EZIOTracer: Unifying Kernel and User Space I/O Tracing for Data-Intensive Applications

Mohammed Islam Naas
University of Western Brittany
France
MohammedIslam.Naas@univ-brest.fr

François Trahay
Télécom SudParis, Institut Polytechnique de Paris
France
francois.trahay@telecom-sudparis.eu

Alexis Colin
Télécom SudParis, Institut Polytechnique de Paris
France
alexis_colin@telecom-sudparis.eu

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

Stéphane Rubini
University of Western Brittany
France
Stephane.Rubini@univ-brest.fr

Frank Singhoff
University of Western Brittany
France
Frank.Singhoff@univ-brest.fr

Jalil Boukhobza
ENSTA Bretagne
France
jalil.boukhobza@ensta-bretagne.fr

Abstract

Tracing is a popular method for evaluating, investigating, and modeling the performance of today's storage systems. Tracing has become crucial with the increase in complexity of modern storage applications/systems, that are manipulating an ever-increasing amount of data and are subject to extreme performance requirements. There exists many tracing tools focusing either on the user-level or the kernel-level, however we observe the lack of a unified tracer targeting both levels: this prevents a comprehensive understanding of modern applications' storage performance profiles. In this paper, we present EZIOTracer, a unified I/O tracer for both (Linux) kernel and user spaces, targeting data intensive applications. EZIOTracer is composed of a userland as well as a kernel space tracer, complemented with a trace analysis framework able to merge the output of the two tracers, and in particular to relate user-level events to kernel-level ones, and vice-versa. On the kernel side, EZIOTracer relies on eBPF to offer safe, low-overhead, low memory footprint, and flexible tracing capabilities. We demonstrate using FIO benchmark the ability of EZIOTracer to track down I/O performance issues by relating events recorded at both the kernel and user levels. We show that this can be achieved with a relatively low overhead that ranges from 2% to 26% depending on the I/O intensity.

CCS Concepts: • **Information systems → Storage management**; • **Computer systems organization → Secondary storage organization**; • **Software and its engineering → Secondary storage**; **File systems management**.

Keywords: Storage system, tracing tools, Linux I/O stack, Kernel, eBPF

ACM Reference Format:
Mohammed Islam Naas, François Trahay, Alexis Colin, Pierre Olivier, Stéphane Rubini, Frank Singhoff, and Jalil Boukhobza. 2021. EZIOTracer: Unifying Kernel and User Space I/O Tracing for Data-Intensive Applications. In *Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems (CHEOPS '21), April 26, 2021, Online, United Kingdom.* ACM, 11 pages. https://doi.org/10.1145/3439839.3458731

CHEOPS '21, April 26, 2021, Online, United Kingdom

ACM ISBN 978-1-4503-8302-8/21/04…$15.00
https://doi.org/10.1145/3439839.3458731

1 Introduction

In the era of Big data, High Performance Computing (HPC) and cloud systems, applications are manipulating an unprecedented scale of data volume. Indeed, the International Data Corporation (IDC) has estimated that the worldwide installed base of storage capacity will grow to 6.8 zettabytes (ZB) in 2020, an increase of 16.6% over 2019 [25]. In that context, a deep understanding of modern storage systems' performance is absolutely crucial to efficiently manage a massive and ever-increasing volume of data [41].

Tracing is a widely used method for evaluating, investigating, and modeling the performance of storage systems [36]. It

consists of adding software probes at instrumentation points, i.e. strategic locations in the program as well as in the storage stack. These probes are used to collect certain data (trace event) that will be analyzed to study the performance profile of a storage system.

Many tracers have been proposed. On the one hand, several trace application I/Os at the user level [1–4, 8, 13, 20, 24, 29, 37, 39, 40]. They focus on events regarding program and library I/O buffers management, lock contention, user-level I/O requests, etc. On the other hand, some tracers target the Operating System (OS) kernel [6, 7, 11, 16, 18, 21–23, 31–33, 35, 36]. They allow to collect events at various levels of the storage management stack [36, 43, 44], in order to break down the latency of application requests, to monitor in-kernel inter-application interference, caches management, etc.

As shown by the wide variety of existing tools in each of the two aforementioned categories, tracing at both the user and the kernel levels is crucial in order to fully understand the performance profile of a storage system subject to a particular application workload. In that context, we observe that there exist no tool designed to consider both levels in a unified way. Although some kernel-space tracers/frameworks [16, 18, 21–23] provide support for user-space probes that can be used to trace some particular function calls, the use of these probes requires potentially expensive application porting which is unacceptable in some situations (e.g. proprietary application which sources are unavailable). Furthermore, simply using a pair of existing tracers, one from each category, does not solve the problem because of a fundamental lack of compatibility: there is no easy way to relate events recorded at one level with events monitored at the other level (e.g. the various in-kernel block-level read requests triggered by a user-level call to `fread()`). Existing tracers also use a variety of output formats that are sometimes incompatible. We conclude that *in order to fully understand, investigate, model and optimize the complex I/O behavior of modern applications running on top of sophisticated OS storage management stacks, there is a strong need for a unified user- and kernel-level storage I/O tracer.*

To address this issue, in this paper we propose EZIOTracer, a unified I/O tracer for both kernel-space and user-space, focusing on data-intensive applications running on top of the Linux OS. EZIOTracer relies on three main components: First, *IOTracer*, a kernel-level tracing tool we developed, monitoring I/O operations at various levels of the Linux kernel I/O stack (Virtual File System, page cache, file system and block layer. Second, *EZTrace*, an existing user-level tracer [39] monitoring I/O events by intercepting calls to selected functions of various popular HPC/data-intensive libraries. Finally, *EasyTraceAnalyzer* (ETA), an existing generic trace analysis framework that we upgraded to be able to merge the traces generated by IOTracer and EZTrace into a unified trace that links I/O events from both the kernel-space and user-space.

The kernel-level tool IOTracer, makes use of the extended Berkeley Packet Filters, eBPF [22], a feature that was first added to Linux since version 3.15. eBPF can run various types of sandboxed programs, including lightweight monitoring tools, in the Linux kernel without changing the kernel source code or loading potentially unsafe kernel modules. It can be used for various application domains such as networking, security, application profiling/tracing and performance troubleshooting. IOTracer makes use of different types of probes (tracepoint, kprobe, kretprobe, etc) and leverages eBPF filtering capacities [22] in order to keep both its runtime overhead as well as its memory footprint low and configurable according to the degree of precision requested by the user: it is indeed possible to trace not only the entire system, but also to filter the tracing process online based on a PID as well as a given file or a directory.

