presents the best trade-off regarding performance overhead and collected events when considering only the tracing step for a local disk backend.*

C. INLINE ANALYSIS PIPELINE

Next, we assess the best configurations to provide an efficient inline analysis pipeline (*i.e.*, in which traced data is sent directly to the Elasticseach backend). We start by studying the impact of varying the *ring buffer* size in DIO, and evaluating both Sysdig and DIO when sending batches of different sizes to Elasticsearch. Then, we discuss the advantages and drawbacks of following an inline *vs* offline approach (*i.e.*, saving data to disk and sending it posteriorly to Elasticsearch).

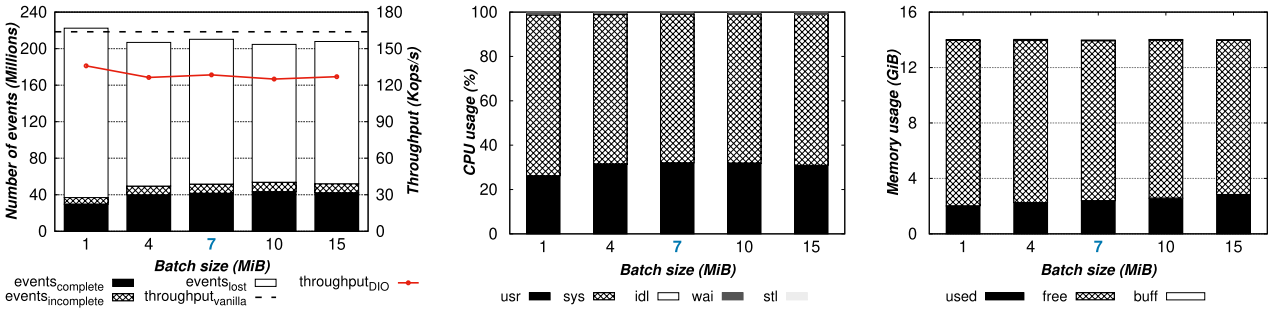
For these experiments, both tracers are configured with the $detailedP_{all}$ setup for a fair comparison.

1) RING BUFFER'S SIZE IMPACT IN DIO

To further explore *Takeaway 3*, it is important to assess the impact that different *ring buffer* sizes have on DIO's performance and amount of collected events.

As expected and shown in Fig. 14a, the larger the *ring buffer*, the more events DIO collects. With a smaller configuration (16 MiB), DIO saves 38M events. If a larger *ring buffer* is used (4096 MiB), it saves up to 79M events.

However, increasing the *ring buffer* size impacts the performance overhead imposed over Filebench. Namely, overhead ranges from 21% (16 MiB configuration) to 27% (4096MiB configuration). Regarding resource usage, varying the *ring buffer* size has minimal impact on CPU usage, as shown in Fig. 14b. As for memory consumption (depicted in Fig. 14c), the larger the *ring buffer*, the more *used*



(a) Throughput and collected events. (b) CPU usage. (c) Memory usage.

FIGURE 16. Performance overhead, collected events, and resource usage for different batches sizes in DIO. The blue color pinpoints the default configuration.

memory DIO needs, ranging from 1.4GiB (16 MiB) to 5.4GiB (4096 MiB).

The 1024 MiB configuration offers a good trade-off between performance overhead (23%), collected events (52M), and memory usage (3.7 GiB in total), and therefore was selected as the default *ring buffer* configuration for DIO in all the experiments discussed at this section.

> **Takeaway 5.** *With a larger ring buffer,* DIO *collects more events from kernel in user-space. However, it increases performance overhead and memory usage.*

2) ELASTICSEARCH'S BATCH SIZE IMPACT

To complement the conclusions from *Takeaways 2* and *4*, we next assess the impact of using different batch sizes for transmitting traced data to Elasticsearch.

Fig. 15a shows the throughput and number of collected events for Sysdig when Logstash sends batches of 125, 250, 500, 1K, 2K, 4K, and 15K events to Elasticsearch. The default configuration used in our experiments (125 events) imposes the least overhead (12%) but saves less information (≈2M events). Increasing the batch size results in higher performance overhead, while the number of collected events increases only for sizes inferior to 1K. For the latter, Sysdig collects more events (≈6.3M) and imposes a performance overhead of 31%. For larger batches, the overhead tends to stabilize around 40%, but Sysdig starts collecting less information (*e.g.*, with a batch size of 15K, Sysdig collects 4.9M events). As depicted in Fig. 15b, CPU *usr* time increases with larger batches. The *used* memory (shown in Fig. 15c) varies between 1.6 GiB and 1.9 GiB.