EZTrace is an existing user-space tracer [39] that aims at generating automatically execution traces from HPC programs. EZTrace supports the recording of calls to functions of different parallel programming models (MPI, OpenMP, Pthread, and others) with a possible monitoring of the processes memory consumption. To that end, EZTrace either uses the LD_PRELOAD functionality of modern dynamic linkers, or dynamically patches the application binary [9].

ETA is a generic trace analysis framework that can read traces under different file formats like OTF2 [19], or Paje [15]. EZIOTracer relies on this framework to merge the traces generated by IOTracer and EZTrace into one unified trace. The unified trace contains and links I/O events from both the kernel- and the user-space. ETA relies on the type, the address, the size, the timestamp of the I/O request as well as the thread identifier issuing this request to merge and unify both traces.

We have tested and evaluated EZIOTracer using the FIO benchmark [10] with multiple scenarios. The experiments demonstrate EZIOTracer's ability to record in a unified trace I/O events that happen in userland and in the kernel. Analyzing the multi level events allows to detect the source of performance problems in several scenarios. The evaluation also shows that EZIOtrace has an overhead that ranges from 2% to 26% depending on the I/O intensity.

The contributions presented in this paper are the following:

- We introduce *IOTracer*, a kernel-level Linux I/O tracer leveraging eBPF for low overhead, high precision, and safety;
- We integrate IOTracer with the user-level application tracer EZTrace into a unified tracer called EZIOTracer. This integration includes a method to unify and merge the outputs of both IOTracer and EZTrace using an analysis framework named ETA;

- We evaluated EZIOTracer, and we show that EZIOTracer successfully pinpoints bottlenecks in several parallel I/O intensive scenarios using FIO.

The rest of this paper is structured as follows: Section 2 presents related works. Section 3 presents our unified tracer EZIOTracer and details each of its components showing the interactions between them. Section 4 describes the evaluation part of EZIOTracer using various use cases and studying relevant metrics. Section 5 concludes and discusses perspectives for future work.

2 Related Works

Tracing is a highly popular techniques targetting modern storage system performance, including monitoring, evaluation/comparison, performance debugging and optimization, characterization/ modelling, etc. Many tools have been proposed to trace application storage I/Os in user space [1–4, 8, 13, 20, 24, 29, 37, 39, 40]. They monitor various storage-related events from standard libraries such as calls to `read` or `fread`, as well as from highly popular libraries such as MPI. These tools can give information about high level metrics such as the time elapsed between sending and completing an I/O request, the request data size and the library I/O buffers management, etc. Nevertheless, this information is sometimes insufficient to fully understand all the intricacies leading to the complex I/O performance profiles that can be observed during tasks such as performance debugging or optimization. In the context of these user-space tools, one of the main reasons behind these issues is the lack of kernel-space information such as inter-application interference at the operating system level.

Other tools/frameworks [6, 7, 11, 16, 18, 21–23, 31–33, 35, 36] can collect I/O patterns at the kernel level and monitor such cross-application interactions, overcoming the limits of the user application level tracers mentioned above. However they suffer from the inverse problem: as they do not cover user space, it is sometimes hard to link performance issues observed in the kernel to the user space events from which they may have been triggered. This lack of unified and global user/kernel tracing coverage prevents from getting a comprehensive understanding of modern storage system complex performance profiles.

Only a few tools make it possible to trace I/O at both user and kernel levels [16, 18, 22, 23, 26, 34]. Unfortunately, these tools present several problems: First, they are often very generic and most can be qualified as a mechanism rather than a tracer, and developing a proper storage I/O tracer on top of them requires a non-negligible engineering effort. For example, tools such as exposed user-land probes (Userland Statically Defined Tracing, USDT) require to update and recompile application code, which is unacceptable in some situations (e.g. proprietary application which sources are unavailable). Second, these generic mechanisms are designed to trace many subsystems [16, 18, 22, 23] (process, memory, storage I/O, network, virtualization) rather that focusing on storage, which may increase their intrusiveness.

3 EZIOTracer Design

In this section, we present the design of the EZIOTracer tool. We start by giving an overview of the tool. Then, we describe its main components. In the last part of this section, using a sequence diagram, we show the interaction/cooperation between/of the main components of EZIOTracer.

3.1 EZIOTracer Overview

Figure 1 presents an overview of EZIOTracer architecture. As shown in this figure, our system is composed of three main components. The first is *EZTrace*, a user-space tracer which traces calls from user processes to popular HPC/data-intensive libraries such as OpenMP and POSIX threads. A second component is *IOTracer*, a kernel-space tracer that traces I/O requests at each level of the Linux I/O stack. The last component is the *EasyTraceAnalyzer* (ETA), a generic analysis framework, which retrieves the traces made by the two aforementioned tracers and merges them into a single unified trace. This unified trace allows to get a global and comprehensive understanding of the application's performance profile, and can then be studied manually or automatically processed by analysis tools. For example, it can be used to detect particular user events leading to performance issues in the kernel such as lock contention triggered by inter-application interference. Other use-cases include the identification of the kernel-level mechanisms slowing down user-level request, such as cache management and disk optimization tasks. In the rest of this section, we detail each of EZIOTracer's components.

Note that in this part, we will not detail the existing user-space tracer EZTrace as it is already detailed in [39].

3.2 EZTrace

EZTrace [39] is an existing Open Source user-space tracer[1] that targets HPC/data-intensive applications. EZTrace is a modular tracer and uses plugins in order to trace functions from different libraries and parallel programming runtimes (Pthread, OpenMP, Posix I/O, etc.). The interception is realized by injecting monitoring code using the LD_PRELOAD functionality of modern dynamic linkers, see Figure 1. The loaded plugins intercept applications events and produce a trace in the OTF2 [19] format. The recorded events contain entry/exit information (eg. thread id, timestamp, parameter values) on the intercepted functions. This trace can be visualized with tools like ViTE [14], or analysed with trace analysis tools.