In DIO, due to a different approach in terms of design and implementation (*i.e.*, it does not resort to Logstash, as the *tracer* sends information directly to Elasticsearch to be more efficient), batch sizes must be configured in MiB instead of the number of events. Fig. 16a shows the throughput and number of collected events for DIO when configured with batches of 1, 4, 7, 10, and 15 MiB. Like in Sysdig, with larger sizes, DIO collects more events and

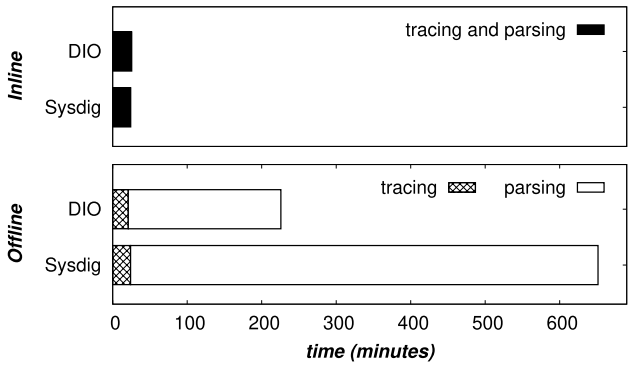


FIGURE 17. Execution times for *inline* and *offline* analysis approaches with Sysdig and DIO.

imposes higher performance overhead. However, the variance across different size configurations is small, always capturing more than 37M events and imposing no more than 24% of overhead. In detail, a batch size of 1 MiB imposes the smallest overhead (17%) but collects less information (37M events). On the other hand, a size of 10 MiB allows collecting more events (54M) but imposes the highest overhead (24%).

Contrarily to Sysdig, increasing the batch size in DIO has a negligible effect on CPU usage (Fig. 16b), but increases *used* memory, going from 2.0 GiB (batch size of 1 MiB) to 2.8 GiB (batch size of 15 MiB). Therefore, our experiments consider a default batch size of 7MiB for DIO.

> **Takeaway 6.** *In Sysdig, increasing the batch size has a bigger effect in the balance between performance overhead and events captured than in* DIO.

> **Takeaway 7.** *When considering different batch size configurations,* DIO *is able to capture from 6x to 27x more events than Sysdig. Also, the performance overhead in* DIO *is always kept below 23%, while in Sysdig, it drops up to 41% for larger batches.*

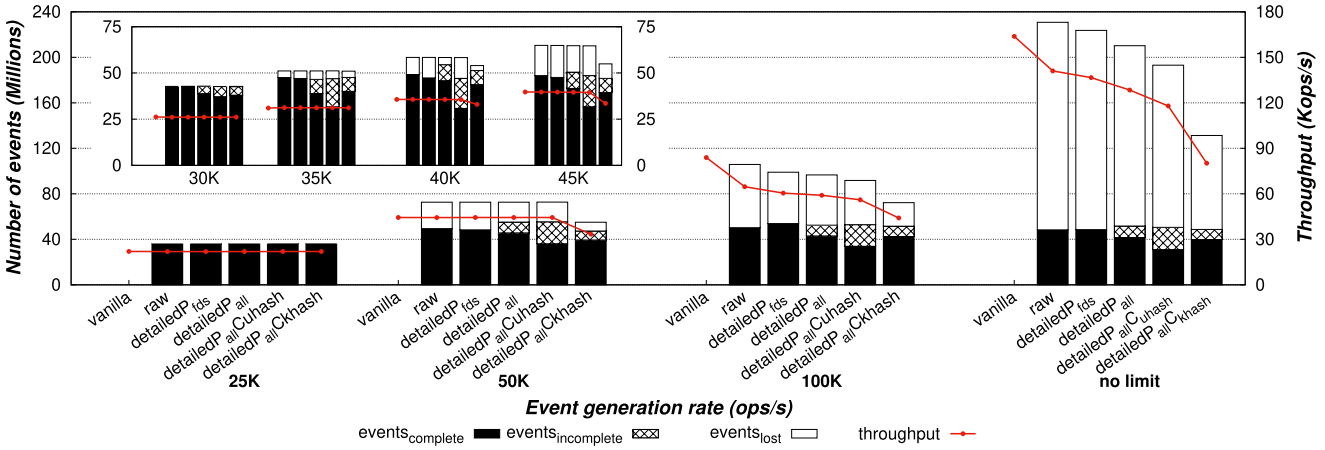


FIGURE 18. Performance overhead and collected events of DIO's setups (with Elasticsearch backend) when tracing Filebench with different I/O rates.

3) INLINE VS. OFFLINE ANALYSIS

According to the results from §V-B and *Takeaway 2*, both Sysdig and DIO save more tracing information when writing data to a local disk. However, such an option requires users to, later on, parse and forward this information from disk to Elasticsearch in an offline fashion. Next, we evaluate the benefits and drawbacks of following offline and inline approaches.

For inline experiments, Sysdig is configured with a batch size of 1K since it allows collecting more events, and DIO with the default batch size of 7MiB, which offers a good trade-off between performance overhead, collected events, and resource usage, as observed in §V-C2.

For offline experiments, a custom DIO's trace processor is used to read traced data from disk, parse, and forward it to Elasticsearch. For Sysdig, since data is written in a binary format, we use Sysdig's functionality to read the tracer's binary data and redirect its output to Logstash (via a Unix pipe) for parsing and forwarding data to Elasticsearch. In both tracers, we use a batch size of 15K events (the largest size supported for our experimental setup) to optimize data transmission speed to the backend.

Fig. 17 shows the time spent by each tracer when collecting information (*tracing*), and processing and forwarding data to Elasticsearch (*parsing*). Starting with the inline approach, both tracers present similar execution times (≈24 mins for Sysdig and ≈26 mins for DIO), but Sysdig can only send 6M events to Elasticsearch, while DIO sends up to 52M events.

When following an offline approach, DIO sends up to 108M events to Elasticsearch, taking 21 mins to collect information (tracing phase) and 205 mins to process and forward it (parsing phase). Sysdig can collect even more information (around 230M events) during ≈24 mins but takes about ≈627 mins (*i.e.*, 10 hours and 27 min) to process and forward all these events to Elasticsearch. By further inspecting these results, we noticed that Sysdig takes only ≈32 mins reading the 230M events saved on disk, thus exposing Logstash as the main reason for the long parsing time.

> **Takeaway 8.** *Inline approaches significantly reduce the time for users to start analyzing collected data, at the cost of discarding syscalls issued by the targeted application.*

> **Takeaway 9.** *When following an offline approach, Sysdig captures more events but takes an impractical amount of time to parse and forward traced data to the backend. Since* DIO *is implemented and optimized to interact directly with Elasticsearch, it exhibits better performance.*

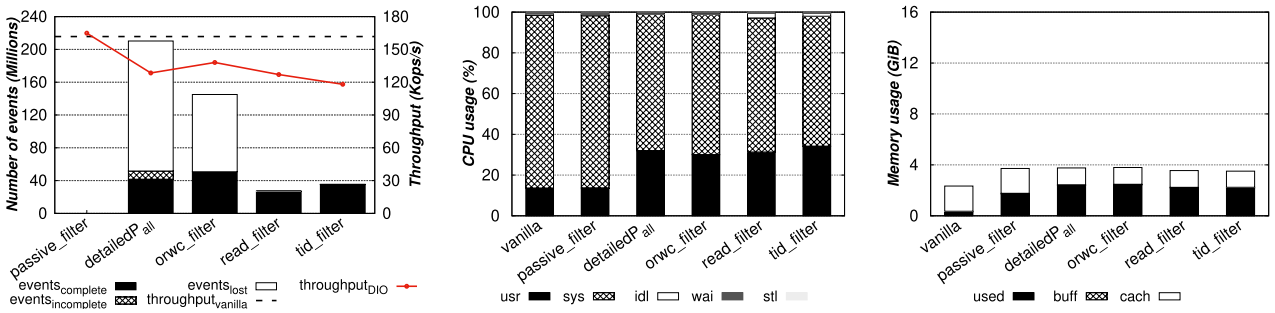
D. DIO'S ADAPTABILITY TO DIFFERENT I/O RATES

In practice, data-centric applications (*e.g.*, databases, key-value stores) access storage resources with different I/O rates. Fig. 18 shows DIO's performance overhead and collected events when Filebench is configured to generate I/O operations at specific rates (or inferior if the system cannot handle these), starting at 25 Kops/s.