[1] Available as open source at https://eztrace.gitlab.io/eztrace

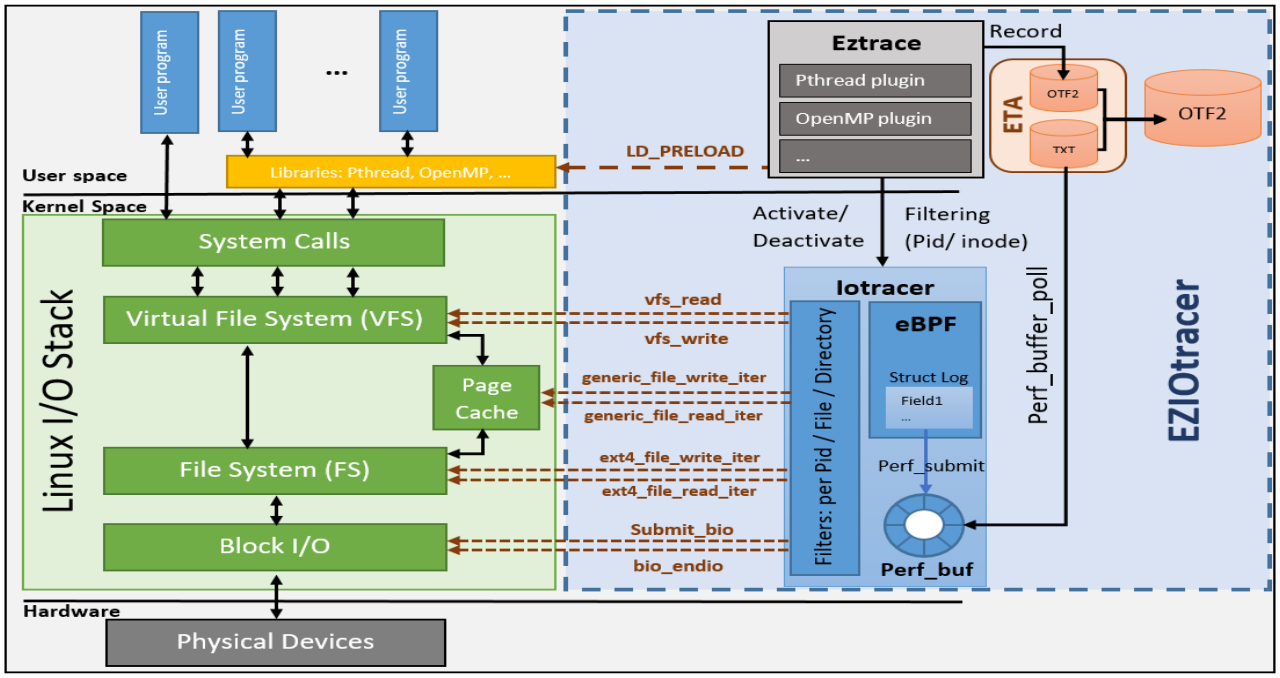


Figure 1. EZIOTracer architecture and its integration in Linux: the tracers on the left are composed of the user-space (EZTrace, grey box) and kernel (IOTracer, dark blue box) parts. The userland tracer monitors an application and uses `LD_PRELOAD` to trace particular functions calls such as POSIX I/Os. The kernel one uses eBPF to monitor calls to functions at various levels of the kernel storage stack (green boxes on the left). After the traces are collected, they are merged by ETA into a unified trace (orange box on the top right).

3.3 IOTracer

For tracing storage I/O events at various levels of the Linux kernel I/O stack, we developed IOTracer [2]. The decision to develop a new kernel-level tracer rather than use an existing one was made mainly because existing non-generic tracers are too specific. Indeed, they either focus only on a subset of the storage stack such as the block layer [11] or on a subset of storage devices such as embedded Flash [31, 32]. Also, some of these existing tracers are obsolete due to the use of depreciated tracing mechanism such as Jprobes [31–33].

Tracing frameworks [16, 18, 21–23, 35] are generic, allowing to trace events all over the kernel. IOTracer was built based on the BPF compiler collection (BCC) [5] to instrument the Linux kernel. BCC relies on the eBPF framework [22] which offers safe, low-overhead, low memory footprint and flexible in-kernel tracing capabilities.

As shown in Figure 1, IOTracer traces I/O requests at various levels over the entire Linux I/O storage stack: the Virtual File System (VFS), the page cache, the File System (FS), and the block layer. In each of these levels, IOTracer uses different probes placed on kernel functions in order to intercept I/O events, see Figure 1. The next section details each of the IOTracer probes, which are summarized in Table 1. IOTracer focuses on data read/write operations, as they represent the most performance critical operations, and it targets the common Ext4 file system [38] on block devices. However, IOTracer can easily be extended and we leave out support for further operations (metadata operations, memory-mapped I/Os, etc.), more file systems, and additional devices (such as NVMe) for future work.

[2]Available as open source at https://github.com/medislam/IOTracer

3.3.1 IOTracer Probes.

Probes are placed using eBPF/BCC on functions at various levels of the storage I/O stack, see Figure 1. Each probe has an associated handler that is executed when the function in question is called. Each handler records in the trace a timestamp and a series of parameters that are specific to the function called.

VFS Level. A user application wanting to perform a read or write operation on a file first calls the appropriate function of the system call interface. Upon reception by the kernel, this system call is then redirected to the VFS layer. The latter is the software layer in the kernel that provides the file system interface to user-space programs. It also provides an abstraction within the kernel which allows different file system implementations to coexist (e.g. block, in-memory, pseudo file systems, etc.) [27][12]. In order to monitor I/O operations at the VFS level, IOTracer places two probes, on the `vfs_write` and `vfs_read` functions (see Figure 1), in

Table 1. IOTracer probes

Probe	Event description
vfs_write	Data write operation on a file at the VFS level
vfs_read	Data read operation from a file at the VFS level
generic_file_write_iter	Data write operation in the page cache
generic_file_read_iter	Read operation from the page cache (i.e. page cache hit)
ext4_file_write_iter	Beginning of the handling of a write operation at the Ext4 File System level
ext4_file_read_iter	Beginning of the handling of a read operation at the Ext4 File System level
submit_bio	Start of a block I/O request
bio_endio	Completion of a block I/O request

order to mark the arrival of a write I/O operation or a read I/O operation at the VFS layer.

Page Cache. The page cache is a DRAM cache used by Linux mainly to speed up storage access [12]. On read request, the kernel searches for the page in the cache and serves the read directly from DRAM if the page is found (cache hit). If not, the kernel retrieves the page from the underlying block device (cache miss). Compared to a read hit, a read miss in the page cache incurs a significant latency. Upon reception of a write request, the kernel updates the page in the cache with new data. The modified page can be sent immediately to the block device, or it can remain in the dirty (modified) state in the cache, depending on the synchronization policy of the kernel.