When Filebench issues operations at a rate inferior or equal to 25 Kops/s, all setups collect the full information from issued syscalls (36M events).

With a rate of 30 Kops/s, the setups capturing more detailed information ($detailedP_{all}$, $detailedP_{all}C_{uhash}$, and $detailedP_{all}C_{khash}$) save some incomplete events (4M to 6M). *Lost* events happen for rates equal to or superior to 35 Kops/s (3M to 5M), but DIO only starts impacting Filebench's performance for rates over 40 Kops/s and mostly with $detailedP_{all}C_{khash}$.

The difference between vanilla and DIO's setups is more noticeable for rates equal to or greater than 100 Kops/s, where performance overhead ranges from 15% to 51%, and the number of *lost* events varies from 21M to 182M syscalls.



(a) Throughput and syscall stats. (b) CPU usage. (c) Memory usage.

FIGURE 19. Performance overhead and collected events of DIO (with *detailedP$_{all}$* setup and Elasticsearch storage backend) when applying different filters.

Takeaway 10. *The performance overhead and number of events collected by* DIO *changes according to the I/O rate of applications. Under 25 Kops/s, all setups collect the full set of issued operations, while most setups only have a noticeable performance impact over Filebench when surpassing 100 Kops/s.*

E. DIO's FILTERS IMPACT

As discussed in §III-B, capturing only events of interest helps users reduce the overhead imposed on the target application and the volume of data to analyze. To evaluate the impact of filtering events at the tracing phase, we now compare the default configuration of DIO (*detailedP$_{all}$* mode with the Elasticsearch storage backend, which does not apply any filters) with four new setups:

- *passive_filter* - captures only the `rename` syscall type, which is never invoked by Filebench.
- *orwc_filter* - captures a subset of syscalls (*i.e.*, `open`, `read`, `write`, and `close`).
- *read_filter* - captures only `read` syscalls.
- *tid_filter* - captures all syscalls made by a specific thread of Filebench.

The *passive_filter* evaluates the impact of having an active tracepoint that is never triggered by the targeted application, while *orwc_filter* and *read_filter* assess the impact of activating more or less tracepoints. Finally, the *tid_filter* evaluates the impact of filtering events of interest in kernel-space.

a: COLLECTED EVENTS

Fig. 19a shows the number of collected events and performance overhead for each setup. When capturing all events with *detailedP$_{all}$* (*i.e.*, without any filters), DIO intercepts 210M syscalls, saving 42M *complete* and 10M *incomplete* events while losing 158M events. The *orwc_filter* setup reduces the events of interest to 145M, which allows saving more *complete* events (≈51M) and reducing the *incomplete* and *lost* events to 0 and 94M, respectively. By capturing only *read* syscalls, the *read_filter* further reduces intercepted events to 27M, being able to save them all along with their complete information (*i.e.*, no *incomplete* or *lost* events). Similarly, by filtering events from a specific TID in kernel-space, the *tid_filter* setup intercepts and saves 35M events of interest.

b: PERFORMANCE IMPACT

As shown in Fig. 19a, DIO does not impose extra overhead if an active probe is never triggered (*passive_filter*), achieving a performance throughput similar to vanilla (165 Kops/s). When activating all supported tracepoints (*detailedP$_{all}$*), DIO introduces an overhead of 22%. By filtering events by a specific subset of syscalls (*orwc_filter*), DIO reduces the number of active tracepoints and therefore decreases the overhead to 16%.

However, the *read_filter* setup, which only activates one tracepoint, imposes similar overhead as in *detailedP$_{all}$*. These results are explained by the number of *eventPath* events (used to map file descriptors to file paths, as described in §III-C3) saved by each setup. Namely, the *detailedP$_{all}$* and *orwc_filter* setups can only save between 122K to 358K *eventPaths*. On the other hand, the *read_filter* setup saves all generated *eventPaths* (≈7M), which requires copying more data from kernel to user-space, imposing a higher performance overhead. The same is valid for *tid_filter*, which increases overhead to 28% because it also saves a large number of *eventPaths* (≈5M).

c: RESOURCE USAGE

Figs. 19b and 19c show CPU and memory usage results. The *passive_filter* setup presents similar CPU consumption as in vanilla since it does not intercept any syscall. The other setups increase *usr* time by ≈16%. As for memory consumption, all DIO's setups present similar results, which is explained by the static allocation of memory done by our system (*e.g.*, for the *ring buffer*). Regarding storage usage, by reducing the events of interest, the *read_filter* and *tid_filter* setups minimize the *detailedP$_{all}$*'s index size by 42% (7GiB) and 50% (6GiB), respectively.

Takeaway 11. DIO*'s filters enable users to target only events of interest, which reduces storage overhead, improves the number of collected events, and, depending on the filter type and I/O workload, reduces performance overhead.*

F. SUMMARY

To sum up, the results and takeaways discussed in this section show that the approach followed by DIO is key for building an integrated tracing and analysis pipeline that can offer a good trade-off regarding performance overhead, tracing accuracy, and timely analysis for users.

First, by relying on the eBPF technology, DIO can intercept applications' syscalls without modifying their source code while minimizing tracing performance overhead.

Secondly, the flexibility offered by DIO's different tracing modes allows balancing the tracing accuracy with the performance and storage overheads by configuring the detail of the information being captured. Likewise, DIO's filtering capabilities enable discarding fewer I/O events of interest while reducing the storage capacity needed at the *Backend*. Specifically, the storage overhead imposed by DIO varies according to the tracing mode, filters used, and the number of events generated by the application (*e.g.*, 86 MiB for the Redis use case with ≈600K events collected with the *detailed*P_{all} mode, 90 GiB for the RocksDB use case with more than 500M events collected with the *raw* mode).

Moreover, by following an inline approach, DIO reduces the need to store traced data locally and enables users to analyze and visualize collected data in near real-time, which is not possible when following an offline design. Further, our custom design integrating the *Tracer* directly with the *Backend* component exhibits significantly better accuracy (*i.e.*, in terms of the amount of collected data at the *Backend*) than by combining different state-of-the-art tools (*i.e.*, Sysdig and Logstash).

In practice, and as shown in §IV, data-centric applications such as RocksDB, Elasticsearch, and Redis do not fully stress the underlying storage resources, and, by leveraging DIO's customized, flexible, and integrated design, users can capture the full set of syscalls or, at least, have a negligible number of events discarded that do not compromise the diagnosis of such applications.

VI. RELATED WORK

Table 4 shows a comparison between DIO and related 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.

TABLE 4. Comparison between DIO and related solutions regarding: *i)* tracing and (O-offline, I-inline) analysis functionalities, and *ii)* support for tracing (T) and analyzing (A) the use cases from §II and §IV.

		Strace [11]	*Sysdig* [15]	*Re-Animator* [16]	*RepTrace* [18]	*Tracee* [17]	*CaT* [4]	[3]	[36]	*LongLine* [19]	*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	§II	-	-	-	-	-	-	-	-	-	TA
	§IV-A	-	T	-	-	-	-	-	-	-	TA
	§IV-B	-	-	-	-	-	-	-	-	-	TA
	§IV-C	T*	T	-	-	T	T	-	-	T	TA

* Only supported for versions 5.15 or later.

A. I/O TRACING

Storage I/O diagnosis is often done by capturing applications' requests in user-space through source code instrumentation [7], [8], [9], [10]; through middleware libraries [37], [38] that are restricted to specific sets of applications (*e.g.*, *LD_PRELOAD* only works with dynamic libraries); or at lower kernel layers [5], [20], [38], 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 4, these explore distinct tracing technologies, including ptrace ([11], [18]), eBPF ([4], [15], [17]), LTTng ([3], [16], [36]), and auditd ([19]), 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], [15], [17], [19]), which is useful for observing the I/O patterns at §IV-B, and §IV-C. However, DIO is the only tool that collects *file offsets*, which are crucial for diagnosing the use case presented in §IV-B.

Only CaT [4], Tracee [17], 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* [11] and *Sysdig* [15], that support filtering at the tracing phase.

B. INTEGRATED ANALYSIS PIPELINE

Several solutions only cover the tracing step, leaving the integration with analysis pipelines to be done by users [11], [15], [16], [17]. 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], [18], [36]. Only DIO and Longline [19] automatically parse and forward traced events to the analysis pipeline by following an inline (near real-time) approach.

C. SYSCALL ANALYSIS AND VISUALIZATION

Some of the existing tools support analysis modules specialized for their concrete use cases (*e.g.*, causality [4], [18], security analysis [19]), 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], [36].