As shown in Figure 1, IOTracer uses two probes in the page cache, `generic_file_write_iter` and `generic_file_read_iter`, to indicate respectively a write and a read I/O operations. Note that, for write operations, data are always written in the page cache before being synchronized with the disk, unless the file is opened with the `O_DIRECT` flag. Page cache misses and hits for read requests can easily be inferred from the trace by checking if a given VFS read operations triggers file system operations (cache miss) or not (cache hit).

FS Level. As previously mentioned, an I/O operation should be first served from the page cache if the data exists in the cache. Otherwise, the I/O operation will be forwarded to the concrete file system that performs several tasks of address resolution and translation, transforming a logical I/O operation (I/O issued by the application to the file system) to a physical operation intended for the lower layers of the I/O stack. These operations are issued by the file system to the block layer.

In this work we focus on Ext4 as it is a widely used Linux file system; in particular, it is the default one for many Linux distributions [17, 38]. Note that IOTracer is easily extensible to support other file systems as one would simply have to provide the file read and write functions to the tracer which corresponds to adding approximately dozen lines of code to probe a function in the new file system.

To mark the entry time of an I/O operation at the file system level, IOTracer uses two probes: `ext4_file_write_iter` and `ext4_file_read_iter` (see Figure 1) to indicate respectively the start of handling of a write operation and a read operation in the File System level.

Block Level. Once the I/O operation arrives in the block layer, the block layer creates a list of block I/O structures called BIO [30], the basic container for block I/Os within the kernel. Each structure represents an I/O operation to be submitted to disk. The BIO structures are queued and scheduled in the block layer depending to various objectives/algorithms, e.g. to minimize hard disk head movement or to increase the overall throughput by merging small requests, before redirecting I/O operations to the hardware driver. As a matter of fact, with these optimization tasks at the block level, the processing of an I/O operation could be delayed in order to optimize the overall performance of the system, for example to apply request reordering algorithms. To keep track the lifetime of an I/O operation at the block level, IOTracer uses two probes at the block layer level: `submit_bio` and `bio_endio`, see Figure 1. The former probe marks the entry time of an I/O operation (be it a write or a read operation) at the block while the latter marks its completion time.

It is worth mentioning that in the case of storage systems with a NVMe interface, the I/O operations take another path in the block layer (they bypass the scheduler) [42], and to intercept them, it is necessary to use other probes which will be added to IOTracer in the future.

After having presented the different probes used by IOTracer in each level of the Linux I/O stack, in the next part, we first detail the event recording mechanism deployed by IOTracer and then, the event filtering methods offered by IOTracer.

3.3.2 Event Recording. As aforementioned, IOTracer is based on eBPF to collect I/O events in the Linux kernel. eBPF allows three different mechanisms to transfer data from the kernel space to the user space: (1) the *perf buffer*, a kernel ring buffer supporting per event push operation from a tracer, (2) eBPF *maps*, a key-value structure useful for performing various statistics (e.g. request latencies, number of requests, etc.) and (3) the *debugfs* file system to write directly into a global file named `trace_pipe`. Note that `trace_pipe` is globally shared between several tracers, so concurrent programs will have clashing eBPF output and the reading will

be inconsistent [5]. IOTracer uses the perf buffer to record per event custom data [5]. As shown in Figure 1, using this mechanism, the captured events are saved in a custom C structure (`struct log` in Figure 1) which will be pushed to the perf buffer using the `perf_submit` function. Then, data must be fetched in user space with the `perf_buffer_poll` function. It is worth mentioning that in case of the perf buffer saturation (i.e. the frequency of events written in the buffer is higher than the frequency of reads), two methods can be employed: either ignore the new events or overwriting the oldest events with the new ones. In this work, we deploy the first method: ignore new events.

The main parameters that are recorded at each kernel level by IOTracer are: <"timestamp", "address", "size", "type", "thread id", "inode">. These parameters are used to relate kernel level events to user space ones, and vice-versa, and then to analyze the performance of the associated I/O request.

3.3.3 Filters. In order to increase the precision of the trace, to preserve the memory footprint and to minimize the runtime overhead, IOTracer leverages eBPF's filtering capabilities and offers two filtering methods: (1) filtering by PID and (2) filtering by file or directory inode. The former forces IOTracer to record only the events initialized by (a thread of) the process given by its PID. Events initialized by other processes will be ignored. The latter forces IOTracer to record events issued to a specific file if a file inode is given or events issued for all files within a directory if a directory inode is given. The two filtering methods can be combined to trace specific file (or directory) with a specific process, or separately.

One must note that IOTracer depends on the functions and symbols provided by the kernel which change from one version to another. The current design of IOTracer is made to be supported by recent version (5.10) of the Linux kernel. However, it is possible to easily adapt IOTracer to be supported by other versions of the Linux kernel by modifying the used eBPF's probes.

3.4 EasyTraceAnalyzer

EasyTraceAnalyzer (ETA) is an Open Source[3] generic trace analysis framework built in C++. ETA reads traces under different file formats like OTF2 [19], or Paje [15], and internally builds an abstract trace representation. This internal representation gives access to communications, computing units (e.g. threads, or processes) and events in a unified manner, allowing to implement trace analysis algorithms regardless of the initial trace format.

Trace Merging and Analysis. In order to analyze the whole I/O stack of an application, we extend ETA for it to be able to merge the traces generated by EZTrace and IOTracer into one trace that contains I/O events from both the user and kernel spaces. This processing is executed *post-mortem*, once the tracing is over, and it does not affect the performance of the application. To achieve the merge, ETA first identifies the execution flows that correspond to the same threads in both traces. ETA relies on a common thread identification convention used in both EZTrace and IOTracer. The user and kernel events from a thread can then be integrated in a single trace. Since EZTrace uses `clock_gettime` and IOTracer uses `bpf_ktime_get_ns`, the recorded timestamps use the same reference clock (that is the number of nanoseconds elapsed since the boot).

Merging the traces into a single execution flow boils down to applying a merge sort between two arrays, as depicted in Figure 2. The complexity of the merging of traces is thus linear with regards to the number of events. The trace resulting of EZTrace and IOTracer merging can be exported under OTF2 format for further analysis using ETA or another trace analysis suite.

3.5 EZIOTracer Components Interaction

As shown earlier, EZIOTracer involves several components: EZTrace to trace I/O operations in the user space, IOTracer to trace I/O operations at the kernel level, and ETA to merge and unify the traces made by EZTrace and IOTracer. In this part, we will show, using a sequence diagram, the interactions between the different components of EZIOTracer.

The managing component in EZIOTracer is EZTrace. As depicted in Figure 3, to use EZIOTracer, the user first launches EZTrace and specifies the filters to be used in IOTracer. EZTrace instruments the application according to the selected plugins, and it calls IOTracer while specifying the filters entered previously by the user. Then, EZTrace starts recording the I/O events of application taking place in the user space. For each intercepted I/O event, EZTrace records OTF2 events at the beginning and the end of the function call.

Meanwhile, IOTracer traces the I/O operations in the kernel space and saves them to a text file format. When the traced user application ends its execution, EZTrace stops logging I/O events and sends an `end` message to IOTracer. Then, the ETA framework (1) converts the trace made by IOTracer to OTF2 format, and then (2) merges both traces made by EZTrace and IOTracer which now use the same OTF2 format. The final unified and merged trace, containing I/O events in both user and kernel spaces for the application being traced, is returned to user for later and in-depth analysis.

4 Evaluation

In this section, we evaluate the EZIOTracer tool. We start by giving the evaluation methodology. Then, we describe the used experimental platform. Finally, we discuss the obtained results.

[3] Available as open source at https://gitlab.com/parallel-and-distributed-systems/easytraceanalyzer

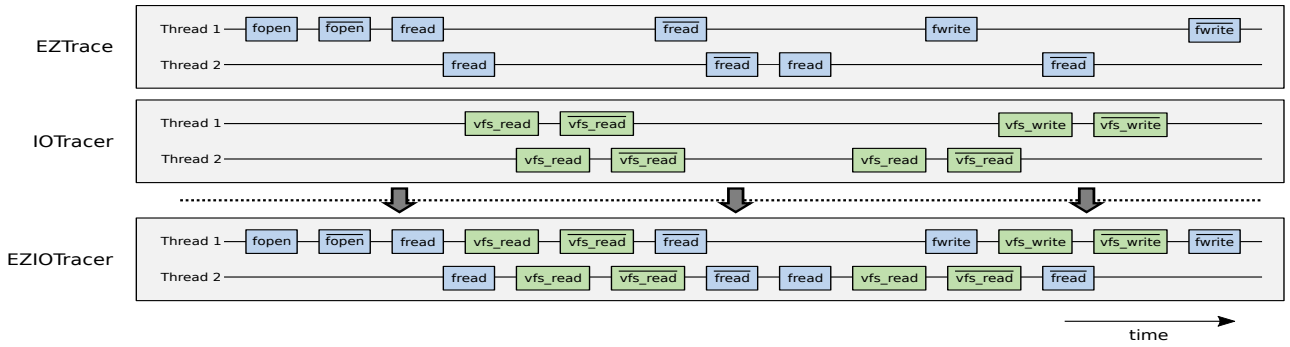


Figure 2. Merging the user- and kernel-two traces of two threads captured by EZTrace and IOTracer: timestamps from a common clock source are used to order events in the output unified trace.

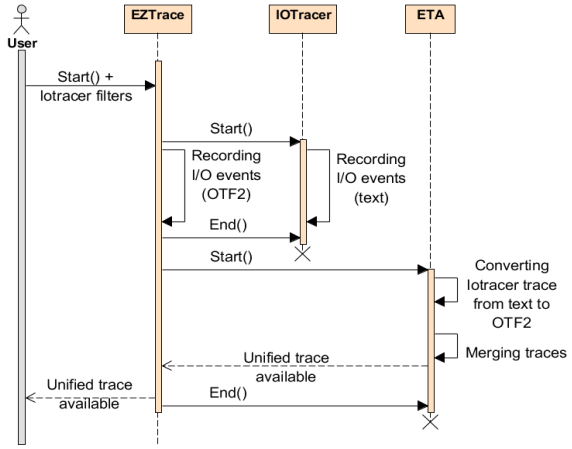


Figure 3. Sequence diagram illustrating EZIOTracer operation and execution.

4.1 Methodology

This evaluation has two goals:

1. To demonstrate the ability of EZIOTracer to *detect I/O bottlenecks* and pinpoint the source of I/O performance problems given a unified user-kernel trace recorded by EZIOTracer;
2. To evaluate the *overhead* of our tool, i.e. the performance loss due to the logging of I/O operations in both user and kernel spaces for a given application.

First, to demonstrate the usefulness of EZIOTracer in the detection of bottlenecks, we relied on the computation of the SCI score [28]. It is a theoretical estimation of the Slowdown Caused by thread Interference, which calculation method is given below in the related subsection. By analyzing the contention scores of applications with and without I/O buffer (buffered and non-buffered I/Os below), we demonstrate that thanks to the unified traces from the two levels, we were able to locate and rationally analyze the source of the contention.

Second, to measure the overhead of EZIOtracer, we compared three different software configurations:

Table 2. Experimental configuration.

	Buffered I/O	**Non-buffered I/O**
I/O block size	4 KiB	4 KiB
Number of threads	1,2,4,8,16,32,64	1,2,4,8,16,32,64
Data file size	100 MiB	10 MiB

- The *vanilla* configuration, in which the application runs without tracing;
- The *EZTrace* configuration, in which the application runs with user-space tracing only, through the use of with EZTrace. All the Posix I/O functions calls (e.g. `read`, `write`, `fread`, `fwrite`, etc.) are logged;
- The *EZIOTracer* configuration, in which the application runs with user- and kernel-level tracing through the use of EZIOTracer.

In terms of traced application we used the FIO benchmark [10], following two different settings: (1) traditional buffered I/Os that leverage the Linux page cache; and (2) non-buffered I/Os which bypass the page cache and access the storage device directly through the use of the `O_DIRECT` open flag. As described in Table 2, for each setting, we vary the number of FIO threads generating I/Os to be 1, 2, 4, 8, 16, 32 and 64. Each thread writes in parallel (with the other threads) a 100 MiB data file in case of *buffered I/Os* and a 10 MiB data file in case of *non-buffered I/Os* (since the non-buffered I/O are very slow, we reduced the amount of written data to speed up the experiments), with a 4096 bytes request size using the FIO `sync` engine (basic write functions [10]) doing only sequential writes. We report the minimum, maximum, and average of 5 measurement times of each experiment. In all our experiments, all the kernel space I/O events were successfully captured: IOTracer did not miss any I/O event due to a saturation of the perf buffer.