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.

Moreover, 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], [19], [36].

To sum up, DIO is the first solution providing an integrated inline diagnosis pipeline that is designed to be flexible and customizable, while covering a larger set of information from syscalls than other state-of-the-art solutions.

VII. CONCLUSION

This paper presents DIO, a generic tool for observing and diagnosing I/O interactions between applications and in-kernel POSIX storage backends. 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 observe I/O issues while reducing the search space for finding their root cause when, for instance, source code inspection is required.

Our experiments with widely used systems show that DIO provides key information for exploring I/O requests, observing inefficient or erroneous I/O access patterns that lead to performance degradation or data loss, and identifying resource contention in multithreaded I/O that leads to high tail latency. Further, a detailed evaluation comparing DIO with state-of-the-art tracers shows that our integrated diagnosis pipeline enables users to diagnose applications in a more timely fashion while providing the best balance in terms of performance overhead and tracing accuracy.

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 *Proc. 15th USENIX Conf. File Storage Technol.*, 2017, pp. 149–166.
[2] D. Roselli, J. R. Lorch, and T. E. Anderson, "A comparison of file system workloads," in *Proc. USENIX Annu. Tech. Conf.*, 2000, pp. 41–54.
[3] H. Daoud and M. R. Dagenais, "Performance analysis of distributed storage clusters based on kernel and userspace traces," *Softw., Pract. Exper.*, vol. 51, no. 1, pp. 5–24, Jan. 2021.
[4] T. Esteves, F. Neves, R. Oliveira, and J. Paulo, "CAT: Content-aware tracing and analysis for distributed systems," in *Proc. 22nd Int. Middleware Conf.*, Dec. 2021, pp. 223–235.
[5] A. Saif, L. Nussbaum, and Y.-Q. Song, "IOscope: A flexible I/O tracer for workloads' I/O pattern characterization," in *Proc. Int. Conf. High Perform. Comput.* Cham, Switzerland: Springer, 2018, pp. 103–116.
[6] T. Esteves, R. Macedo, R. Oliveira, and J. Paulo, "Diagnosing applications' I/O behavior through system call observability," in *Proc. 53rd Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. Workshops*, Apr. 2023, pp. 1–8.
[7] (2022). *Jaeger: Open Source, End-to-End Distributed Tracing*. [Online]. Available: https://www.jaegertracing.io
[8] (2022. *Zipkin*. [Online]. Available: https://zipkin.io
[9] 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 *Proc. SC Companion, High Perform. Comput., Netw. Storage Anal.*, Nov. 2012, pp. 18–23.
[10] K. Vijayakumar, F. Mueller, X. Ma, and P. C. Roth, "Scalable I/O tracing and analysis," in *Proc. 4th Annu. Workshop Petascale Data Storage*, Nov. 2009, pp. 26–31.
[11] (2022). *Strace: Linux Syscall Tracer*. [Online]. Available: https://strace.io
[12] M. Gebai and M. R. Dagenais, "Survey and analysis of kernel and userspace tracers on Linux: Design, implementation, and overhead," *ACM Comput. Surv.*, vol. 51, no. 2, pp. 1–33, Mar. 2019.
[13] S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture," in *Proc. Winter USENIX Conf.*, vol. 46, 1993, pp. 259–269.
[14] M. Desnoyers and M. R. Dagenais, "The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux," in *Proc. Ottawa Linux Symp.*, 2006, pp. 209–224.
[15] (2022). *Sysdig*. [Online]. Available: https://github.com/draios/sysdig/
[16] I. U. Akgun, G. Kuenning, and E. Zadok, "Re-animator: Versatile high-fidelity storage-system tracing and replaying," in *Proc. 13th ACM Int. Syst. Storage Conf.*, May 2020, pp. 61–74.
[17] (2022). *Tracee: Linux Runtime Security and Forensics Using eBPF*. [Online]. Available: https://github.com/aquasecurity/tracee
[18] 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 *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2019, pp. 527–538.
[19] S. Yoo, J. Jo, B. Kim, and J. Seo, "LongLine: Visual analytics system for large-scale audit logs," *Vis. Informat.*, vol. 2, no. 1, pp. 82–97, Mar. 2018.
[20] D. N. Jha, G. Lenton, J. Asker, D. Blundell, and D. Wallom, "Holistic runtime performance and security-aware monitoring in public cloud environment," in *Proc. 22nd IEEE Int. Symp. Cluster, Cloud Internet Comput. (CCGrid)*, May 2022, pp. 1052–1059.
[21] (2022). *Elasticsearch: The Heart of the Free and Open Elastic Stack*. [Online]. Available: https://www.elastic.co/elasticsearch/
[22] (2022). *Kibana: Your Window Into the Elastic Stack*. [Online]. Available: https://www.elastic.co/kibana/
[23] Redis. (2022). *Redis*. [Online]. Available: https://redis.io
[24] (2022). *Vega-Lite—A Grammar of Interactive Graphics*. [Online]. Available: https://vega.github.io/vega-lite/
[25] A. Bijlani and U. Ramachandran, "Extension framework for file systems in user space," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 121–134.
[26] (2022). *Fluent Bit: An End to End Observability Pipeline*. [Online]. Available: https://fluentbit.io
[27] Facebook. (2022). *RocksDB: A Persistent Key-Value Store for Fast Storage Environments*. [Online]. Available: https://rocksdb.org