4.2 Experimental Platform

To conduct experiments, we used a server equipped with 2 Intel Xeon Gold 5220R clocked at 2.20 GHz (turbo 4GHz)

with 24 cores/48 threads each, and 192 GiB of DRAM. The machine has two 1.2 TiB 10K RPM disks in RAID1, controller by a PERC H740P RAID controller. It runs Linux 5.10, and programs are compiled with GCC 10.2.

4.3 Detection of I/O Bottlenecks

In this part, we evaluate how EZIOTracer can be used for locating a contention problem in the I/O stack. We analyze the traces generated with the user-space only EZTrace, and then with the unified kernel- and user-space EZIOTracer. We compute the Slowdown Caused by thread Interference (SCI) score [28].

4.3.1 SCI Score Calculation. The SCI score is computed as follows: we browse the trace files and detect I/O operations as sequences of events (for instance for EZIOTracer the following sequence: `{Enter write}`, `{Enter VFS_write}`, `{Exit VFS_write}`, `{Exit write}`), and we compare the duration D_i for similar sequences in the trace. For each set of similar sequences, we consider that the fastest sequence is almost contention free, and its duration D_{min} is the duration of an I/O operation when there is no contention. This assumption is based on existing work on the SCI metric [28]. We then compute the total slowdown of the sequence caused by contention: $Slowdown = \sum_{n=1} D_n - D_{min}$.

A single thread relative slowdown is computed as the sum of all its I/O sequences' slowdowns divided by the life duration of the thread, that is the sum of I/O time and compute time of the thread. It represents the proportion of time the thread lost due to contention. Thus, a contention score of 0 means that the sequence did not suffer from contention and all the sequences have the same duration D_{min}. A contention score close to 1 means that the thread spends most of its execution time waiting due to contention. We report the top contention score of an application execution (i.e. of a trace) as the maximum score among all threads.

The use of the SCI score allows us (1) to show the ability of EZIOtracer to unify and to link I/O events from both user and kernel spaces when computing the SCI score, and (2) to detect I/O performance problems by pointing out I/O requests suffering a contention problem (I/O requests with a high SCI score).

Figures 4a and 4b report the top contention score measured when running FIO with non-buffered (Figure 4a) and buffered (Figure 4b) I/Os. As expected, the contention score is low when FIO runs with few threads, and it grows as the number of thread increases. One should notice that contention values for the two experiments are not comparable. Indeed, in the case of non-buffered I/Os, the variation of I/O times are lower than for buffered I/Os. In the latter case, the minimum contention occurs when the requests are satisfied from the main memory. Thus the difference with the case where requests are satisfied from the storage system is higher.

Table 3. Number of traced events for the 32 threads experiment.

Level	Buffered I/O (100 MiB)	Non-buffered I/O (10 MiB)
VFS	673193	80867
FS	673193	80868
Block	0	86458
User	819201	81921

The contention score, when we consider only the user-space events (EZTrace), is similar to the one measured when both user-space and kernel-space events are considered, which is consistent as the contention does not depend on the tracers used. In the next subsections, we analyze how combining I/O events from both the user and the kernel spaces allow to pinpoint the source of performance problems.

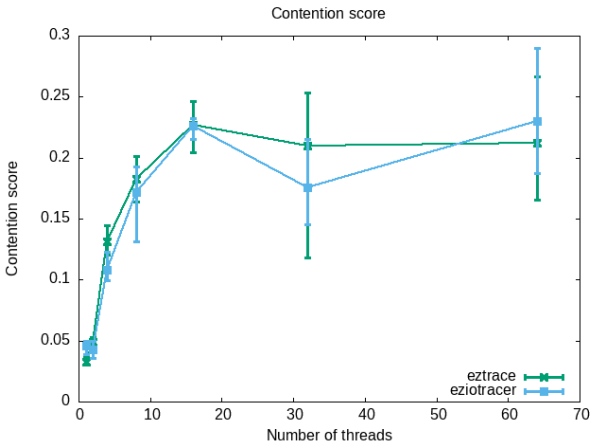
4.3.2 Performance Analysis of FIO with Non-Buffered I/Os. In order to detect the source of the contention when running FIO with non-buffered I/O, we analyze in details one set of traces obtained when running 32 threads. The EZTrace trace (that only contains user-space I/O events) shows that the threads spend up to 29% of their run time in the `write` function. Moreover, this function has a low contention score ranging from 0.02 to 0.08 for most of the 32 the threads. However, several threads have higher contention scores (up to .2). This indicates that for some threads the application suffers from moderate I/O contention. However, it does not indicate whether the source of the performance problem is in the kernel, or in user-space.

To pinpoint the performance problem, we analyzed EZIOTracer's unified trace file that contains user and kernel I/O events. It appears that all calls to `write` generate a syscall that records 80867 `VFS_write`, 80868 `FS_write`, and 86458 `BLK_write` events as shown in Table 3. Note that when multiplying the number of events by the block size (4 KiB) and dividing by the number of threads (here 32), we find the approximate file size used (10 MiB).

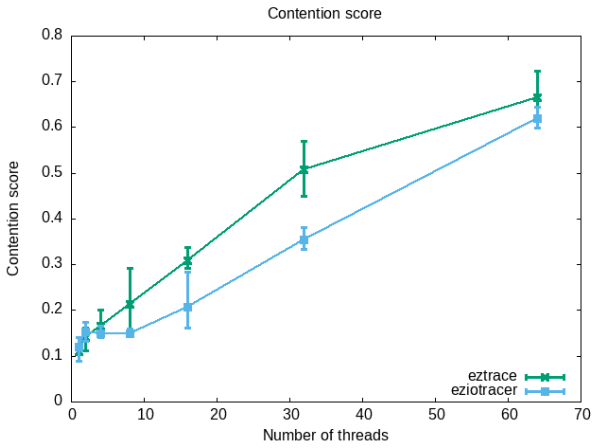
This indicates that the performance problem in this application comes from the kernel space as all I/Os are submitted to the I/O block layer. This confirms the fact that when activating the `O_DIRECT` flag, all write operations traverse the whole I/O stack without taking advantage from the page cache. This creates a contention at the kernel level when the number of I/O threads is high. The contention metric used does not show a high value as all I/Os go through the I/O stack, so the execution time difference is small.

4.3.3 Performance Analysis of FIO with Buffered I/Os. In this section, we analyze the trace files generated by EZIOTracer when running FIO with buffered I/O. We analyze in details one set of traces obtained when running 32 threads.

The EZTrace trace (application trace) shows that the I/O threads spend up to 67% of their execution in the `write`

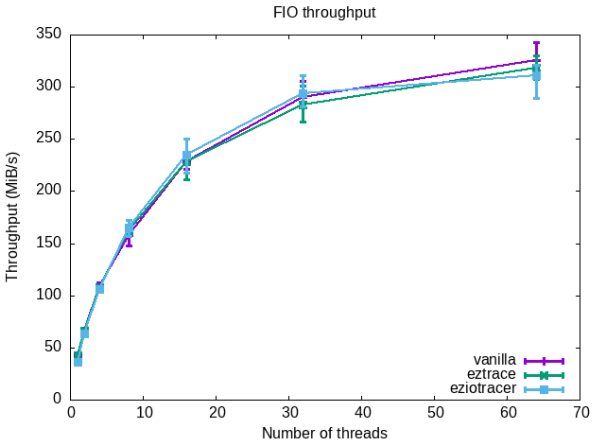


(a) Non-buffered I/O.

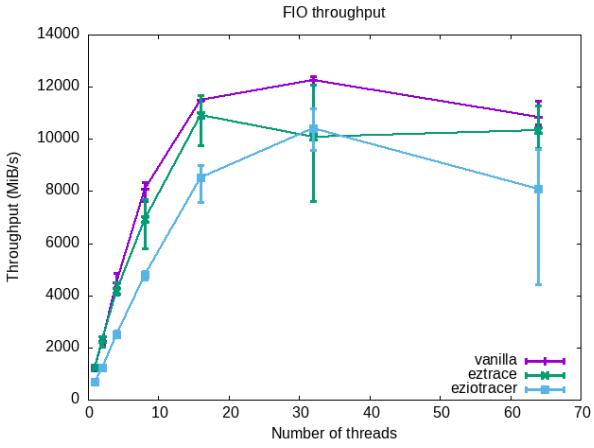


(b) Buffered I/O.

Figure 4. Evolution of FIO top contention score with the number of I/O threads.



(a) Non-buffered I/O.



(b) Buffered I/O.

Figure 5. Evolution of FIO throughput with the number of I/O threads.

function, whose contention score ranges from 0.14 to 0.34. In order to identify the source of this contention problem, we analyze the file that contains both EZTrace and IOTracer events.

In the EZIOTracer trace (that contains both user and kernel events), the number of events at the kernel level shows that even if all application level calls to `write` generate a call to a `VFS_write` and a `FS_write`, the block layer seldom generates, see Table 3. This shows that the write traffic is mostly absorbed by the page cache. Note that the difference between the number of events at the user and kernel level is mainly due to the write back policy used by the kernel which delays the writes. As a consequence, the EZIOTracer stopped the tracing before all requests traverse the kernel (this is subject to future enhancement).

4.4 Overhead of EZIOTracer

We evaluated the overhead of logging the I/O operations of an application with EZIOTracer.

We present FIO's throughput for our 3 scenarios (no tracing, user tracing only with EZTrace, unified user-kernel tracing with EZIOTracer) on Figure 5a. We observe that with non-buffered I/O, the measured throughput increases as the number of threads grows higher. For large number of threads, the maximum throughput (around 330 MiB/s) is reached and performance stops increasing. This is consistent with the measured SCI score reported in Figure 4a. The throughput grows linearly for small numbers of threads (*i.e.* when the contention is low), and it grows moderately when the number of threads is 16 or higher (*i.e.* when the contention is moderate). Instrumenting the application with EZTrace, or EZIOTracer has little effect on the measured performance:

for 64 threads, EZTrace and EZIOTracer degrade the performance by respectively 2.1%, and 4.3%.

With buffered I/O, the measured throughput quickly increases and reaches a maximum of about 12 GiB/s with 16 threads, see Figure 5b. This is consistent with the measured throughput reported in Figure 4b. For small number of threads, the throughput increases, and it stalls with 16 or more threads (*i.e.* when the SCI score becomes significant). Collecting user-space I/O events with EZTrace causes a 5% performance degradation in most cases. Instrumenting both user-space and kernel-space I/O with EZIOTracer degrades the performance by up to 26%. This is due to the overhead of instrumentation that is constant. Since buffered I/O generate many fast I/O operations, the overhead becomes noticeable.

To summarize, the experiments show that logging I/O events that occur in the user space allows to detect performance issues. However, logging both user and kernel I/O events permits to pinpoint the source of the contention in the I/O stack. It also shows that the performance degradation caused by EZIOTracer can be low to moderate depending on the application I/O intensity.

5 Conclusion

In this paper, we propose EZIOTracer, a unified I/O tracer that monitors and links I/O events in both kernel- and user-space in order to fully understand, investigate, model and optimize the complex I/O behavior of modern storage systems. EZIOTracer relies on three tools (1) EZTrace, to record calls from user processes to libraries at the user-space level; (2) IOTracer, an in-kernel eBPF-based tool we developed, tracing I/O requests at various levels of the Linux I/O stack; and (3) ETA, that we modified, to merge the traces generated by IOTracer and EZTrace into a unified trace. The experiments demonstrate EZIOTracer capacity to record the unified user-kernel performance behavior, and show that EZIOTracer has an overhead that ranges from 2% to 23% depending on the I/O intensity.

For future works, we plan to test EZIOTracer with extensive experimentation using real HPC/Big data applications with different use cases. We are currently extending EZIOTracer to support MPI jobs. Also, we plan to extend EZIOTracer to support memory-mapped I/Os tracing, other file systems types like ReiserFS, BtrFS and ZFS, and other storage interfaces like NVMe. EZIOTrace will be made available online under an open source license.

References

[1] Cloud trace web site. URL https://cloud.google.com/trace.

[2] Datadoghq web site. URL https://www.datadoghq.com/.

[3] Jackplay web site. URL https://github.com/alfredxiao/jackplay.

[4] Service pilot web site. URL https://www.servicepilot.com.

[5] Bcc github repository. URL https://github.com/iovisor/bcc.

[6] bpftrace github repository. URL https://github.com/iovisor/bpftrace.

[7] iotrace github repository. URL https://github.com/Open-CAS/standalone-linux-io-tracer.