[28] 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 *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 753–766.
[29] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput.*, Jun. 2010, pp. 143–154.
[30] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, Feb. 2013.
[31] 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 *Proc. ACM Symp. Cloud Comput.*, Nov. 2014, pp. 1–14.
[32] (2022). *Logstash: Centralize, Transform & Stash Your Data*. [Online]. Available: https://www.elastic.co/logstash/
[33] V. Tarasov, E. Zadok, and S. Shepler, "Filebench: A flexible framework for file system benchmarking," *Login, USENIX Mag.*, vol. 41, no. 1, pp. 6–12, Mar. 2016.
[34] (2022). *Filebench: A Model Based File System Workload Generator*. [Online]. Available: https://github.com/filebench/filebench
[35] (2022). *Dstat: Versatile Tool for Generating System Resource Statistics*. [Online]. Available: https://linux.die.net/man/1/dstat
[36] 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, Aug. 2021.
[37] S. Snyder, P. Carns, K. Harms, R. Ross, G. K. Lockwood, and N. J. Wright, "Modular HPC I/O characterization with Darshan," in *Proc. 5th Workshop Extreme-Scale Program. Tools (ESPT)*, Nov. 2016, pp. 9–17.
[38] 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 *Proc. Workshop Challenges Opportunities Efficient Performant Storage Syst.*, 2021, pp. 1–11.

TÂNIA ESTEVES is currently pursuing the Ph.D. degree with the Doctoral Program in Informatics, University of Minho. She is also a Researcher with the HASLab, one of the research units of INESC TEC and University of Minho. Her current research interest includes distributed systems focuses on the tracing and analysis of storage I/O. For more information, please visit: https://www.inesctec.pt/en/people/tania-conceicao-araujo.

RICARDO MACEDO received the joint Ph.D. degree from the Universities of Minho, Aveiro, and Porto, in 2023, under the MAP-i Doctoral Program. He is currently an Assistant Researcher with the HASLab, one of the research units of INESC TEC and the University of Minho, Portugal. His research interests include storage and operating systems. For more information, please visit: https://www.inesctec.pt/en/people/ricardo-goncalves-macedo.

RUI OLIVEIRA received the Ph.D. degree from École Polytechnique Fédérale de Lausanne, in 2000. He is currently an Associate Professor in Habilitation with the Informatics Department, University of Minho, a member of the Board of Directors of INESC TEC, the Director of the Minho Advanced Computing Centre, and the Co-Director of the UT Austin Portugal Program. His research interests include fault-tolerant distributed agreement and epidemic communication algorithms and in the conception, development, and assessment of dependable database systems. For more information, please visit: https://www.inesctec.pt/en/people/rui-carlos-oliveira.

JOÃO PAULO received the Ph.D. degree from the Universities of Minho, Portugal, Aveiro, and Porto, in 2015, under the MAP-i Doctoral Program. He is currently an Assistant Professor with the University of Minho, Portugal, and a Senior Researcher with the HASLab, one of the research units of INESC TEC and the University of Minho. His research interests include distributed and operating systems with an emphasis on storage and database solution's scalability, performance, and dependability. For more information, please visit: https://www.inesctec.pt/en/people/joao-tiago-paulo.