[8] Open telemetry web site. URL https://opentelemetry.io.

[9] Charles Aulagnon, Damien Martin-Guillerez, François Rue, and François Trahay. Runtime function instrumentation with EZTrace. In *PROPER - 5th Workshop on Productivity and Performance - 2012*, August 2012.

[10] Jens Axboe. Fio-flexible io tester, 2014. URL https://fio.readthedocs.io.

[11] Jens Axboe and Alan D Brunelle. Blktrace linux manual page, 2006. URL https://man7.org/linux/man-pages/man8/blktrace.8.html.

[12] Jalil Boukhobza and Pierre Olivier. *Flash Memory Integration*. ISTE Press - Elsevier, 1st edition, 2017. ISBN 978-1-78548-124-6.

[13] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. Understanding and improving computational science storage access through continuous characterization. *ACM Transactions on Storage (TOS)*, 7(3):1–26, 2011.

[14] Kevin Coulomb, Augustin Degomme, Mathieu Faverge, and François Trahay. An open-source tool-chain for performance analysis. In *Tools for High Performance Computing 2011*, pages 37–48. Springer, 2012.

[15] Benhur de Oliveira Stein, J Chassin de Kergommeaux, and G Mounié. Pajé trace file format. Technical report, ID-IMAG, Grenoble, France, 2010.

[16] Mathieu Desnoyers and Michel R Dagenais. The lttng tracer: A low impact performance and behavior monitor for gnu/linux. In *Ottawa Linux Symposium*, pages 209–224, 2006.

[17] Borislav Djordjevic and Valentina Timcenko. Ext4 file system in linux environment: Features and performance analysis. *International Journal of Computers*, 6(1):37–45, 2012.

[18] Frank Ch Eigler. Systemtap tutorial, 2010. URL http://www.sourceware.org/systemtap/tutorial.pdf.

[19] Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E Nagel, and Felix Wolf. Open trace format 2: The next generation of scalable trace formats and support libraries. In *PARCO*, volume 22, pages 481–490, 2011.

[20] Sebastien Godard. Sysstat utilities home page. *pagespersoorange. fr/sebastien. godard*, 2010.

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

[22] Brendan Gregg. *BPF Performance Tools*. Addison-Wesley Professional, 2019.

[23] Brendan Gregg and Jim Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional, 2011.

[24] Cespedes Juan and Machata Petr. Ltrace man page. URL https://man7.org/linux/man-pages/man1/ltrace.1.html.

[25] Ljupco Kocarev and Jasna Koteska. Digital me ontology and ethics. *arXiv preprint arXiv:2012.14325*, 2020.

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

[27] Walter B Ligon III and Robert B Ross. An overview of the parallel virtual file system. In *Proceedings of the 1999 Extreme Linux Workshop*, 1999.

[28] Mohamed Said Mosli Bouksiaa, François Trahay, Alexis Lescouet, Gauthier Voron, Remi Dulong, Amina Guermouche, Elisabeth Brunet, and Gaël Thomas. Using differential execution analysis to identify thread interference. *IEEE Transactions on Parallel and Distributed Systems*, 30 (12):2866–2878, December 2019.

[29] Adrian Munera, Sara Royuela, Germán Llort, Estanislao Mercadal, Franck Wartel, and Eduardo Quiñones. Experiences on the characterization of parallel applications in embedded systems with extrae/paraver. In *49th International Conference on Parallel Processing-ICPP*, pages 1–11, 2020.

[30] Brown Neil. A block layer introduction part 1: the bio layer, 2017. URL https://lwn.net/Articles/736534.

[31] Pierre Olivier, Jalil Boukhobza, and Eric Senn. Flashmon v2: Monitoring raw nand flash memory i/o requests on embedded linux. *Acm Sigbed Review*, 11(1):38–43, 2014.

[32] Pierre Olivier, Jalil Boukhobza, Mathieu Soula, Michelle Le Grand, Ismat Chaib Draa, and Eric Senn. A tracing toolset for embedded linux flash file system. In *Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools*, pages 153–158, 2014.

[33] Hamza Ouarnoughi, Jalil Boukhobza, Frank Singhoff, and Stéphane Rubini. A multi-level i/o tracer for timing and performance storage systems in iaas cloud. In *3rd IEEE International Workshop on Real-Time and Distributed Computing in Emerging Applications (REACTION)*, 2014.

[34] Junghwan Rhee, Hui Zhang, Nipun Arora, Guofei Jiang, and Kenji Yoshihira. Uscope: A scalable unified tracer from kernel to user space. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–8. IEEE, 2014.

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

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

[37] Robert Schöne, Ronny Tschüter, Thomas Ilsche, and Daniel Hackenberg. The vampirtrace plugin counter interface: introduction and examples. In *European Conference on Parallel Processing*, pages 501–511. Springer, 2010.

[38] Rahul Shinde, Vinay Patil, Akshay Bhargava, Atul Phatak, and Amar More. Inline block level data de-duplication technique for ext4 file system. In Suresh Chandra Satapathy, P. S. Avadhani, Siba K. Udgata, and Sadasivuni Lakshminarayana, editors, *ICT and Critical Infrastructure: Proceedings of the 48th Annual Convention of Computer Society of India-Vol II*, pages 559–566, Cham, 2014. Springer International Publishing.

[39] François Trahay, François Rue, Mathieu Faverge, Yutaka Ishikawa, Raymond Namyst, and Jack Dongarra. EZTrace: a generic framework for performance analysis. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 05 2011.

[40] Karthik Vijayakumar, Frank Mueller, Xiaosong Ma, and Philip C Roth. Scalable i/o tracing and analysis. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, pages 26–31, 2009.

[41] Jun Xu, Jun Xu, and McDermott. *Block Trace Analysis and Storage System Optimization*. Springer, 2018.

[42] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance analysis of nvme ssds and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*, pages 1–11, 2015.

[43] Bin Yang, Xu Ji, Xiaosong Ma, Xiyang Wang, Tianyu Zhang, Xiupeng Zhu, Nosayba El-Sayed, Haidong Lan, Yibo Yang, Jidong Zhai, et al. End-to-end i/o monitoring on a leading supercomputer. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 379–394, 2019.

[44] Fang Zhou, Yifan Gan, Sixiang Ma, and Yang Wang. wperf: generic off-cpu analysis to identify bottleneck waiting events. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 527–543, 2018.